

Collins
Dale and
Michie

INTELLIGENCE
+
MACHINES

Edinburgh

MACHINE INTELLIGENCE 1

EDITED BY

N. L. COLLINS

DONALD MICHIE

EXPERIMENTAL PROGRAMMING
UNIVERSITY OF EDINBURGH

WITH A PREFACE BY

SIR EDWARD COLLINGWOOD F.R.S.

EDINBURGH at the University Press

© 1971 Edinburgh University Press
22 George Square, Edinburgh

First published by Oliver & Boyd
Volume 1, 1967; Volume 2, 1968

This edition, Edinburgh University Press 1971

Printed in Great Britain by
T & A Constable Ltd., Edinburgh

ISBN 0 85224 205 0

Library of Congress
Catalog Card Number
67-13648

PREFACE

Machine intelligence is a term of art which embraces a considerable range of concepts at differing levels of sophistication and arising in a variety of different contexts. Synonyms or near-synonyms are Artificial Intelligence and Mechanisation of Thought Processes. All three are suggestive terms which invite us to explore an area of science and technology which was virtually beyond our reach before the advent of the high speed electronic computer but is now seen to be full of promise and excitement. It is true that the mechanisation of logical and arithmetical operations is not a new idea, as the names of Stanhope and Babbage remind us. But the Stanhope "Demonstrator" had a built-in program of logical operations and Babbage's computer was thought of simply as a calculator. One essential ingredient of the modern notion of Machine Intelligence is the idea of a program or the combination of a mechanism and a program which has the capability to optimise its operation through time and may thus be said to "learn" from experience how to function if not faultlessly at least in an acceptable manner.

This idea carries us conceptually far beyond the simple feed-back mechanisms, like the mechanical governor and the thermostat, which have been familiar for more than a century, and their present-day and much more sensitive and complex successors. The essential difference lies in the ability to store and retrieve information in the course of the operating sequence in such a way as to optimise the outcome towards some predetermined goal. Solving a puzzle, playing a game against an opponent and controlling an industrial process are examples of thought processes which have been shown to be susceptible to mechanisation, although even the first two present formidable difficulties even in apparently simple cases. The difficulties increase with attempts to construct programs of a more sophisticated kind for such purposes as pattern recognition, language translation or theorem proving. We may expect that the more "intellectual" the task the greater the difficulty of programming an acceptable analogue of it will be.

This volume reports the proceedings of the first Annual Machine Intelligence Workshop organised by Dr Donald Michie and his Experimental

PREFACE

Programming Unit in the University of Edinburgh in September 1965. The contributions, by workers from a number of different centres of research, cover a wide range of projects in Machine Intelligence which are now being actively pursued and together give a very clear picture of the state of the art at the time of the meeting. The second Annual Workshop has already taken place and will be reported in due course. This is a field of immense potential, both theoretical and practical, and the contribution which the Workshop can make to its development by the stimulus it imparts and the publication of its proceedings is to be warmly welcomed and commended.

SIR EDWARD COLLINGWOOD, F.R.S.

CONTENTS

	PAGE
PREFACE	v
INTRODUCTION	ix
ABSTRACT FOUNDATIONS	
1. Linear graphs and trees: H. I. SCOINS	3
2. Mathematical proofs about computer programs: D. C. COOPER	17
THEOREM PROVING	
3. Beth-tree methods in automatic theorem-proving: R. J. POPPLESTONE	31
4. The resolution principle in theorem-proving: D. LUCKHAM	47
MACHINE LEARNING AND HEURISTIC PROGRAMMING	
5. Tree-searching methods with an application to a network design problem: R. M. BURSTALL	65
6. Experiments with a learning component in a Go-Moku playing program: E. W. ELCOCK and A. M. MURRAY	87
7. An approach to automatic problem-solving: J. DORAN	105
8. Complete solution of the 'Eight-Puzzle': P. D. A. SCHOFIELD	125
9. Strategy-building with the Graph Traverser: D. MICHIE	135
COGNITIVE PROCESSES: METHODS AND MODELS	
10. Networks as models of word storage: G. R. KISS	155
11. Will seeing machines have illusions? R. L. GREGORY	169
PATTERN RECOGNITION	
12. Perception, picture processing and computers: Dr M. B. CLOWES	181
13. Automatic speech recognition: a problem for machine intelligence: D. R. HILL	199

CONTENTS

PROBLEM-ORIENTED LANGUAGES

- | | |
|--|-----|
| 14. Simply partitioned data structures: the compiler-compiler re-examined:
R. A. BROOKER and J. S. ROHL | 229 |
| 15. The third-order compiler: a context for free man-machine communication:
R. B. E. NAPPER | 241 |
| 16. Principles for implementing useful subsets of advanced programming
languages: G. F. COULOURIS | 257 |
| 17. Interrogation languages: J. M. FOSTER | 267 |

INTRODUCTION

There is no more powerful attractant to the passer-by than a sign-board saying "Men at Work". Let a man begin to dig a hole in the ground, and he will before long have collected twenty onlookers. Let him be joined by nine colleagues, and the crowd swells to two hundred. So also with our scientific symposia and congresses: the perspiration of the few is converted into the inspiration of the many, and we would be foolish not to recognise how vital a part this plays in the advancement of science.

But sometimes there is a feeling that a gap exists between the large formal symposium and the laboratory *tête à tête*. It was towards this gap that we aimed the first Machine Intelligence Workshop, which provided the material for this book. Participants were as far as possible limited to active contributors in the field. The results were striking and pleasurable. Each one plunged in without inhibition; no one flagged; the end of the week found the Workshop exhausted but happy.

In a sense machine intelligence existed when the first abacus was invented, for no one would deny that even elementary arithmetic involves processes of intelligent thought. What, then, is the new factor? The vital distinction, we think, is between the machine as an aid and the machine as a prime mover. Computer aids to the thinking of the mathematician, the designer and the practical planner tend initially to be restricted to the relatively slavish roles of fact retrieval, routine computation and display of results: the *strategy* remains wholly the user's. The distinctive aim of machine intelligence within the general field of computer science is artfully to encroach on this preserve, annexing to the machine's domain more and more of the human's elusive aptitudes—his ability to handle non-numerical data structures of all kinds, to generate and apply data-description schemata, to learn from past experience to abstract features common to different problems, to devise and apply criteria of "relevance", to use approximate and intuitive arguments, to construct general rules from particular instances and to define concepts *via* examples. In their general gist these phrases from the language of everyday life are clear enough to us. Yet in reality almost everything about them is

INTRODUCTION

vague, ill-defined and subjective. To replace them with exact and unambiguous definitions as the basis of well-understood procedures engineered with known limits of error is the target on which machine intelligence research concentrates its fire.

It is a pleasure to release this first of our annual succession of salvos, and to thank Messrs. Oliver & Boyd for their finger on the trigger.

N. L. COLLINS
DONALD MICHIE

ABSTRACT FOUNDATIONS

1

LINEAR GRAPHS AND TREES

H. I. SCOINS

UNIVERSITY COMPUTING LABORATORY
NEWCASTLE UPON TYNE

CONCEPTS

The fundamental idea in useful linear graphs is the *relation*. Two typical elements x and y are assumed to be associated or related in some way so that the pair (x, y) has some significance. x and y may be similar in nature, such as towns joined by a road, or essentially different in type as in the relation 'individual x takes examination paper y '.

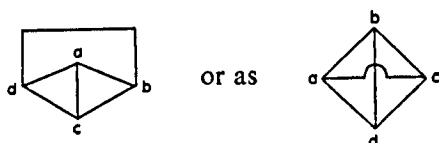
The relation may be directed or oriented so that statements like $x > y$ may be meaningful. If the background relates to collections of sets the partial ordering may be better represented by $x \in y$ or $x \subset y$. We do not necessarily associate numerical values with x and y or the relation $r(x, y)$, other than the addresses or labels which may be required for reference purposes within a computer.

A linear graph is just a collection or set of elements ($x \in E$) together with a set of relationships ($r(x, y) \in R$). It is useful to be able to draw or imagine a picture of such a set of relationships, in which the elements are represented by points or *nodes* and the relationships by lines joining the points in pairs.

Let us restrict attention to a situation where the relationships are not directed. Then the graph is *connected*, if for any two elements p, q there is at least one sequence of relations $r(p, t), r(t, u), r(u, w), \dots, r(z, q)$ which form a *path* from p to q . If a graph is connected, it is said to be multiply-connected provided that there are at least two independent paths between any two elements.

While it is convenient to think pictorially, it is also important to realise that the actual drawing of the graph on two-dimension paper contains more

properties than were originally ascribed to the graph. Thus a Wheatstone-bridge circuit (the edges of a tetrahedron) may be drawn as:



The linear graph is the same; the pictorial representation is different.

In general we cannot tell readily whether a linear graph can or cannot be drawn out on a two-dimensional piece of paper without lines intersecting or overlapping as in the second of the two diagrams above. A graph that can be drawn out without any overlap may be termed flat or *planar*. An example of a non-planar graph is the graph of the twelve edges of an octahedron, taken with the join of a pair of opposite vertices.

One class of linear graph which is necessarily planar is the *tree*. A tree is necessarily a connected graph with no closed cycles or loops, so that there is one and only one path between any two nodes of the tree. It is not difficult to show that this necessarily implies that a tree of n points necessarily contains $n - 1$ lines.

ROOTED TREES AND ORDERED TREES

The low level of specialisation of the tree defined above is not particularly useful, but one can ask and answer all sorts of interesting problems of a combinatorial nature. Perhaps the simplest is: 'How many different trees can be drawn to join n distinguishable labelled nodes?' This problem was solved by Cayley (1889-1897) and the answer is n^{n-2} . Many other problems exist in this field, there is an outstanding theorem by Polya (1937) and significant progress has recently been made by Harary (Harary & Prins 1959; Harary, Norman & Cartwright 1965) and by Good (1965), which is not of immediate interest here.

Since trees to be useful are really only a convenient representation of some other entity, we must be clear when two trees are the same. We say that two labelled trees (or graphs) are *similar* if there is a permutation of the labels which changes one tree into the other, but one must still be careful to specify in what sense we think of two trees being the same.

Two useful specialisations of the class of trees are those of *rooted* trees and *planted* trees. A rooted tree is one in which one node, termed the root, or anchor, has been given a special significance. This introduces a structural direction in the tree; up or away from the root and down or towards the root can be given a meaning.

A planted tree is a rooted tree, the root of which is directly joined to just one other node. Thus  is a rooted tree and  is a

planted (and necessarily rooted) tree. In these diagrams the root is marked with a short horizontal line.

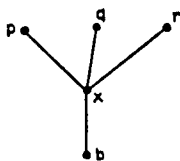
In drawing the tree on a two-dimensional piece of paper, we have built into the representation something which was not originally present. Is the above rooted tree, quite apart from any labelling of the nodes, to be con-

sidered the same as  or not?

If we take the additional property into account, we have what is best described as an *ordered* tree, in that there is a specified cyclic order of the lines about any node. In the case of the ordered and rooted tree (an ordered tree does not have to be rooted), there is a meaning to such phrases as 'move up left' or 'the node which is reached from the root by moving up the tree, taking the leftmost line at each step until there is no node higher up the tree'.

It is difficult to find any standardised language for describing these ordered trees, for it depends on how they are drawn. Family trees are upside-down when compared with the arboreal kind. However, if positive rotation is interpreted as clockwise rotation a fairly consistent language can be built up. The surprising thing is that amongst the many people using trees in the context of computing, there has been little unifying influence acting between the various fields of application.

An additional concept which may help in discussing this type of tree is the idea of a package. Suppose that we are discussing the local neighbourhood of a node x (see diagram).



If the tree is rooted, x has a node immediately below, b , unless x is the root of the tree. We might write $b = \text{below}(x)$ and can use Algol or similar language to control operations on the tree.

If the tree is not ordered, then there is nothing to distinguish the three nodes above x , i.e., those nodes which have x as below. However, if the tree is ordered, the nodes p, q, r have an ordering of left to right (facing away from the root). The package belonging to x is the subset of nodes with x as 'below' and the lines joining them to x . It is also convenient to speak of p as the *negative neighbour* and r as the *positive neighbour* of q within the package. Clearly, in our example, r has no positive neighbour and p no negative neighbour.

The package should be distinguished from the sub-tree which stands planted on b and includes all parts of the tree above and connected to x other than through b , and which may be referred to as the sub-tree of x .

FIELDS OF APPLICATION

Before detailing some of the computer representations of graphs and trees, it is as well to bear in mind the wide range of the current applications of linear graphs in practical and theoretical situations.

While the problem of electrical networks with complex impedances is one of the older examples in relation to extended computations, we find essentially the same network in the representation of a steel-framed structure under load. These problems usually involve the solution of large sets of simultaneous equations and the economisation of setting up and solving of these equations is of prime importance.

Recently critical path analysis and similar techniques have been coming into practical use in this country. Here the linear graphs are directed, usually have one beginning and one end, and must have no cycles taking the sense of direction into account. Numerous optimal flow problems, like the travelling salesman problem or the transportation problem, are to be found in the field of mathematical programming. In these two cases one is interested in finding an optimal sub-graph which spans the complete graph; in the former case we require an optimal closed cycle and in the latter case an optimal tree is to be found.

List processing either in the form of the manipulation of algebraic or symbol strings or in the form of language translation, information retrieval or computer programs is a less obvious field where graphs and particularly trees are applied. In these cases one may make a minor distinction between the case where a linear graph or tree is to be set up and then used in a static manner as in the case of a dictionary of some sort, and the case where the dominant feature is the manipulation of a linear graph according to rules so as to change one representation into another as when an algebraic expression is manipulated into a command sequence of the Reverse Polish type.

Certain sorting techniques are essentially based on the partial ordering implied by an ordered rooted tree. In fact it is this partial ordering which can be represented in many different ways that is the most important feature of a tree.

COMPUTER REPRESENTATIONS OF GRAPHS

A linear graph or a tree can always be represented by a simple list of pairs. Such a representation automatically produces an implicit labelling of the lines or relations and also introduces an orientation which may have no significance. While it is a form which is very convenient for input into a computer, it is not usually convenient as an internal representation if the structure is at all complex, or the graph is to be manipulated. For example, it would imply extensive search techniques to check in such a representation if the graph were connected or not.

Allied to this form is the *incidence* matrix, H , which is a binary matrix

with a typical element $H_{ik}=1$ if the i th node belongs to the k th line, otherwise zero. This implies a prior labelling of lines as well as nodes and will tend to be wasteful of storage in the case of large graphs. The incidence matrix is a rather specialised rectangular matrix with just two 1s in each column. It is favoured by those interested in electrical networks, particularly in the case of logical networks.

A slightly more sophisticated representation is to hold a list of adjacent nodes for each node in the graph. This is a fairly compact form, there is a certain redundancy if the graph is not directed, but is very convenient for finding paths through a graph. Storing only one address per word, the storage requirements would be $N(d+1)$ where N is the number of nodes and d the average number of nodes joined to a typical node. The extra N locations are required for access information.

Allied to this form is the binary *connection* matrix, C , in which the typical matrix element $C_{ij}=1$ if the nodes i and j are joined, and zero otherwise. For a non-directed graph, the matrix is symmetric and it can be modified to cover the case of the directed graph. It is useful for graphs where N is comparable with $d \times w$ (w =number of bits in a computer word), but for large $N \gg d \times w$, the extra space required for storage is too high a price to pay for the ease of binary manipulation assuming the computer programming is at the level of machine code.

One interesting feature of the *connection* matrix is that the element $(C^p)_{ij}$ of C^p indicates the number of distinct ways of moving from node i to node j in just p steps, and numerous elegant proofs can be based on this type of logic.

In the case of the ordinary connected linear graph, it will sometimes be convenient to separate the graph into two parts: a suitable spanning tree together with the subset of the lines not included in the spanning tree. In general the subset or *co-tree* will not itself be connected, but it is still a linear graph. The spanning tree may well be rooted on a particular node, but the introduction of ordering into such a tree would require careful consideration as the original graph cannot be 'ordered' unless it is planar, if the ordering is to have a real meaning as distinct from being an internal device to assist manipulation.

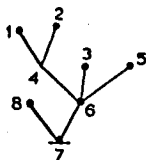
Whatever the representation of the spanning tree, one would expect a standard operation to be the exchange of a line in the *co-tree* with a line in the spanning tree in such a way as to create a new spanning tree. This is exactly the standard operation in the normal iteration sequence used in solving the transportation problem.

THE REPRESENTATION OF NON-ORDERED TREES

There does not appear to be any convenient way of representing an unlabelled, unrooted tree within a computer, but once some direction (for representation purposes only) is introduced by means of marking one node as a root, convenient representation is possible.

ABSTRACT FOUNDATIONS

For a rooted tree, with nodes labelled 1 to N , a simple representation is the ordered sequence or array of labels: below(i), provided we introduce some special rule to indicate the root, such as, the 'below' of the root is itself.



Thus the non-ordered tree in the figure can be compactly represented by the sequence

4, 4, 6, 6, 6, 7, 7, 7.

The fifth number, for example, tells us that node 6 is immediately below node 5.

Such a sequence is particularly convenient for finding a path from one node to another, but it is not very useful if there is any question which involves ascending the tree as this implies searching the array, which would be undesirable as a frequent operation in a large tree.

To find the path from, for example, node 1 to node 5 we descend the tree from 1 through the sequence 1, 4, 6, 7, 7, to the root, where each element of the sequence supplies the address of its 'below'. Descending from 5 we have the sequence 5, 6, 7, 7 again stopping on the repeated root. Ascending both sequences together we find 6 is the highest common point and the path from 1 to 5 is determined.

If the tree is very large, but the paths to be found are likely to be relatively short (the only guarantee on sequence length in a tree is that the length of the largest chain is $< N$), it may be worth carrying a generation or level number for each node which records the number of steps from a given node to the root, whose generation number is zero. In this case the two sequences of descent, if constructed together, do not need to extend below the highest common point.

This representation in terms of 'belows' has very little redundancy, although not every sequence of N numbers chosen from N will correspond to a rooted tree. Therefore we must be prepared to check that the representation does in fact correspond to a tree or, in other circumstances, to construct the representation from some other form. Such operations usually imply a limited search through the list of node pairs and allocating each of them to rooted trees which are gradually growing, or using them to graft or join trees together, or allocating them to the co-tree pool on the grounds that they form a closed loop on some already constructed tree. This implies an ability to take a rooted tree and change its root, which is simply done by reversing the direction of 'down' on that path which joins the old root to the new root.

Referring to the earlier example, if the root is to be changed to node 3, all that has to be done is form the chain down 3, 6, 7, 7 and change in reverse

sequence, the below of 7 to 6, the below of 6 to 3 and that of 3 to 3; the new representation is then 4, 4, 3, 6, 6, 3, 6, 7.

Clearly an extended array or vector of 'belows' may correspond to a large collection of trees (maybe a forest), since there is nothing in the representation to require that labels adjacent in the numeric sense should belong to the same tree.

THE REPRESENTATION OF ORDERED ROOTED TREES

The simplest form is that in which attention is restricted to a binary rooted tree. In this case each node possesses a below and may possess an up-left and an up-right. In the context of the Monkey Puzzle Sort, for example, the 'aboves' are optional, while in some forms of list processing a node either has two 'aboves' or is a terminal node at which information of a different kind is pendant.

If the main operation in making use of the tree is to ascend the tree searching for something, each node must carry the two addresses, the up-right and the up-left, or alternatively a differently marked address leading to 'information'. This latter address may be replaced by the information itself in simple cases.

In such a binary tree one can only go up the tree, but it is possible to search the tree systematically by keeping a dynamic record of the path up the tree to the current extremity. These extremities can be examined in the order of positive rotation about the root.

If this tree is to be used as a dictionary, where the pendant information is in some lexicographic order, additional information, possibly in the form of a partial key word, may be attached to each node. In this case a single run up the tree is possible to locate an entry in the dictionary, or to prove that no such entry exists.

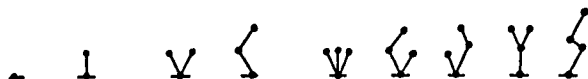
The Monkey Puzzle method of sorting involves the sorting of addresses where the information is kept and is most appropriate where each item of information to be sorted is significantly larger than the space required for three addresses. The general technique is to build a single binary tree by taking the items in their initial order and attaching them to the nodes of the tree. To add one more item to the tree, one compares it with the item at the root. If it has precedence, one moves up left, otherwise up right, and repeats the comparison with the item already attached or sitting upon the new node. This process of comparison continues until a free node is reached and the current item is attached thereto.

The specific technique of removing the information, or rather the addresses of such information, from the tree in correct order is not of importance here except that it implies moving up and down the tree in a systematic fashion. A partial ordering has been set up in constructing the tree, and the tree structure is sufficient to provide a proper ordering if we can readily move up and down within the tree and pick up information without further comparisons.

It is interesting to observe that computer hardware has had an influence here. The binary tree with information at its terminals is favoured by those who make use of a short computer word which naturally splits into two addresses, while those with access to computers with a longer word which easily splits into three, have tended to attach information to the nodes of the binary tree. This author believes that the latter situation is a hybrid and that if information is to be attached to all the nodes of a tree, it should not be restricted to being a binary ordered tree, but should be capable of having any number of above at a given node.

THE NUMBER OF ORDERED ROOTED TREES

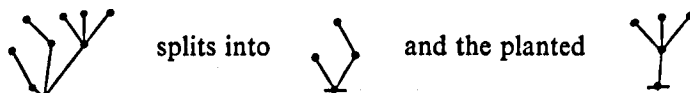
The first few trees of this type can usefully be drawn out in full:



If the number of such trees, unlabelled and distinct, with n nodes is t_n , then the sequence $t_i (i=1, 2, 3, \dots)$ is 1, 1, 2, 5, 14, 42, $\dots$ as can be observed by drawing them out. The general situation is less obvious, but is interesting. It can be shown that

$$t_n = t_1 t_{n-1} + t_2 t_{n-2} + \dots + t_{n-1} t_1.$$

Consider an ordered rooted tree of n nodes ($n \geq 2$). Then at the root there is an extreme right line, leading up to at least one further node. The package at the root may have only one line, in which case the tree is a planted tree. However, we can speak of a right-hand planted sub-tree standing on the root. Let this planted sub-tree contain $q+1$ nodes including the root ($1 \leq q \leq n-1$). If the sub-tree is removed, but the root left, we have a complementary sub-tree of p nodes ($1 \leq p \leq n-1$). For example, the tree of eight nodes



In this case $n=8$, $p=4$ and $q=4$.

Since the separation is unique, the number of trees t_n is given by

$$t_n = \sum_{p=1}^{n-1} t_p t_q \text{ where } p+q=n,$$

since any tree of q nodes can stand planted on the rightmost line. In this discussion one must distinguish carefully between the two special cases: (i) $p=1$, $q=n-1$ which corresponds to the set of planted trees with n nodes and (ii) $p=n-1$, $q=1$ which corresponds to those trees with n nodes which have a single line at the extreme right of the root.

If we define a counting series, $T(x) = \sum_{n=1}^{\infty} t_n x^n$,

then the above relationship implies that

$$T(x) = x + \{T(x)\}^2,$$

the isolated x representing the trivial tree of one node which cannot be factorised into two parts.

This expression for $T(x)$ can be formally solved and explicit algebraic expressions obtained for t_n , but these actually take longer to evaluate directly than if the recurrence relation is used.

A SIMPLIFIED BRACKETED ALGEBRAIC STRUCTURE

If we examine the written form of an Algol arithmetic expression, where brackets have been inserted to represent the priority rules, we find that every operation requires two values and one operator, with the possible exception of the unary minus as in $-a$ as distinct from $a-b$. In other words the operators are binary ordered operators, since in general they are non-commutative.

Ignoring the fact that there may be different operators, we are entitled to represent the set of possible bracketed expressions involving two operands by

$$\langle s \rangle ::= * | (\langle s \rangle \langle s \rangle)$$

where $\langle s \rangle$ represents a permitted structure and $*$ represents an indivisible operand. The Backus notation implies that a structure is an asterisk or two structures enclosed in round brackets.

The surprising thing is that this set of possible structures has the same counting series as the ordered rooted trees. The first few structures are $*$, $(**)$, $((**))$, $((**)*)$ and for order four $(((**))*)$, $((**)(**))$, $(((**))*)$, $((**(**)))$, $((**(**))*)$.

This suggests a one-one mapping of these algebraic forms on to the ordered trees, and in fact several are possible. It is probably easier to establish the one-one correspondence and so identify the counting series than to prove independently that the counting series are the same. The peculiar correspondence between the shape of the equation for $T(x)$ and the form of the Backus definition is a matter for contemplation.

The correspondence we choose to discuss here is founded on the statement that if $S_1 \leftrightarrow T_1$ and $S_2 \leftrightarrow T_2$ are structure-tree correspondences, and if $S_3 \equiv (S_1 S_2)$, then T_3 is formed by adding a single line at the right of the root of T_1 and attaching T_2 to the upper end of this line by the root. If S_1 is simply $*$, T_1 is a single node and $(S_1 S_2)$ corresponds to T_2 planted on a stalk. Thus

$$(**) \leftrightarrow \begin{array}{c} \diagup \\ \diagdown \end{array} \quad \text{and} \quad (***) \leftrightarrow \begin{array}{c} \diagup \\ \diagdown \\ \diagup \end{array}$$

while

$$\begin{array}{c} \diagup \\ \diagdown \\ \diagup \end{array} \leftrightarrow (S_1 S_2) \quad \text{where}$$

$$S_1 \leftrightarrow \begin{array}{c} \diagup \\ \diagdown \end{array} \quad \text{and} \quad S_2 \leftrightarrow \begin{array}{c} \diagup \\ \diagdown \\ \diagup \end{array}$$

and by similar recursive argument

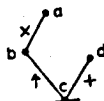
$$S_1 \equiv ((**))*, S_2 \equiv (((**))**) \quad \text{and} \quad S_3 \equiv (((**))**)((**))*).$$

ABSTRACT FOUNDATIONS

In establishing this correspondence we are enabled to see two completely different representations of what are essentially equivalent mathematical forms. If we return to the general algebraic expression, we have to insert fundamental operands for the asterisks, or attach them to the nodes, while the operator has to be inserted between the structures or attached to line joining the corresponding trees. Thus, for example, $a \times b \uparrow c + d$ after applying precedence rules becomes $((a \times (b \uparrow c)) + d)$ which has the structure $((**(**))*)$, the ordered rooted tree:



and if we insert the operands and operators as labels this becomes



This is not put forward as a useful representation if the object is simply to evaluate the expression, but is meant to illustrate the fundamental correspondence between the bracketed structure and the ordered rooted tree.

One may prefer to reflect the rooted trees before establishing the correspondence.

THE ORDERED ROOTED BINARY TREE

If one considers the set of possible binary rooted ordered trees with N terminals (excluding the root) in which every node has two 'aboves', or none, we find that this possesses the same counting series as that already discussed.

The mapping of one of these binary trees into a bracketed structure of asterisks is relatively straightforward, since it does not require a bias to the right as in the earlier discussion. If B is used to represent one of these binary trees, then the combination rule is simple: if $B_1 \leftrightarrow S_1$ and $B_2 \leftrightarrow S_2$ and $S_3 \equiv (S_1 S_2)$ then B_3 is formed by planting B_1 and B_2 on the same and additional root, with B_2 to the right of B_1 . Thus, comparing with our previous example

$$(**(**)) \leftrightarrow \begin{array}{c} \diagup \quad \diagdown \\ \diagup \quad \diagdown \end{array}, \quad ((**(**)) \leftrightarrow \begin{array}{c} \diagup \quad \diagdown \\ \diagup \quad \diagdown \end{array}$$

while

$$S_1 \leftrightarrow \begin{array}{c} \diagup \quad \diagdown \\ \diagup \quad \diagdown \end{array} \quad \text{and} \quad S_2 \leftrightarrow \begin{array}{c} \diagup \quad \diagdown \\ \diagup \quad \diagdown \end{array}$$

It is not so obvious how to establish a direct mapping of the binary tree directly into the multi-furcating tree with information attached to all nodes, but once the pattern is established, conversion is not difficult in either direction.

DERIVATIVE TREES

Given an ordered rooted tree, each node can have a 'positive successor' defined as:

- (i) the positive neighbour of the node in the sense of 'rooted trees and ordered trees' (p. 5) if it has one, or
- (ii) as the positive successor of the below if it has one, or,
- (iii) in the case of the root, itself.

A new tree can be constructed in which the below relationship of the new tree is the positive successor relationship of the old tree. This requires the addition of certain diagrammatically obvious ordering relations, but an example will adequately illustrate the point. Taking a labelled tree of six nodes *a, b, c, d, e, f* the solid lines indicate the original tree and the dotted lines the derived tree:



The derived (ordered) tree is repeated for clarity, and in this case is structurally the reflection of its antecedent, neglecting the labelling.

This operation is remarkable in that it has an inverse and forms a cyclic group within the set of ordered trees with given number of nodes. The length of the cycle depends on whether the operation is related to a labelled or unlabelled tree, but provides an interesting grouping into cycles of the ordered trees.†

A GENERAL REPRESENTATION

The simplest representation we have discussed using the array of belows was inconvenient to use in some circumstances but compact. The usual representation of the binary tree was specially designed for ascent and it implied information attached to the terminals. It would appear that a useful representation of a multi-furcating tree is to represent it by means of a $2 \times N$ array, which carries two addresses and two signs for each node. Although the representation is complex it has proved to be a useful representation which can be readily manipulated in Algol, but is also appropriate to more sophisticated handling.

If *x* is a typical node we have already defined positive neighbour, where it exists. The pord or, in full, positive neighbour or down, of *x* is defined as +(the positive neighbour) or if it has none then -(the below of *x*).

† These ideas were communicated to the author by Dr Holt of Princeton University, over beer, during the IFIP Congress in Munich in 1962. Time, space and circumstances may have modified the ideas.

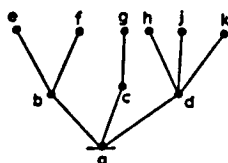
ABSTRACT FOUNDATIONS

The uplth or, in full, *up left* or *thread* of x , is defined as $+$ (the up left of x taking the first member of the package of x) or, if there is none such, $-$ (the thread of x).

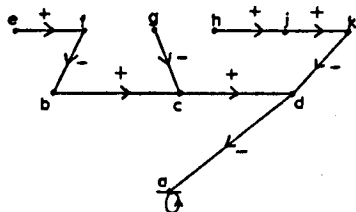
In deciding what would be a suitable definition for $\text{thread}(x)$, several possibilities have been tried, but the most satisfactory has proved to be: descend the tree from x until a node is reached which has an up-left other than the node just passed. This is then $\text{thread}(x)$. The thread of the top left extremity of the whole tree is the root and the thread of the root is itself if it requires definition.

If we consider the various possibilities, we observe that $\text{thread}(x)$ or $\text{below}(x)$ really represents an un-ordered tree, $\text{pord}(x)$ and $\text{uplth}(x)$ represent un-ordered trees with signs attached to the lines, while positive $\text{neighbour}(x)$ and $\text{upleft}(x)$ are incomplete in their structure.

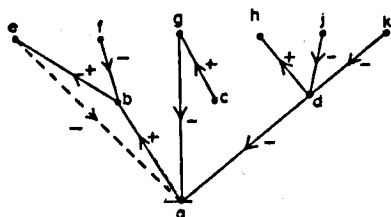
To illustrate the representation, consider the ordered labelled tree together with its associated pord and uplth , which are not themselves ordered.



ordered rooted tree



pord of above tree
(directed towards root)



uplth of same
(directed to top right and thence to root)

The example makes the fine point that the relationship between the original root and the top-left node is a special relationship in the signed tree which is formed by uplth.

Although the information carried in the pord-uplth representation seems to be partly redundant in the case of any specific operation or application, it seems to provide a possible universal background for describing algorithms involving operations on ordered, rooted trees.

REFERENCES

- Cayley, A. (1889-1897). *Collected Mathematical Papers*. Cambridge: University Press.
- Good, I. J. (1965). The generalisation of Lagrange's expansion and the enumeration of Trees. *Proc. Camb. phil. Soc. math. phys. Sci.* 499-517, **61**.
- Harary, F., Norman, R. Z., & Cartwright, D. (1965). *Structural Models: an introduction to the theory of directed graphs*. New York: Wiley.
- Harary, F., & Prins, G. (1959). The number of homeomorphically irreducible trees, and other species. *Acta math., Stockh.*, **101**, 141-162.
- Polya, G. (1937). Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und chemische Verbindungen. *Acta math., Stockh.*, **68**, 145-253.
- Riordan, J. (1958). *An introduction to combinatorial analysis*, chapter 6. New York: Wiley.

2

MATHEMATICAL PROOFS ABOUT COMPUTER PROGRAMS

D. C. COOPER

CARNEGIE INSTITUTE OF TECHNOLOGY
MASSACHUSETTS

PROOFS CONCERNING PROGRAMS

The purpose of this chapter is to put forward some ideas and to describe some work which has been done with the general idea of proving in as rigorous a manner as is possible various properties of given programs. Some of the kind of things it would be nice to find out about programs are:

- primarily 1. Does the program solve the given problem?
- and also 2. Is the program equivalent to another? i.e., for all relevant data do both programs compute the same results?
- 3. Under what conditions of the input data does the program get stuck in a loop?
- 4. Given two equivalent programs which is the more efficient and by how much?
- 5. Among many other relatively simple questions, how large is the program? What facilities does it use?

Apart from questions of type 5, these kinds of questions are usually very difficult to answer, or even to know how to tackle at all and very little work has been done in the field, although interest in these kinds of questions is now arising in various places. Some of the questions in complete generality are known to be undecidable. Thus, for example, with a very plausible definition of what a program is, we know that there can be no program which answers questions 2 or 3 for all programs (see, for example, Davis 1958). This, however, does not mean that we cannot answer questions 2 and 3 for a sufficiently useful class of programs.

This chapter gives examples of work directed at questions 1, 2 and 3 only. Question 4 depends on a definition of efficiency, on which there is some theoretical work (see, for example, Hartmanis & Stearns 1965), but we will

not discuss this further. Neither will question 5 be discussed further; these are questions that are already being answered by present day compilers. Further references to work which describes the motives of research into mathematical theories of computation and its importance are McCarthy (1962, 1963) and Wang (1963).

CURRENT TECHNIQUES

All program writers now have to answer the above questions to the best of their ability, as they must try to convince themselves that the program produced solves the given problem. The usual technique is to watch the program's performance on specially selected data, and to show that the program produces the correct answer in these cases. This in general proves very little about the program's performance on other data and all programmers can point to cases of mistakes being found in programs after the programmer had proved the program worked on his data. In this chapter we are describing work which tries to prove that programs will work on all data (even if, as is the usual case, we have an infinite or extremely large number of possible data sets). The technique of trial and error on all data sets cannot be used and we look for proofs of the correctness of programs having similar reasoning mechanisms to those used in mathematical proofs.

It is an interesting line of research to try to find problems and classes of solutions of problems which have the property that the current testing technique is, indeed, a valid proof that the program works in all cases, i.e., can we find a class of programs such that by looking at one of these programs we can deduce n data sets with the property that if the program works on these n data sets it will work on all data sets?

WHAT IS A PROBLEM?

We will now consider question 1, how do we prove that a program solves a given problem? We cannot answer this question until we know what we mean by a program, by a problem and by a proof. There are many suitable definitions of a program, both theoretical and practical, but no suitable definition of a problem. The work to be described will treat each situation individually and in each give a careful definition of what problem is being solved, and then give a proof that a given program solves the problem.

The situation is represented in Fig. 1.

The work to be described uses various languages A . Frequently a natural language such as English will be used, and at this state of the game this is perfectly satisfactory. Some proofs set up a special language in which to define the problem, but even here English has to be resorted to for a complete problem specification (e.g., restrictions on the data are added in English).

In some cases a problem can be defined by giving a correct program in some language C for solving that problem and asking for a solution in

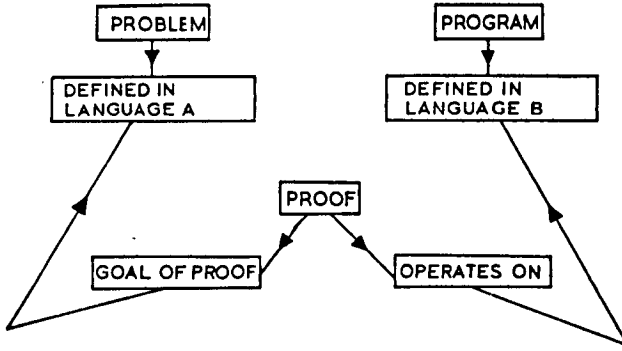


FIG. 1.

some other language D. We can then prove the correctness of this proposed solution by proving its equivalence to the language C program. We then do not need a problem specification language. We can, therefore, view question 2 of the first paragraph as a special case of question 1.

An interesting idea (see McCarthy 1962) is to take a problem as being specified when we have a technique for recognising whether a proposed solution is a solution or not. This test procedure can be specified in a programming language, so again we have no need for language A.

Referring to Fig. 2 a program, PD, with two inputs (I and O) and which always terminates with a single output which is either 'yes' or 'no' is called a problem definition. A program PS which does not loop and which has a single input I and a single output O is called a proposed solution.

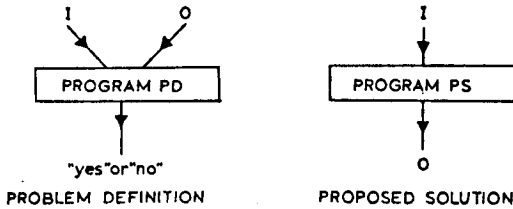


FIG. 2.

It is an actual solution of the problem if for any I the O produced by PS will, when added to I as a second input to PD, make PD produce the answer 'yes', i.e., if, with an obvious notation,

$$\text{for all } I, PD(I, PS(I)) = \text{'yes'}$$

Note that I and O may themselves be lists or some more complex data structures and that either or both will usually be restricted to being members of some given sets.

It is an arguable question as to whether all problems can be specified in this manner. The attraction of the definition is that it only involves the notion of a program for which we have good definitions. At first sight a curious feature of the definition is that if we know our problem has a solution

and if O belongs to a countable set then we can mechanically produce program PS from program PD, i.e., we can solve all problems to which solutions exist.

PROVING THINGS ABOUT COMPILERS: EVANS' WORK

Three recent reports from the Carnegie Institute of Technology describe work which involves the proof of the correctness of programs for the recognition phase of a compiler. This is that part of the compiler which takes as input the source language program, e.g., an Algol program, which recognises the syntactic structure of the program, and in the Carnegie system produces as output a string of characters in Reverse Polish (or postfix) form, i.e., it is a program which would convert string 1 into string 2.

string 1 $A := B * C + D * (E + F)$

string 2 $ABC * DEF + * + :=$

A. Evans (1964) has described this postfix notation and given an overall picture of the first Carnegie Algol compiler. Feldman (1964) has given a description of a language, called FSL, in which to write compilers including a simple method for writing recognisers using a single pushdown cell as intermediate workspace. The subset of FSL for writing the recogniser is known as Production Language and is originally due to Evans. It is beyond the scope of this chapter to describe these in any detail, and so we will restrict our discussion on proving the correctness of a recogniser to such a level that no knowledge is required of Production Language nor of the Algol compiler beyond an understanding of the tasks they have to perform.

Evans (1965), in his Ph.D. thesis, took as his aim the proving of the correctness of the recognition phase of the Carnegie Algol compiler. He did not achieve this completely but worked with subsets of the Algol language. However, there seems to be no reason why the techniques would not have worked on the full language.

The programming language (B of Fig. 1) in this case was Production Language, but, as we have already remarked, it is important to define our problem: in this case a description of the syntax of our subset of Algol and some way of describing the desired output from each string of the language. For this purpose Evans defined the problem by giving the Backus Naur Form definition of the language (see Naur 1963) and associated with each such definition a translation rule to specify the required output. For details of this, see Evans (1965); for our purpose we can regard the definition of the problem in this case as being given by a program in a language C (BNF plus translation rules) which is accepted as being correct. We then have to prove that a program in language D (Production Language) is equivalent to the language C program.

Evans takes a simple subset of Algol (consisting of arithmetic assignment statements with arbitrary arithmetic expressions on the right not involving functions or conditional expressions) and gives both the C and D programs.

Before proving the programs equivalent Evans showed that both are legal programs. In the case of the C program this meant showing that a unique answer was produced for all legal input strings (the C language as defined did not guarantee this). Evans showed that a sufficient condition for this was that the BNF grammar be unambiguous,[†] and he then proved that his BNF grammar was indeed unambiguous. This proof involved a careful definition of the concept of a properly bracketed expression, e.g., $((A+B) * (C-D))$ is properly bracketed, but $((A+B)+C$ and $(-B))+(C$ are not properly bracketed, and of a character being enclosed in brackets. These concepts were sufficient to prove that the BNF was unambiguous. The proof of the fact that the D program was a legal program meant showing that it could not get stuck in a loop. This proof was integrated with the equivalence proof which took the form of two theorems:

- (a) Given as input a legal string, the production language program will halt with the correct output string having been produced.
- (b) Given as input an illegal string (i.e., one consisting of the legal characters put together in an illegal manner such as $A := B + * C$;) the production language program will halt with ERROR as output.

Lemmas in the proof of (a) were roughly of the form: if the contents of the workspace are . . . , and if we are at label . . . in the program, then we must get to label . . . in the program; when we get there the contents of the workspace will be . . . , and we will have output. . . . These lemmas were proved by mathematical induction on the length of strings of characters.

A reverse approach has to be used to prove (b). Evans considered the set of all those strings which, when placed in the workspace and the production language program started at some specific point, will cause the program to arrive at some other given point with a given character in the workspace. He was then able to prove that these sets of strings satisfied equations very like the BNF definitions and thus prove (b).

Evans then took a more extensive subset of Algol, adding Boolean variables and expressions, and produced the same kind of proofs. These proofs were much longer and involved the investigation of a large number of cases. It would appear that the proof processes could probably be applied to the whole of the Algol compiler, but some kind of mechanisation of the work would be essential. Evans also considered the proof of the equivalence of some programs written in the production language.

PROVING THINGS ABOUT COMPILERS: EARLIER WORK

An earlier Ph.D. thesis by R. L. London (1964) took the same task as Evans: proving the correctness of a recogniser for a language defined by Backus Naur Form. However, London restricted himself to definitions using a single syntactic class such as

$$\langle W \rangle ::= 0 \mid \langle W \rangle \vee \langle W \rangle \mid (\langle W \rangle)$$

[†] This is a technical term in the theory of BNF (or context free) grammars; for its definition see Chomsky (1963) or Evans (1965).

ABSTRACT FOUNDATIONS

where $0 \vee ()$ are the characters of the language. The recogniser also has a different form to Evans' Production Language: it consists of a series of rules such as what character can be first or last, what characters can be next to each other and 'parenthesis counting rules'. For example, the following rules are necessary and sufficient to define the legal character strings belonging to $\langle W \rangle$.

1. The first character is either '0' or '('
2. The last character is either '0' or ')'
3. The next character, if any, after '0' or ')' is either ' $\vee$ ' or '('
4. The next character, if any, after ' $\vee$ ' or '(' is either '0' or '('
5. Initialise a counter to zero and scan the string from left to right. For every '(' encountered add 1. For every ')' encountered subtract 1. The count must always be non-negative and end at zero.

London proved the necessity and sufficiency of these conditions. In addition, he described a computer program which accepts as input any BNF definition (of a single syntactic class) and possibly also a set of rules similar to those above. If no rules have been given to the program a proposed set will be generated by the program. The program then proves the necessity of the rules (if they have been supplied by the user; its own rules are automatically necessary) and then goes on to the much more difficult task of either proving their sufficiency, or of producing as a counter-example a string of characters in the language but not covered by the rules. The program attempts these tasks by case analysis and mathematical induction, and is not infallible, i.e., there are BNF definitions with associated recognising rules for which the program can neither produce a sufficiency proof nor a counter-example. In his thesis London gives a proof of the validity of some of the techniques used in his program.

J. Earley (1965) described a program for automatically generating a recogniser as a Production Language program for a language defined by a set of BNF definitions. Once again this program does not work on all BNF definitions. (It cannot, as a wider class of languages can, be defined in BNF that can be recognised by a Production Language program.) He gave a proof that the techniques used in his program are valid, i.e., that if his program produces a Production Language recogniser, then that recogniser recognises the character strings defined by the BNF and only those.†

SOME THEORETICAL TECHNIQUES

J. McCarthy (1963) has described a mathematical theory of computation and in 1960 (also McCarthy 1962) he gave a technique whereby a program, given in the form of a flow chart, may be converted to a series of function definitions in his formalism. He showed how, in his formalism, starting from given functions new functions may be defined by recursive definitions involving conditional expressions.

† The proof is now known to be incorrect. Earley's algorithm appears to work on a large class of BNF definitions, but a simple definition of this class is not known.

Some examples of such definitions are:

- (a) Definition of the multiplication function, $f(m, n)$.

$$f(m, n) = \text{if } n=0 \text{ then } 0 \text{ else } m + f(m, n-1) \quad (1)$$

- (b) Definition of the factorial function, $g(n)$.

$$g(n) = \text{if } n=0 \text{ then } 1 \text{ else } n * g(n-1) \quad (2)$$

An alternative definition of the factorial function, $h(n)$.

$h(n) = k(n, 0, 1)$ where $k(n, m, a)$ is defined by

$$k(n, m, a) = \text{if } n=m \text{ then } a \text{ else } k(n, m+1, (m+1) * a) \quad (3)$$

- (c) Definition of the list function $c(x, y)$ which forms the concatenate of two lists a and b (this definition uses the LISP functions $null(x)$: true if the list x is empty, $car(x)$: the first member of the list x , $cdr(x)$: the list x minus its first member, and $cons(a, x)$: the list obtained by adding a at the front of the list x).

$$c(x, y) = \text{if } null(x) \text{ then } y \text{ else } cons(car(x), c(cdr(x), y)) \quad (4)$$

Having specified a method for defining functions, we wish to have techniques for proving relations between the defined functions. For example, with the above definitions we would like (subject to the obvious restrictions on the arguments of the functions) to prove equalities such as:

$$g(n) = h(n) \quad \text{and} \quad c(x, c(y, z)) = c(c(x, y), z)$$

McCarthy listed the following tools for proving such relations:

- I. Substitution of expressions for free variables in equations and other relations.
- II. Replacements of a sub-expression in a relation by an expression which has been proved equal to the sub-expression. In conditional expressions this equality need only have been proved under the conditions applying at the substitution point in the conditional expression. For example, in the expression

$$\text{if } p \text{ then } a \text{ else } b$$

if we have proved that, assuming p is true, $a = c$, then we may substitute c for a in the above expression.

- III. Use of certain relations satisfied by conditional expressions such as:

$$(\text{if } true \text{ then } a \text{ else } b) = a$$

$$f(\text{if } p \text{ then } a \text{ else } b) = \text{if } p \text{ then } f(a) \text{ else } f(b)$$

- IV. A rule called recursion induction for proving the equivalence of two functions. This is used for making arguments which, if the functions involved are numerical, will be made by mathematical induction. It roughly states that there can be only one function satisfying the equations involved in a recursive function definition. Exactly when an equation or set of equations constitute a legal recursive definition depends on particular arguments about the convergence of the recursive definition. For example, it is 'obvious', that because of properties of the integers, (2) is a good definition of a function if the argument of the function is a non-negative integer. Thus, defining

ABSTRACT FOUNDATIONS

this function to be the factorial function, any other function defined in any way whatsoever that can be shown to satisfy (2) must, over the non-negative integers, be equal to the factorial function.

As an example of a proof using these techniques we prove that

$$g(n) = h(n)$$

but ignore questions of the convergence of the definitions of g , k , and of the function k' defined below.

Define $k'(n, m, a)$ by

$$k'(n, m, a) = \text{if } n = m \text{ then } a \text{ else } n * k'(n-1, m, a) \quad (5)$$

Then

$$k'(n, 0, 1) = \text{if } n = 0 \text{ then } 1 \text{ else } n * k'(n-1, 0, 1)$$

But this shows that $k'(n, 0, 1)$ regarded as a function of n satisfies (2), regarded as an equation involving $g(n)$. Therefore, by recursion induction we have

$$g(n) = k'(n, 0, 1) \quad (6)$$

as indeed is obvious.

Now define k'' by the equation

$$k''(n, m, a) = \text{if } n = m \text{ then } a \text{ else } k'(n, m+1, (m+1) * a) \quad (7)$$

Note that this is not a recursive definition, and the right-hand side follows the form of (3) but with k' instead of k . Now consider (5) with k' replaced by k'' . Using (7) we get for consideration

$$\begin{aligned} &\text{if } n = m \text{ then } a \text{ else } k'(n, m+1, (m+1) * a) = \\ &\quad \text{if } n = m \text{ then } a \text{ else } n * \\ &\quad \{\text{if } n-1 = m \text{ then } a \text{ else } k'(n-1, m+1, (m+1) * a)\} \end{aligned}$$

But using (5) on the left-hand side and properties of conditional expressions and of the integers, it is easy to show that this equation is indeed true. We have, therefore, shown that k'' satisfies (5), and therefore by recursion induction

$$k''(n, m, a) = k'(n, m, a)$$

Using this in (7) we see that k' satisfies (3) for k and therefore again by recursion induction we have

$$k'(n, m, a) = k(n, m, a) \quad (8)$$

From (2), (3), (6) and (8) it is an immediate conclusion that $g(n) = h(n)$.

FURTHER THEORETICAL TECHNIQUES

In Cooper (1965) I have taken some further examples of known equivalent programs. I have expressed these programs as recursive definitions with conditional expressions and then proved the equivalences by using the techniques of the previous section.

I started with three different definitions of the factorial function (the recursive definition of (2), the iterative definition counting up of (3), and a

further iterative definition counting down). I generalised the equations by replacing the specifically numerical functions, such as multiplication, by general functions and then proved the equivalence of the three generalised definitions. A further particular case of the generalised functions is shown to be a function to reverse the order of symbols on a list. In fact, the generalised definition defines any function which is defined directly by a primitive recursive schema in terms of known functions. As a further example of proofs by recursion induction, I proved the equivalence of two programs for computing an approximate value for the definite integral of a function, one program being recursive and the other a program for the same integration technique but implemented by an iterative program.

Having produced these proofs, some of the strategies used in them were examined. This led to the discovery of a new proof rule in which many of the equivalences were an immediate consequence. This new rule was not at all applicable to some of the equivalences (in fact, to those that required the more complicated proofs) but where it was applicable it proved a powerful technique. Of course, in all these cases the new proof rule was no more powerful than recursion induction but other proofs using the new rule are known for which no proof by recursion induction has yet been found; whether the new rule is more powerful than recursion induction is not yet known.

We illustrate the rule in a particular case and use the rule to prove directly that k (3) and k' (5) are the same function. Many recursive definitions give the definition of the function at a point x in terms of its definition at another point $u(x)$; for example, frequently in numerical applications $u(x)$ is $x-1$ or in list processing applications $u(x)$ is $cdr(x)$.

Thus the recursive definition of a function f often takes the form:

$$f(x) = C(f(u(x)), x) \quad (9)$$

where C is some conditional expression.

Suppose we have a second such definition of a function g which we are trying to prove equivalent to f :

$$g(x) = D(g(v(x)), x) \quad (10)$$

Now further assume that for all relevant x

$$u(v(x)) = v(u(x)) \quad (11)$$

Then the functions f and g are the same function if and only if for all relevant x and ϕ :

$$C(D(\phi, u(x)), x) = D(C(\phi, v(x)), x) \quad (12)$$

We will not be more explicit here as to what we mean by all relevant x and ϕ .

The necessity of (12) assuming f and g are the same function can be shown as follows:

$$\begin{aligned} f(x) &= C(f(u(x)), x) && \text{by (9)} \\ &= C(g(u(x)), x) \\ &= C(D(g(v(u(x)))), u(x)), x) && \text{by (10)} \end{aligned}$$

$$\begin{aligned}
 \text{Also} \quad f(x) &= g(x) \\
 &= D(g(v(x)), x) && \text{by (10)} \\
 &= D(f(v(x)), x) \\
 &= D(C(f(u(v(x))), v(x)), x) && \text{by (9)}
 \end{aligned}$$

Now by (11) and the equivalence of f and g we have

$$f(u(v(x))) = g(v(u(x))) = \phi$$

and we have proved (12).

By a generalised form of a mathematical induction argument one can also show that if (11) and (12) are satisfied then f and g must be the same function.

It turns out that in this special case one can also give a proof by recursion induction that f and g are the same function; however, the technique generalises, and it is not easy to see how, if at all, recursion induction could be used in a more general case. One generalisation involves allowing other relations between u and v than (11); in fact, as long as there are two different compositions of the functions u and v which are equal for all x then the method applies (but with a different (12)). For example, if we had

$$u(v(v(x))) = v(u(x))$$

Another generalisation involves allowing f and/or g to be defined in terms of the function value at several other points, e.g., instead of (9) we might have

$$f(x) = C(f(u_1(x)), f(u_2(x)), f(u_3(x)), x)$$

We close this section by applying the rule to the proof that k (3) and k' (5) are the same function, first making the trivial change of regarding k and k' as functions of a single variable x which is a vector with three components, rather than being functions of three variables. The correspondence of (3) and (5) with (9) and (10) is obtained by taking

$$\begin{aligned}
 f &\equiv k, \quad g \equiv k', \quad x \equiv \{n, m, a\} \\
 u(x) &\equiv \{n, m+1, (m+1) * a\}, \quad v(x) \equiv \{n-1, m, a\} \\
 C(\phi, x) &\equiv \text{if } n=m \text{ then } a \text{ else } \phi \\
 D(\phi, x) &\equiv \text{if } n=m \text{ then } a \text{ else } n * \phi
 \end{aligned}$$

where m , n and a have to be regarded as functions of x selecting its first, second or third component.

It can easily be shown that (11) is satisfied.

(12) becomes

$$\begin{aligned}
 &\text{if } n=m \text{ then } a \text{ else (if } n=m+1 \text{ then } (m+1) * a \text{ else } n * \phi) = \\
 &\text{if } n=m \text{ then } a \text{ else } n * (\text{if } n-1=m \text{ then } a \text{ else } \phi)
 \end{aligned}$$

Since this is obviously true we have proved that k and k' are the same function.

TECHNIQUES FOR PRACTICAL PROGRAMS

In the last two sections we have considered some techniques whereby computer programs could be proved equivalent, but it is apparent that these methods could not be applied directly to programs of much greater complexity than the factorial function definitions. The technique of McCarthy for

reducing a program to the recursive function definitions is perfectly general: we can thus state the equivalences as proposed theorems, but our proof techniques are nowhere near powerful enough to be able to prove any of these theorems other than the simplest. If we are to tackle a large program we must have some way of seeing its structure as a whole, of breaking it up into its constituent parts, of proving results about the parts and then putting these results together again in order to prove facts about the complete program. The approaches of the last two sections could be used on the small parts, but seem no use if tried out directly on a large program.

Another way of putting this is to view techniques of the last section, and hopefully more powerful similar techniques, as producing deep results about small programs. What we also need are techniques for producing shallow results about large programs. By shallow results I mean results not depending in too complex a manner on the semantics of the programming language used and relations between built-in functions of the language, but rather depending on simple properties of the language and following from how these simple properties are put together in the large.

A research project has just started at Carnegie Institute of Technology to investigate this 'shallow results about real programs' aspect. This project has as its first aim the specification and programming of a scheme which accepts as input some program and gives as output conditions on the input data of this program which guarantee that it cannot get stuck in a loop. (We use the word scheme to avoid confusion with the programs that are the scheme's data.) Of course, this problem in complete generality is unsolvable and so the scheme will not work on all programs; but it is reasonable to expect that it is possible to produce such a scheme which will work on a large portion of programs produced by man. The level of program aimed at in the first instance is that produced by students after they have passed the trivial exercise stage; for example, a program to find all the moves in a game of {if in USA then checkers else draughts}.

The scheme is thought of as being a collection of *ad hoc* tests of the same nature as used by humans to prove or disprove the looping of their programs. In the first instance only proofs that can be made using the ordering properties of real numbers are envisaged. Although the scheme is thought of as a collection of *ad hoc* tests these have to be organised in some manner. We have to have a good way of thinking about programs in order to try out our techniques. Since, if we are investigating looping properties, the way flow of control moves through a program is very important, it seems natural to think about the flow chart representation of programs.

We, therefore, represent a program in the computer by storing it as a directed graph. With each node of the graph we associate predicates which depend on the current values of certain variables and which determine which arc out of the node is to be followed. With each arc of the graph we associate one or more assignment statements which change the value of some of the variables.

ABSTRACT FOUNDATIONS

Work on this project is in too early a stage to be described. Programs are available for reading in programs in a user-oriented language and converting these into the graph form described above. When a program is available to find all the simple cycles on a directed graph, we hope then to be able to classify somehow the program as to its cycle structure, e.g., 'here is a loop within a loop'. Work is also being done to find transformations which it is sensible to make on graphs, interpreted as programs, and then to see what simplification can be made in graphs by applying these transformations.

ACKNOWLEDGEMENT

The research reported here was supported by the Advanced Research Projects Agency of the U.S. Department of Defense under the Contract SD-146 to the Carnegie Institute of Technology.

REFERENCES

- Chomsky, N. (1963). Formal properties of grammars, *Handbook of Mathematical Psychology*, Vol. 2, Luce, R. D., Bush, R. R., & Galanter, E., eds., vol 2. New York: Wiley.
- Cooper, D. C. (1965). *The equivalence of certain computations*. Pittsburgh: Computation Center, Carnegie Institute of Technology.
- Davis, M. (1958). *Computability and unsolvability*. New York: McGraw-Hill.
- Earley, J. C. (1965). *Generating a recognizer for a BNF grammar*. Pittsburgh: Computation Center, Carnegie Institute of Technology.
- Evans, A. (1964). An Algol-60 Compiler. *Ann. Rev. Autom. Progrm.*, 4, 87-124.
- Evans, A. (1965). *Syntax analysis by a production language*, Ph.D. thesis. Pittsburgh: Carnegie Institute of Technology.
- Feldman, J. A. (1964). *A formal semantics for computer oriented languages*, Ph.D. thesis. Pittsburgh: Carnegie Institute of Technology.
- Hartmanis, J., & Stearns, R. E. (1965). On the computational complexity of algorithms. *Trans. Amer. Math. Soc.*, 117, 285-306.
- London, R. L. (1964). *A computer program for discovering and proving sequential recognition rules for well-formed formulas defined by a Backus normal form grammar*, Ph.D. thesis. Pittsburgh: Carnegie Institute of Technology. See also *Proc. 19th Nat. Conf. Ass. comput. Mach.*, A1, 3-7.
- McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine. *Communs Ass. comput. Mach.*, 3, 184-195.
- McCarthy, J. (1962). Towards a mathematical theory of computation. *Information processing, 1962; Proceedings of IFIP congress 1962*, pp. 21-28.
- McCarthy, J. (1963). A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*. Braffort, P., & Hirschberg, D., eds. 33-70. Amsterdam: North-Holland.
- Naur, P., ed. (1963). Revised report on the algorithmic language Algol-60. *Communs Ass. comput. Mach.*, 6, 1-17.
- Wang, H. (1963). Mechanical mathematics and inferential analysis. *Computer Programming and Formal Systems*, Braffort, P., & Hirschberg, D., eds. Amsterdam: North-Holland.

THEOREM PROVING

3

BETH-TREE METHODS IN AUTOMATIC THEOREM- PROVING

R. J. POPPLESTONE

EXPERIMENTAL PROGRAMMING UNIT
UNIVERSITY OF EDINBURGH

INTRODUCTION

This paper contains an account of a procedure for proving theorems mechanically. A description is first given of the language in which the theorems to be proved are stated (i.e., the first-order functional calculus). The axioms of group theory are then listed in this language. Next a process is defined for finding proofs of statements made in the language, and finally a specialised procedure for proving equalities is proposed and the results of some experimental tests are described and discussed.

The method of searching for a proof by means of a tree or 'semantic tableau' is due originally to Beth (1960). It is perhaps closer to Wang's (1960) approach than Robinson's (1965).

FUNCTIONAL CALCULUS

In order to prove theorems mechanically, a formal language with known syntactic rules is assumed to be necessary. The first-order language used in this paper has the following components:

- (i) symbols of fixed meaning (delimiters): $\&$, $\vee$, $\Rightarrow$, $\neg$, $\exists$, $\forall$, $=$, meaning 'and', 'or', 'implies', 'not', 'there exists', 'for all' and 'equals' respectively;
- (ii) symbols whose meaning is defined by the user of the language (these are predicates (which are properties or relations), functions, associative binary operators and variables);
- (iii) brackets serving to indicate the scope of delimiters, and of predicates, functions and operators.

THEOREM PROVING

For example, let N be a predicate meaning 'is a natural number' and let Gr be a predicate such that $Gr(x, y)$ means $x > y$ then

$$\forall x, y, z ((N(x) \& N(y) \& N(z) \& Gr(x, y) \& Gr(y, z)) \Rightarrow Gr(x, z))$$

is a statement saying that 'for all values of x, y and z , if x is a number, y is a number and z is a number, and $x > y$ and $y > z$ then $x > z$ '.

Equality could be regarded as a predicate, since it is clearly a relation between objects, but in the proof search method to be described it is treated specially, and so becomes a symbol fixed in meaning.

The associative binary operators are entities such as $+$, $\times$, $\cap$, $\cup$. These are logically equivalent to functions of two variables, but their use makes the explicit definition and application of an axiom of associativity unnecessary. Thus $a+b+c$ is a unique representation of the prefix forms plus (plus (a, b), c) and plus (a , plus (b, c)).

By a *term* we mean a well-formed combination of functions, variables and binary operators. Thus if f is a function then $f(a+b)$ is a term, as is $a+b$ or $f(x)$.

The *atomic statements* of the language consist of predicates (or equals) with terms as arguments. Finally, the statements of the language are atomic statements combined with the logical operators $\&$, $\vee$, $\Rightarrow$, $\neg$, $\forall$, $\exists$ (see (i) above).

Cursive letters $\mathcal{A} \mathcal{B} \mathcal{C} \dots$ will be used to denote statements. Thus $\mathcal{P}(x_1 \dots x_n)$ is a statement containing some of the terms $x_1 \dots x_n$.

I shall not explicitly introduce constants, as they can be regarded as functions of no variables.

Variables themselves will be used both bound by the quantifiers $\forall$ and $\exists$ and free. Thus in the statement

$$\forall x \exists y (x+y=z)$$

x and y are bound and z is free. The system is first order in the sense that quantification over functions or predicates is not allowed. Thus the axiom of induction, which states that *any property* of a number which is true for zero and whose truth for any number implies its truth for its successor, cannot be expressed within the system described here.

GROUP THEORY

The method I shall describe could be used to find proofs for any theory expressible in the first order functional calculus. I shall use Group Theory as an example. Most of the problems attempted on the computer have been from Group Theory. For an elementary account of Group Theory see Birkhoff & MacLane (1953). Group Theory will be considered as being anything that can be said in terms of a predicate G , meaning 'lies in the group', an associative binary operator for which I shall use an asterisk $*$, a function e of no variables, being the identity of the group, and a function I of one variable, being the inverse operation, more usually written $^{-1}$. The axioms these obey are:

POPPLSTONE

AG1	$\forall x, y(G(x) \& G(y) \Rightarrow G(x * y))$	closure with respect to *
AG2	$\forall x(G(x) \Rightarrow G(I(x)))$	closure with respect to inversion
AG3	$G(e)$	existence of identity
AG4	$\forall x(G(x) \Rightarrow x * e = x \& e * x = x)$	properties of identity
AG5	$\forall x(G(x) \Rightarrow x * I(x) = e \& I(x) * x = e)$	properties of inverse

Of these, AG1 and AG2, besides being closure axioms, enable the construction of the product and inverse of terms in the proof procedure to be described. There is no need for an axiom of associativity, since, as explained in 'functional calculus', this is implicit in the use of *. The closure properties of AG1 and AG2 would be useful if, say, the number axioms were to be adjoined to AG1 to AG5 in order to consider the interaction of groups and numbers. Thus, for example, we would have a function $P(x, n)$ meaning x^n and with closure axiom

$$\forall x, n(G(x) \& N(n) \Rightarrow G(P(x, n)))$$

where the predicate N means 'is a natural number'. Group Theory as understood by the mathematician may involve almost any branch of mathematics.

BETH TREES

Once having the apparatus to express mathematical statements in a symbolic language, the next step is to decide whether any such statement is a logical consequence of others. Beth (1962) developed a procedure, the growing of Beth trees (semantic tableaux), which is guaranteed to find a proof of any valid statement expressed in the first-order calculus. Other such exhaustive search procedures are known, for instance that described by Davis and Putnam (1960). Such procedures tend to involve a large amount of effort in proving even simple theorems.

A Beth tree is grown as follows. Each node of the tree consists of two lists of statements. One list can be thought of intuitively as a list of statements known to be true, and the other of statements one of which we would like to prove true. These lists are called K for 'known' and R for 'required to prove' and at the start contain the axioms and the theorem statement respectively.

Given a node (K, R) , the node or nodes directly below it (we grow our trees downwards) are produced as follows. If all statements in K and R are atomic then the next node is the same as the current one. Otherwise a non-atomic statement in K or R is selected to be operated on. This selection is performed either by a rule for selecting statements in rotation or by a heuristic device. Next, the most global logical operator of the chosen statement is found. Thus in the statement

$$(G(x) \& G(y)) \Rightarrow G(x * y)$$

$\Rightarrow$ is the most global operator. If this operator is one of the propositional operators $\&, \vee, \Rightarrow, \neg$ then Table 1 gives the rule for forming the descendent

THEOREM PROVING

node or nodes. In the next section we will consider what happens when a quantified statement is to be dealt with.

A branch of a Beth tree ends in a TIP if one of the statements in K is the same as one of those in R , that is $K \cap R \neq \phi$. If all of the branches of the tree end in tips then the statement placed in R at the root of the tree is a consequence of the axioms placed in K . Conversely, if one (or more) branch has no tip, and so is infinite, then the original statement in R is not a consequence of the axioms. In some cases it may be possible to see that a branch of a Beth tree is infinite, as for example when all statements are atomic after a certain number of nodes have been developed.

As an example, consider the statement

$$(A \Rightarrow B) \& (B \Rightarrow C) \Rightarrow (A \Rightarrow C) \quad (1)$$

where A , B and C are predicates of no variables (so that this is a statement in the propositional calculus). Fig. 1 shows a Beth tree proof that (1) is a theorem. (Since (1) is a purely logical result, there are, in this case, no axioms to place in the K -list—the axioms of logic are incorporated in the rules of the Beth tree.)

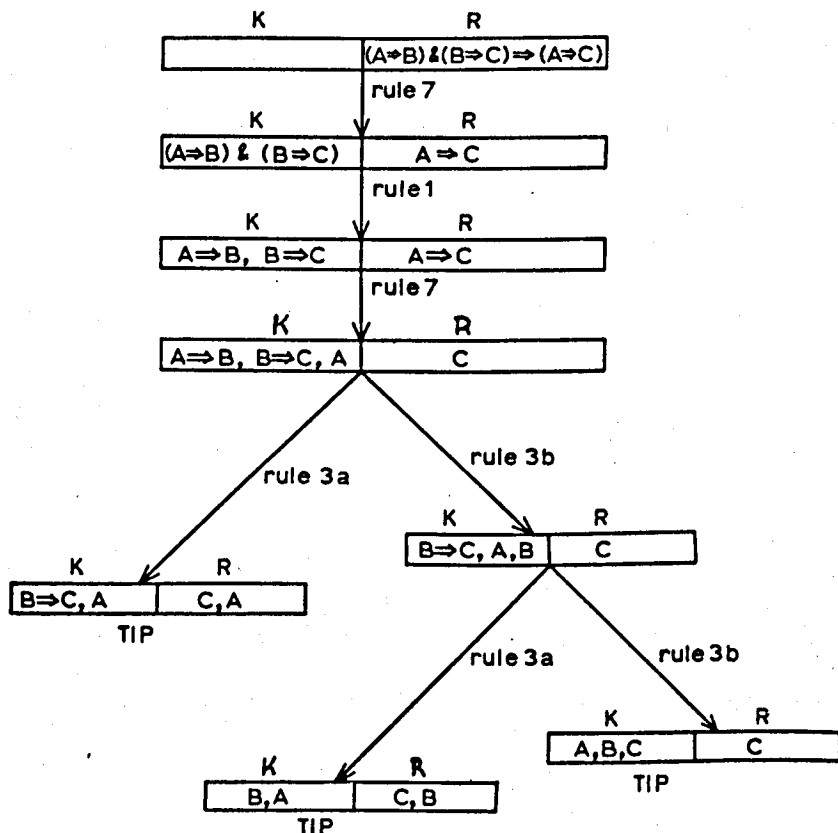


FIG. 1. A Beth-tree proof that $(A \Rightarrow B) \& (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$. The numbered rules are given in Table 1.

TABLE 1

$\&$	1	$\frac{A \& B}{A, B}$
$\vee$	$\frac{2a}{2b}$	$\frac{A \vee B}{\begin{array}{ c } \hline A \\ \hline B \end{array}}$
$\Rightarrow$	$\frac{3a}{3b}$	$\frac{A \Rightarrow B}{\begin{array}{ c } \hline A \\ \hline B \end{array}}$
$\neg$	4	$\frac{\neg A}{A}$
$\&$	$\frac{5a}{5b}$	$\frac{A \& B}{\begin{array}{ c } \hline A \\ \hline B \end{array}}$
$\vee$	6	$\frac{A \vee B}{A, B}$
$\Rightarrow$	7	$\frac{A \Rightarrow B}{\begin{array}{ c } \hline A \\ \hline B \end{array}}$
$\neg$	8	$\frac{\neg A}{A}$

Beth-tree rules for handling the propositional operators. The upper two quadrants of each 2×2 table give the state of the lists (Known and Required to prove respectively) *before* the application of the given rule, and the lower two quadrants the state *after* application of the rule.

QUANTIFIERS IN BETH TREES

The procedures for dealing with quantified expressions in Beth trees will now be discussed. Let K and R be, as before, the 'known' and 'required to prove' lists of a node of a tree.

An existentially quantified statement $\exists x \mathcal{P}(x)$ in K will be replaced by $\mathcal{P}(y)$ where y is a variable not occurring in K or R .

The handling of the case where we have a universally quantified statement $\forall x \mathcal{P}(x)$ in K is crucial to the success of any theorem proving scheme. Beth's method is to add the sentences $\mathcal{P}(t_1) \dots \mathcal{P}(t_m)$ to K , still leaving $\forall x \mathcal{P}(x)$ in K , where $\{t_1 \dots t_m\}$ are all terms occurring at that node. The point of leaving $\forall x \mathcal{P}(x)$ in K is that it can again be used on subsequent occasions to generate new statements and hence new terms. This method, while ensuring that a proof will be found, results in a very rapid growth in the size of K and R , and often in prolific branching of the tree.

For instance, if there were a numerical variable n among the terms at a given node of the tree, Beth's rules would have us add $G(n) \Rightarrow G(I(n))$ to K , which is true but irrelevant, and which leads to a split of the tree into two branches, thus doubling the work involved in finding a proof. This difficulty could be tackled by applying the rules only when one or other of the resulting branches will terminate immediately. A more satisfactory alternative is to introduce a new rule. The rule that I have used is that if one has an axiom of the form 'for all x , premise $\Rightarrow$ conclusion', and *if the premise is in K* when some term is substituted for x , then one may add the conclusion to K . Thus if $G(x * y) \in K$, one may apply AG2, which states $\forall x (G(x) \Rightarrow G(I(x)))$, and add $G(I(x * y))$ to K .

THEOREM PROVING

More formally, and more generally, suppose we have a statement in K of the form:

$$\forall x_1 \dots, x_n (\mathcal{P}_1(x_1 \dots, x_n) \& \dots \mathcal{P}_p(x_1 \dots, x_n) \Rightarrow \mathcal{Q}(x_1 \dots, x_n)) \quad (2)$$

where the $\mathcal{P}_i$ and $\mathcal{Q}$ are statements in $x_1 \dots, x_n$. Notice that this is the form of the group axioms, apart from AG3. Suppose now that θ is a mapping from the set $\{x_1 \dots x_n\}$ of variables of (2) bound by the quantifier into the set $\{t_1 \dots, t_m\}$ of free (i.e., not bound by a quantifier) terms of the current node. Suppose further that for all r from $1 \dots p$

$$\mathcal{P}_r(\theta(x_1) \dots \theta(x_n)) \in K \quad (3)$$

That is to say, each of the $\mathcal{P}_r(x_1 \dots, x_n)$ is a known formula if the term $\theta(x_i)$ is substituted for x_i for all i . Thus in some particular case all the premises of (2) are known, and so we may draw the conclusion. In other words, we add $\mathcal{Q}(\theta(x_1) \dots \theta(x_n))$ to K . This is done at one node for every θ satisfying (3), and for every statement of the type (2), the additional statements being added to the K -list of the next node down the tree, so that $\{t_1 \dots t_m\}$ remains the same throughout the operations on one node.

A simple example of the process described above is the following proof that, in a group, $e * e = e$.

K	R
AG1-AG5	$e * e = e$

There are two terms, namely e and $e * e$, in this node. Consider AG1. This states that $\forall x, y (G(x) \& G(y) \Rightarrow G(x * y))$. Thus there are four mappings from the set $\{x, y\}$ of bound variables of AG1 to the set $\{e, e * e\}$ of terms of the node.

θ_1	defined by	$\theta_1(x) = e$	$\theta_1(y) = e$
θ_2	defined by	$\theta_2(x) = e * e$	$\theta_2(y) = e$
θ_3	defined by	$\theta_3(x) = e$	$\theta_3(y) = e * e$
θ_4	defined by	$\theta_4(x) = e * e$	$\theta_4(y) = e * e$

Now $G(\theta_1(x))$ is $G(e)$ which is the axiom AG3 and so is in K . Similarly $G(\theta_1(y))$ is $G(e)$ and is in K . Thus both premises of AG1, when e is substituted for x and y , are known. So we may add the conclusion $G(e * e)$ to K . On the other hand, if we consider θ_2 , bearing in mind that we are working with the version of K not yet altered by the addition of $G(e * e)$ allowed by θ_1 , we see that $G(x)$ becomes $G(e * e)$ which is not yet in K . So θ_2 does not satisfy (2). So no addition to K results from θ_2 , or indeed from θ_3 and θ_4 . Operating on AG2, AG4 and AG5 likewise, we obtain with the aid of Table 1.

K	R
AG1-AG5, $G(e * e)$, $G(I(e))$, $e * e = e$	
$e * e = e$, $e * I(e) = e$, $I(e) * e = e$	$e * e = e$
TIP	
QED	

Gelernter's Geometry Machine (Gelernter 1960) uses, in effect, a process dual

to the above one. If on substituting for the bound variables in some axiom the statement ' $\mathcal{P} = \mathcal{Q}$ ' is formed, and $\mathcal{Q}$ is a member of R then Gelernter's machine adds $\mathcal{P}$ to R also.

In both Gelernter's machine and my own, heuristic methods of pruning the lists K and R are used.

The chief disadvantage of the Beth-tree method, as compared with Robinson's resolution principle (Robinson 1965), is that general deductions cannot be made. Thus, in a group of exponent 2 ($\forall x(G(x) \Rightarrow x * x = e$) the Beth method can deduce that for some particular x , $x = I(x)$ but not that in general $\forall x(G(x) \Rightarrow x = I(x))$. Thus the fact that, say, $x * y = I(x * y)$ would have to be proved the 'hard way' from the original quantified statement $\forall x(G(x) \Rightarrow x * x = e)$ rather than by using the general deduction $\forall x(G(x) \Rightarrow x = I(x))$. Just how severe a limitation this imposes remains to be discovered.

EQUALITY IN BETH TREES

In the example above, equality was treated as an ordinary predicate. Equality could be treated in this manner throughout, but to do so would require axioms for every predicate and function expressing the fact that equal terms can be substituted for each other. For instance, in Group Theory, axioms such as

$$\begin{aligned} \forall x, y(G(x) \ \& \ x = y \Rightarrow I(x) = I(y)) \\ \forall x, y(G(x) \ \& \ x = y \Rightarrow G(y)) \end{aligned}$$

would need to be added to AG1 to AG5, as well as axioms expressing the reflexive, symmetric and transitive properties of equality. In view of this difficulty, and of the fundamental importance of equality, it is worth introducing special techniques for dealing with this relation. The essence of the method to be described is that when statements of equality appear in the R -list, a special procedure is invoked to demonstrate this equality, using known equalities. If this is successful, the node under consideration is declared to be a TIP. Thus when it is required to prove, say, that $a * I(a) * b = b$ we use the known facts that $a * I(a) = e$, and $e * b = b$, to construct a chain of equalities $a * I(a) * b = e * b = b$. This can be expressed more precisely.

With each *node* of the Beth tree associate a *graph*. The nodes of the *graph* are all terms which can be constructed from the functions, associative binary operators and variables occurring in that node of the *tree* which is associated with the *graph*. Two graph nodes are neighbours if one can be converted into the other by substituting for one occurrence in it of any term $\mathcal{U}$ a term $\mathcal{V}$ known to be equal to $\mathcal{U}$ (i.e., ' $\mathcal{U} = \mathcal{V}$ ' $\in K$). Note that we are now using cursive letters for terms. If we know that $x * e = x$, then $x * x$ and $x * e * x$ are neighbours, but $x * x$ and $x * e * x * e$ are not. We say that the Beth tree has an 'equality tip' if $\mathcal{P} = \mathcal{Q}$ occurs in R and a path can be found from $\mathcal{P}$ to $\mathcal{Q}$ in the graph defined above.

One should notice that a path from $\mathcal{P}$ to $\mathcal{Q}$ does not necessarily exist. A path between two terms may not exist either because the two terms are

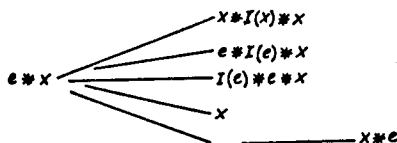
THEOREM PROVING

not in fact equal, or because insufficient deductions have been made from the axioms to prove the terms equal. In either case it is essential to be able to break off the search after a certain point.

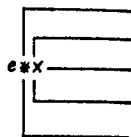
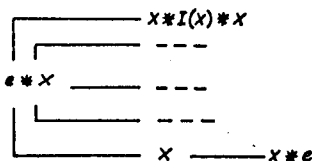
Suppose, for example, that we wish to prove that $e * x = x * e$ in a group. After some application of the group axioms, the following node of the Beth tree has been formed:

$$\begin{array}{c} K \\ G(e * x), x * I(x) = e, I(x) * x = e, e * x = x, x * e = x \\ e * I(e) = e, I(e) * e = e, e * e = e \\ R \\ e * x = x * e \end{array}$$

Exploring the graph defined above gives the following path from $e * x$ to $x * e$.



So we have an equality-tip and, the tree having no other branches, the theorem is true.



THE USE OF HEURISTICS IN PROVING EQUALITY

In this section I shall describe how the search for a path in the equality graph may be controlled. In the experiments described here, use was made of the Graph Traverser (Doran & Michie 1966, and see Doran's and Michie's papers in this volume). This is a program, written in Algol, for finding paths in graphs too large to be represented explicitly inside a computer. For my purposes it suffers from the defect of not being a list-processing program and so being incompatible with the rest of the Beth-tree program, but it has proved very valuable as a tool for demonstrating the effectiveness of heuristic methods in at least one part of theorem-proving activity. The input to this program consists of a statement of an equality that is to be proved together with a list of equalities, as produced by a Beth tree, which are to be used in the proof.

Recall that the task is to find a path from $\mathcal{P}$ to $\mathcal{Q}$, where $\mathcal{P} = \mathcal{Q}$ is a member of R , in the graph where nodes are terms, and two nodes are neighbours when they can be seen to be equal by making one application of

one known equality. Suppose the situation is as in Fig. 2. A search has been organised which has established a connexion between $\mathcal{U}$ (among other nodes generated during the search) and $\mathcal{P}$. The problem is whether to initiate the next phase of the search from $\mathcal{U}$ or whether to proceed from

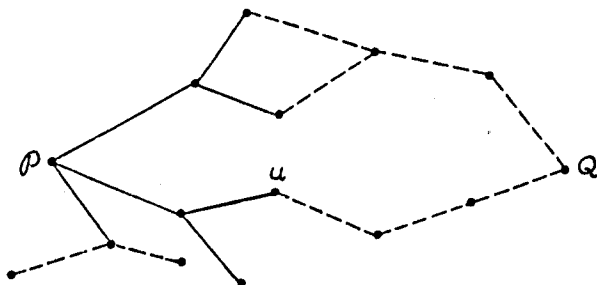


FIG. 2. A part of an equality graph. Unbroken lines denote parts of the graph already explored. Nodes represent terms, and a node can be converted into one of its neighbours by making one application of one known equality.

some other node. To make this choice we have an evaluation function F , and if $F(\mathcal{U})$ happens to be smaller than $F(\mathcal{V})$ for any other node $\mathcal{V}$ under consideration, then we 'develop' $\mathcal{U}$ in the sense of generating descendant nodes from it. A perfect evaluation function would satisfy

$$F(\mathcal{U}) \leq F(\mathcal{V}) \Leftrightarrow d(\mathcal{U}, \mathcal{Q}) \leq d(\mathcal{V}, \mathcal{Q}) \quad (4)$$

where $d(\mathcal{A}, \mathcal{B})$ is the distance between $\mathcal{A}$ and $\mathcal{B}$ in the graph.

Evaluation functions are usually combinations of 'features'. A feature f is a function which measures some property of a node. In my experiments, expressing the evaluation function as a linear combination

$$F = a_1 f_1 + a_2 f_2 + \dots + a_n f_n \quad (5)$$

of features $f_1 \dots f_n$, has proved adequate. An evaluation function of this type is analogous to Samuel's 'scoring polynomial' (Samuel 1959).

In the first experiments using the Graph Traverser three features were used. The first of these was L , where $L(\mathcal{U})$ is the length of $\mathcal{U}$, that is to say, the number of symbols contained therein. In fact, when calculating L , the program ignores $*$, but adds one, corresponding to an end-of-string marker to the length of the string as written. Thus $x * x * e * I(y)$ has a length of 8.

The second feature was E , the number of occurrences of the identity e in $\mathcal{U}$, apart from the first. This was incorporated in order to penalise nodes such as $e * e * x * e$, which are very readily formed, and which are of little use in proving any theorems other than pathological ones.

The third feature, T , was designed to measure the degree of similarity between a node $\mathcal{U}$ and the target node $\mathcal{Q}$. This was defined as the length of the longest string in $\mathcal{U}$ that was also in $\mathcal{Q}$. Thus if $\mathcal{U}$ is $I(a * b)$ and $\mathcal{Q}$ is $I(b) * I(a)$ then $\mathcal{U}$ and $\mathcal{Q}$ have the string $I(a)$ in common, and, this being the longest such string, $T(\mathcal{U}) = 3$.

THEOREM PROVING

Three problems in Group Theory were attempted using an evaluation function:

$$F=L+5E-2T$$

The results were not impressive. The machine was allowed to generate 100 distinct nodes before giving up, and only one of the problems was solved in this time. Two measures to improve the performance of the program were adopted.

The first, and simpler, was to penalise the development of heavily bracketed structures. When asked to prove that $I(b) * I(a) = I(a * b)$, for which a solution is

$$\begin{aligned} I(b) * I(a) &= e * I(b) * I(a) = I(a * b) * a * b * I(b) * I(a) \\ &= I(a * b) * a * e * I(a) = I(a * b) * a * I(a) \\ &= I(a * b) * e = I(a * b) \end{aligned}$$

the unimproved program investigated initial possibilities such as

$$I(b) * I(a) = I(e * b) * I(a) = I(a * b * I(a * b) * b) * I(a)$$

The feature adopted to measure the degree of nesting of bracketed expressions was B , defined by

$$B(\mathcal{U}) = \sum_{\substack{\alpha \in \mathcal{U} \\ \alpha \neq (,), *}} K_{\alpha}$$

where K_{α} is the number of bracket-pairs enclosing α , and $\in$ means, rather loosely, 'is a symbol occurring in'. The usefulness of B and the weight to be attached to it remains to be determined since as yet no problem has been attempted where development inside brackets is to be preferred to development outside.

The second measure was to remedy a deficiency in the feature T . T , as defined originally, regards any matching substring between a node and the target node as being a significant likeness.

T , defined thus, can be very misleading, for instance, when required to prove that $I(a) * I(b) * b * a = a * b * I(b) * I(a)$ for which a solution might run $I(a) * I(b) * b * a = I(a) * e * a = I(a) * a = e = a * I(a) = a * e * I(a) = a * b * I(b) * I(a)$.

Instead of finding the solution, the program investigated nodes such as $I(a) * a * b * I(b)$ which has a string of 6 symbols in common with the target, but which has the term $I(a)$ at the wrong end. In the light of this it seemed sensible to modify T only to take account of those matching substrings for which the non-matching parts are (i) well-formed terms and (ii) equal to each other (extending either node if necessary by $*e$ or $e*$ to avoid having to interpret the null string; we would, for example, regard the common substring $a * b * I(b)$ of the node $I(a) * a * b * I(b)$ and the target node $a * b * I(b) * I(a)$ as being significant if $I(a) = e$). Thus we would evaluate the match between the strings $\mathcal{P}_1 \mathcal{Q} \mathcal{R}_1$ and $\mathcal{P}_2 \mathcal{Q} \mathcal{R}_2$ (where $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{R}$ represent well-formed substrings) as significant if $\mathcal{P}_1 = \mathcal{P}_2$ and $\mathcal{R}_1 = \mathcal{R}_2$.

How therefore are we to decide whether these or any other two terms are

equal? It is known, since the 'word problem' in Group Theory is undecidable, that there is no certain way. We could use the Graph Traverser recursively to try and prove them equal, but this would be uncertain and time-consuming. A method that is less time-consuming (and probably less uncertain) is to calculate the values in some particular group and see if they are equal. Let us call such a group an example-group, and consider a very simple group of four elements defined by its multiplication table.

TABLE 2

*	e	a	b	c	x	I(x)
e	e	a	b	c	e	e
a	a	e	c	b	a	a
b	b	c	e	a	b	b
c	c	b	a	e	c	c

Looking up $I(a)$ in this table, we see that $I(a) = a \neq e$ and so $I(a) * a * b * I(b)$ has no significant likeness to $a * b * I(b) * I(a)$. This device is substantially the same as Gelernter's diagram (Gelernter 1960).

Provided that the example-group satisfies whatever additional assumptions may be in force about the group in the theorem, there is no possibility of equal terms being classified as unequal when calculated in the example. However, unequal terms might have equal values in the example-group. Thus in the group in Table 2 $a * b = b * a$, although this is not true in general.

In passing I might add that a whole battery of example-groups with different properties (commutative, exponent 2, nilpotent, etc.) could be useful in pinpointing those axioms which should be used in a proof. Thus if commutativity is given as an axiom and a matching of terms is significant in a commutation example-group, but not in a non-commutative example, then the Beth-tree routines might be told to apply the commutativity axiom (especially to the non-matching terms).

With the addition of the feature B , and with the modification of T discussed above, the three problems, together with a fourth, were re-run. This time all the problems were solved with reasonable competence. A measure of efficiency for Graph Traversing is the penetrance, defined as:

$$\frac{\text{number of developed nodes on path}}{\text{total number of nodes developed}}$$

The values of the penetrance were 100 per cent, 100 per cent, 85 per cent and 75 per cent.

Let us consider one of these results more closely (problem 2 of Table 3, condition PD omitted). The evaluation function used was

$$F = L + B + 5E - 3T$$

The actual printout from the run involved 77 terms and so is too lengthy to be reproduced here, but an annotated version appears below, with the missing terms replaced by comments in small print. The label $m.n$ denotes the n th node produced of the m th generation.

THEOREM PROVING

	Term	<i>L</i>	<i>B</i>	<i>E</i>	<i>T</i>	<i>F</i>
0.1	$I(b) * I(a)$	9	2	0	0	11
First descendants						
1.1	$I(b * e) * I(a)$	10	3	0	0	13
And other terms with <i>e</i> inside the brackets						
1.3	$I(b) * e * I(a)$	10	2	0	0	12
1.4	$e * I(b) * I(a)$	10	2	0	0	12
1.7	$I(b) * I(a) * e$	10	2	0	0	12

The next generation is produced from the first-generated of the three nodes with a score of 12, that is $I(b) * e * I(a)$. All the descendants of this node, apart from the starting node $I(b) * I(a)$, are longer than it, and so only those with a non-zero *T* can hope to attain a score low enough to be considered as parents of the next generation. There are however no common substrings which satisfy the criterion of significance described above.

Consequently, we have to go back to the previous generation (generation 1) for a new parent, and node 1.4 ($e * I(b) * I(a)$) is next on the list. All of its descendants save one have a zero score in *T*. The exception is

$$I(a * b) * a * b * I(b) * I(a)$$

for which the obvious match with $I(a * b)$ is significant. So the score for this node is

2.6	$I(a * b) * a * b * I(b) * I(a)$	16	4	0	5	5
-----	----------------------------------	----	---	---	---	---

So the evaluation function *F* has the value 5 for this node, and hence the node is again developed. All is now plain sailing, with a steady reduction of length at each node.

3.18	$I(a * b) * a * e * I(a)$	12	3	0	5	0
	etc.					
4.11	$I(a * b) * a * I(a)$	11	3	0	5	-1
	etc.					
5.13	$I(a * b) * e$	7	2	0	5	-6
	etc.					
6.9	$I(a * b)$	6	2	0	5	-7

While the performance of the program in these examples, as measured by the penetrance, is reasonable, the number of nodes produced during a run is unacceptably large. Thus in the last two problems of the four mentioned above, both of which had a path length of 6, the numbers of distinct nodes produced were 77 and 95 respectively. In order to try to reduce this number, a further change has been made in the program. Instead of producing all nodes which are neighbours of a node under consideration, only a few are produced. This 'partial development' is achieved by arranging the list of known equalities in an order which places first those, such as $I(a) * a = e$, which give rise to a reduction in length in a node. ($e = I(a) * a$ is entered separately from $I(a) * a = e$ in the list of known equalities.) The program then makes use of a small number of equalities, starting with the most promising, during one development of one node.

This last modification of the program has made some improvement in the performance. Over the four problems for which comparable figures are available, the total number of distinct nodes produced was reduced by this

POPPLESTONE

device from 223 to 161, and the number of nodes produced (counting repetitions) was reduced from 315 to 189.

A number of problems were attempted using this version of the program. Table 3 gives the results.

TABLE 3
Results obtained with six problems under various conditions.

Condition Omitted	Result	Nodes Listed	Nodes Developed	Inspections	Penetrance %	Path Length
PROBLEM 1. $abI(b)I(a)=e$						
—	S	7	3	6	100.0	3
L	S	11	4	12	75.0	3
B	S	7	3	6	100.0	3
E	S	7	3	6	100.0	3
T	S	7	3	6	100.0	3
PD	S	30	3	39	(100.0)	3
M	S	20	7	22	42.9	3
F	S	25	11	30	63.6	3
PROBLEM 2. $I(b)I(a)=I(ab)$						
—	S	55	19	65	63.2	6
L	S	74	22	92	54.5	6
B	S	99	31	127	38.7	6
E	S	55	19	65	63.2	6
T	F	100	32	130	—	—
PD	S	88	8	122	(75.0)	6
M	F	100	40	128	—	—
F	S	61	25	79	56.0	6
PROBLEM 3. $I(a)I(b)ba=abI(b)I(a)$						
—	S	32	16	38	75.0	6
L	FL	—	—	—	—	—
B	S	32	16	38	75.0	6
E	S	32	16	38	75.0	6
T	F	100	49	134	—	—
PD	FL	—	—	—	—	—
M	F	100	27	125	—	—
F	S	29	14	34	71.4	6
PROBLEM 4. $abI(b)I(a)ba=ba$						
—	S	9	4	8	100.0	4
L	S	17	6	19	66.7	4
B	S	9	4	8	100.0	4
E	S	9	4	8	100.0	4
T	S	9	4	8	100.0	4
PD	S	46	4	63	(100.0)	4
M	S	9	4	8	100.0	4
F	S	36	17	46	58.8	4

THEOREM PROVING

In all the following problems the additional condition $\forall x, x^2 = e$ was assumed.

Condition Omitted	Result	Nodes Listed	Nodes Developed	Inspections	Penetrance %	Path Length
-------------------	--------	--------------	-----------------	-------------	--------------	-------------

PROBLEM 5. $I(a)=a$

—	S	31	13	33	76.9	4
L	S	33	15	35	66.7	4
B	S	57	19	65	52.6	4
E	S	31	13	33	76.9	4
T	S	51	21	58	42.9	4
PD	S	42	5	46	(80.0)	4
M	S	52	22	59	40.9	4
F	S	30	14	32	64.3	4

PROBLEM 6. $ab=ba$

—	S	36	17	44	64.7	6
L	FF	100	45	151	—	—
B	S	36	17	44	64.7	6
E	S	36	17	44	64.7	6
T	F	100	56	196	—	—
PD	S	65	7	95	(85.7)	6
M	FF	100	32	135	—	—
F	S	37	18	43	61.1	6

SUMMARY

Condition Omitted	Total Successes	Medians			Mean Penetrance %
		Nodes Listed	Nodes Developed	Inspections	
—	6	31½	14	35½	79.9
L	5	53½	18½	63½	57.2
B	6	34	16½	41	71.8
E	6	31½	14½	35½	79.9
T	3	> 75½	26½	94	49.8
PD	5	55½	6	79	(88.1)
M	3	> 76	24½	92	47.4
F	6	33	15½	38½	62.5

In the 'condition omitted' column:

- denotes that the full complement of heuristic devices was in use
- L denotes that the coefficient of the feature L was set to zero
- B denotes that the coefficient of the feature B was set to zero
- E denotes that the coefficient of the feature E was set to zero
- T denotes that the coefficient of the feature T was set to zero
- PD denotes that all neighbours of a node were produced at one development
- M denotes that the example group was not in use

POPPLSTONE

- F denotes that the equality was proved in reverse, i.e., that the RHS was converted into the LHS. Note that the LHS is longer than the RHS in every problem where the lengths are different.

In the 'result' column:

- S denotes that the problem was solved
F denotes that the problem was unsolved after 100 distinct nodes had been produced
FF denotes that the problem was unsolved after 100 distinct nodes had been produced but that the problem was solved after restarting one or more times from the most promising node produced so far (see Doran's paper in this volume).
FL denotes that the problem was unsolved because a term too large to hold in the array assigned to hold it was produced.

The 'nodes listed' column gives the total number of distinct nodes produced.

The 'inspections' column gives the total number of nodes produced, counting repetitions.

Penetrance values in the rows marked PD have been bracketed, since their interpretation is somewhat different under the PD condition.

The results given in Table 3 indicate that all the heuristic devices apart from the feature E play a useful part in the solution of problems. Note the great effect produced by conditions T and M acting in combination. T is ineffective without M, and M is, of course, meaningless without T.

Two further problems were attempted, namely $I(a)I(b)=I(b)I(a)$ and $I(a)I(b)ab=e$, and neither was solved. Analysis of the results indicated that the use of a more comprehensive example group or groups might have resulted in a solution of the problems.

CONCLUSION

The Beth tree procedure described in the early part of this paper has been coded and tested on a small computer (a Stantec Computing System), and proved capable of performing manipulations of the kind described between pages 33 and 38. Unfortunately, the machine was too small (c. 700 words of list space) to permit an extensive evaluation of the Beth-tree procedure, and it has not yet proved possible to run the program on a larger machine. The results on the exploration of the equality graph, as described in 'The use of heuristics in proving equality' (p. 38), were obtained on an Elliott 803. At present, the only heuristic device attached to the Beth tree procedure is to delete those statements in the *K*-list which do not refer to nodes developed in the preceding search of the equality-graph (assuming that an equality is to be proved), thus coupling the tree-search to the equality heuristics. I feel that, with relatively small improvements in the control of tree growth, a performance in the proving of equalities in algebraic systems which is

THEOREM PROVING

mathematically useful is attainable. More difficult problems, such as existence proofs, seem to be further over the horizon.

REFERENCES

- Beth, E. W. (1962). *Formal methods*. Dordrecht-Holland: Reidel.
- Birkhoff, G., & MacLane, S. (1953) *A survey of modern algebra* (revised ed.). New York: Macmillan.
- Davis, M., & Putnam, H. (1960). A computing procedure for quantification theory. *J. Ass. comput. Mach.*, 7, 201-215.
- Doran, J., & Michie, D. (1966). Experiments with the Graph Traverser program. *Proc. R. Soc. (A)*, 294, 235-259.
- Gelernter, H. (1959). Realisation of a geometry-theorem proving machine. *Int. Conf. Inf. Process.*, pp. 273-282. Paris: UNESCO.
- Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *J. Ass. comput. Mach.*, 12, 23-41.
- Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM JI Res. Dev.*, 3, 211-229.
- Wang, H. (1960). Towards mechanical mathematics. *IBM JI Res. Dev.*, 4, 2-22.

4

THE RESOLUTION PRINCIPLE IN THEOREM-PROVING

DAVID LUCKHAM

DEPARTMENT OF MATHEMATICS
UNIVERSITY OF MANCHESTER

INTRODUCTION

The evidence is already in favour of the computer becoming an aid to research in branches of mathematics which do not involve numerical computation. Programs have now been constructed for performing certain symbolic operations in mathematics, for example algebraic simplification of equations and indefinite integration, and for solving some of the problems (which are certainly not in the class of numerical calculations) now occurring in theoretical physics and other areas. One may reasonably expect that sooner or later programs of this type will be incorporated in a 'question-answering' package. As the construction of multi-programming systems progresses, and 'on-line' use of the computer is accepted as normal, it will become more and more practicable for the mathematician to rely on the computer for answers to routine non-numerical questions. It is, therefore, of interest from a practical as well as a theoretical point of view to ask if this development can be taken a stage further by mechanising the processes of mathematical proof. In this way the computer could be used as a tool for establishing true mathematical statements or checking proofs for correctness. The problem is essentially to produce practicable programs for proving theorems which will answer some reasonably complicated questions before overstepping the bounds of available time and memory space. This is potentially one of the most rewarding areas of computer applications, but as many people already know, it is also one of the most frustrating.

Although some research into the possibility of constructing heuristic programs based on the psychological knowledge of human problem-solving, and the role that mathematical intuition plays therein, has been reported, for example, in Newell, Shaw & Simon (1957), at present there can be little

hope of constructing a heuristic program which would be able to choose successful proof lines in say, number theory as accurately as a human mathematician. It would seem that, in the first instance, one must look for simple-minded mechanical procedures for doing mathematics, and must not rely on the activities of the human mathematician for inspiration. A natural choice for a starting point is an abstract formalisation of mathematical proof such as the first-order predicate calculus (in the following discussion we shall assume a familiarity with a formulation of this system such as is given in Hilbert & Ackermann (1950); there are many different but equivalent formulations, and we shall refer to them all under the collective title 'first-order logic'). One might hope that successful theorem-proving programs for this theory would form a basis for obtaining mechanical proofs of theorems belonging to various areas of 'real' mathematics (e.g., group theory or number theory) since in principle one would then be able to prove theorems of the various axiomatic theories formalisable within first-order logic. The results of experiments with theorem-proving programs (see, for example, Wos, Carson & Robinson 1964, 1965) would seem to support this hope, at least in the case of elementary group theory, but it is already clear that the axioms of equality will require additional strategies over and above the basic first-order procedures. And in the case of number theory it may well happen that the first-order procedures are of no use at all. However, the logical first step is to see how far first-order procedures can be improved and extended. If the extensions are significant (so that non-trivial mathematical theorems are obtained) one can expect that heuristic search principles will be used, but these may rely more on features of the basic mechanical procedure than on psychology.

At the outset of our attempt to construct a theorem-proving program we must live with the further complication that first-order logic is undecidable—a result due to Church and Turing (see Davis 1965). Essentially this means that it is *not* possible to construct a computer program which will answer correctly for every statement of first-order logic the question whether or not the statement is a theorem. Under this restriction, the best that can be done is to construct a procedure which will produce a proof of any statement that is in fact a theorem (i.e., is provable), but will continue computing for ever in an attempt to find a proof in the case when it is given a non-theorem (and we ourselves will not be able to tell in general from the state of the computation at any stage whether the procedure will ever stop). We shall refer to such procedures as *proof procedures* or *partial decision procedures*.

One positive result is the existence of sub-classes of sentences of first-order logic which *are* decidable and contain some interesting theorems of mathematics (see, e.g., Ackermann 1954). It will be possible to extend a partial decision procedure so that if it is given a sentence belonging to a decidable sub-class, it will answer 'no' in case the sentence is a non-theorem.

A recent hopeful development is a formulation of first-order logic due to Robinson (1965). Robinson's system, which involves one rule of inference

called the *Resolution Principle*, is intended to provide a basic proof procedure upon which efficient theorem-proving programs can be built, and the results so far, as reported, for example, in Wos, Carson & Robinson (1964, 1965), have been encouraging. Our purpose here is to present and justify Robinson's system, and to indicate how one may hope to derive more powerful systems from it. We shall rely heavily throughout on Robinson (1965).

HERBRAND PROCEDURES

A basic algorithm for proof procedures was given in a fundamental paper by Jacques Herbrand (1930). During the past eight years procedures using Herbrand's theorem have been programmed (see Davis, Logemann & Loveland 1962, Davis & Putnam 1960, Gilmore 1960, McIlroy, Chinlund, Davis & Hinman 1964, Wos, Carson & Robinson 1964) and used to obtain proofs of fairly simple theorems in first-order logic and elementary algebra.†

Briefly, one may describe the workings of these programs (sometimes called refutation procedures since the method used to establish a theorem is to show that its negation is inconsistent) in the following way. Suppose sentence $\mathcal{F}$ is a proposed theorem of first-order logic. We consider $\sim \mathcal{F}$, and suppose this is in prenex form (all quantifiers preceding a quantifier-free matrix) and that the matrix is in conjunctive normal form; suppose, e.g.,

$$\sim \mathcal{F} = (\exists x_1)(\forall x_2)(\exists x_3)(\forall x_4)R(x_1, x_2, x_3, x_4)$$

First, each existentially quantified variable is replaced by a Skolem function of the universally quantified variables governing it (a 0-placed function symbol is interpreted as a constant) and the universal quantifiers are dropped; this gives us

$$\mathcal{S} = R(a, x_2, f(x_2), x_4)$$

(Clearly $\sim \mathcal{F}$ is satisfiable (i.e., has a model) if and only if $\mathcal{S}$ is satisfiable, since $\mathcal{S}$ logically implies $\sim \mathcal{F}$, and if $\sim \mathcal{F}$ is true in some universe U , then there are functions over U satisfying $\mathcal{S}$.) Note that $\mathcal{S}$ is a finite conjunction of *clauses*, each clause being a disjunction of atomic formulae and negations of atomic formulae. Next, consider the set of terms, H , obtained from $\mathcal{S}$ by (i) if a is a constant (i.e., a 0-place function symbol) in $\mathcal{S}$ then a is in H ; (ii) if $a_1, a_2, \dots, a_n$ are terms in H , and f is an n -place function symbol in $\mathcal{S}$, then $f(a_1, a_2, \dots, a_n)$ is in H . (If $\mathcal{S}$ has no constants, an arbitrary constant, a , is included in H .) We shall call H the Herbrand domain of $\mathcal{S}$. The program starts by generating substitution instances of $\mathcal{S}$ obtained by substituting terms from H for the variables in $\mathcal{S}$. For example

x_2	x_4	$R(a, x_2, fx_2, x_4)$
a	a	$R(a, a, fa, a)$
a	fa	$R(a, a, fa, fa)$
fa	a	$R(a, fa, ffa, a)$
a	ffa	$R(a, a, fa, ffa)$
...	...	...

† First-order procedures based on natural deduction systems have also been programmed, notably by H. Wang (1960b), and are discussed in this volume by R. Popplestone.

THEOREM PROVING

where x_2, x_4 are replaced in all possible ways by the terms $a, fa, ffa, \dots$ in H , each variable being replaced at all of its occurrences by the same term. The set of all substitution instances of $\mathcal{S}$ obtainable by replacing each variable in $\mathcal{S}$ by a term from H , is called the Herbrand expansion of $\mathcal{S}$. At certain intervals the program tests the conjunction of the instances so far generated for inconsistency. If this conjunction is consistent, it generates further instances and tests the larger conjunction, and continues in this way until an inconsistent set of instances is found. That this will happen precisely in the case that $\mathcal{S}$ is unsatisfiable (and hence $\mathcal{T}$ is a theorem) follows from Herbrand's theorem (a simple proof is given by Quine 1955).

In most of the earlier proof procedures referred to above, the accent was understandably on improving methods of testing propositional conjunctions for inconsistency. However, the usual method of generating the instances of $\mathcal{S}$ produces all instances involving terms of a given level (or depth of nesting of function symbols) before terms of the next higher level appear. It is a simple matter to show that the number of instances of $\mathcal{S}$ increases exponentially as a function of the maximum level of terms occurring, and as a result, some elementary theorems are beyond the capacity of such procedures (see Robinson 1963). The problem therefore shifts from testing propositional forms to finding more astute ways of generating an inconsistent set of instances of $\mathcal{S}$. One proposal, due to Prawitz and Davis, was to generate instances of the clauses of $\mathcal{S}$ only if those instances were 'linked' in the following sense. A set of clauses is *linked* if whenever a literal L occurs in a clause, the complementary literal $\bar{L}$ occurs in another clause. This is a necessary (but not sufficient) condition for a conjunction of a set of clauses to be inconsistent while the conjunction of any subset is consistent. Certainly a program which generates linked sets of instances stands a better chance of finding an inconsistent set before running out of memory space, and this chance will be greatly improved if good criteria can be found for deciding which of the possible linked sets to try first. A second suggestion is Robinson's Resolution Principle. It would seem that the basic resolution procedure is about as efficient as a procedure which generates linked instances of $\mathcal{S}$. However, the resolution procedure has a very concise formulation which makes the task of finding good preference criteria much easier, and this gives it a definite advantage over methods which involve generating the Herbrand expansion explicitly (see, for example, McIlroy, Chinlund, Davis & Hinman 1964, Wos, Carson & Robinson 1964, 1965).

THE RESOLUTION PRINCIPLE OF J. A. ROBINSON

In the usual formulations of first-order logic, the concept of a mathematical proof is formalised as a finite sequence of well-formed formulae (w.f.f.), each member of the sequence being related to earlier members by one of a specified set of transformations (on w.f.f.) called rules of inference; sometimes a certain set of w.f.f. is designated as 'starting formulae' (axioms). The

theorems are the end-formulae of proof sequences. Herbrand's theorem implies that it is possible to specify the transformations so that, in the case that the starting formula is the negation of a first-order theorem, a contradictory pair of formulae will appear in a proof sequence. Thus equivalent systems of first-order logic may be constructed using 'refutation rules'. A theorem is 'proved' in such a system by refuting its negation. Further, in order to prove a theorem T of a finitely axiomatisable theory with axioms $A_1, A_2, \dots, A_n$ (expressible in first-order logic) the implication

$$(A_1 \& A_2 \& \dots \& A_n) \rightarrow T$$

is established by refuting the conjunctive form,

$$A_1 \& A_2 \& \dots \& A_n \& \sim T$$

Robinson's system is of the refutation type. We give a formulation of this system and show that it is both consistent and complete.

On the basis of the remarks in the previous section (p. 49) about Herbrand's expansion, it is sufficient to restrict the formal language so that the formulae are in Skolem free variable conjunctive form. We adopt the following formalism:

We take as primitive symbols of the language:

- (i) **Variables:** $u, v, w, x, y, z, u_1, v_1, w_1, x_1, y_1, z_1, \dots$
- (ii) **Function symbols for each degree n :**
 $f^n, g^n, h^n, f_1^n, g_1^n, h_1^n, \dots$
- (iii) **Predicate symbols for each degree n :**
 $P^n, Q^n, R^n, P_1^n, Q_1^n, R_1^n, \dots$
- (iv) **Logical connectives** $\sim$ (not), $\&$ (and), $\vee$ (or).

Auxiliary symbols $'$, $($, $)$.

We shall practise the habit of omitting the superscripts denoting degree whenever this does not confuse the issue at hand.

From among the strings composed from the above symbols, certain of them are designated as the well-formed (or meaningful) expressions of the language. These are:

- (i) **Terms.** The set of terms is given by the following inductive definition:
 - (a) a variable is a term,
 - (b) if $a_1, a_2, \dots, a_n$ are terms then $f_i^n(a_1, \dots, a_n)$ is a term.
- (ii) **Atomic formulae.** The expression $P(a_1 \dots a_n)$ where P is a predicate symbol of degree n , and $a_1, \dots, a_n$ are terms, is an atomic formula.
- (iii) **Literals.** A literal is an atomic formula or the negation of an atomic formula.
- (iv) **Clauses.** A clause is a finite disjunction of literals (possibly empty).
- (v) **Conjunctive forms.** A conjunctive form is a finite conjunction of clauses.

At certain points in what follows, it will be convenient to think of a clause as being the set of the literals occurring in it, and a conjunctive form as being a set of clauses.

THEOREM PROVING

We assume that the well-formed expressions and sequences of well-formed expressions have a *lexical ordering*:

- (a) the primitive symbols are ordered according to type (variables, functions, predicates, connectives) and then within each type according to superscript, then subscript, and finally by alphabetical order;
- (b) well-formed expressions are ordered first by length, and then two expressions of equal length are placed in the order of their i th pair of symbols if they differ first at the i th position. Similarly sequences of well-formed expressions are ordered by length and then by the order of first members which differ.

For example, x comes before y in the lexical ordering, ffx before ffy , ffy before fgx , and so on. (We shall often omit the auxiliary symbols when this does not cause confusion.) Indeed given any two well-formed expressions (or sequences of such) we can decide their lexical order.

Our transformation rule depends on the basic operation of substituting terms for variables in well-formed expressions. We therefore introduce the following definitions:

If the variable v is replaced at each of its occurrences in the well-formed expression C by the term t , the resulting well-formed expression is called a *substitution instance* of C , and is denoted by $C\{(t, v)\}$. Similarly, the result of *simultaneously* replacing all occurrences of different variables $v_1, v_2, \dots, v_n$ in C by terms $t_1, t_2, \dots, t_n$ respectively, is denoted by $C\{(t_1, v_1), \dots, (t_n, v_n)\}$.

Definition. A *substitution* σ is a set of ordered pairs, $\sigma = \{(t_1, v_1), \dots, (t_n, v_n)\}$, the first element of each pair being a term and the second element a variable, such that $v_i \neq v_j$ and $t_i \neq v_i$. The result of applying σ to C , $C\sigma$, is the substitution instance $C\{(t_1, v_1), \dots, (t_n, v_n)\}$.

Examples

Let	$C = P(x, z, y) \vee P(e, f(u, y), f(a, b)),$
	$\sigma = \{(e, x), (f(u, y), z), (f(a, b), y)\},$
and	$\tau = \{(f(a, b), y)\};$
then	$C\sigma = P(e, f(u, y), f(a, b)) \vee P(e, f(u, f(a, b)), f(a, b)),$
and	$(C\sigma)\tau = P(e, f(u, f(a, b)), f(a, b))$

The condition, $t_i \neq v_i$ ensures that two substitutions are set-theoretically equal if and only if they have the same effect on all well-formed expressions.

The notation $P(\mathcal{S})$ denotes the set of instances obtained by applying to a set $\mathcal{S}$ of well-formed expressions all substitutions with terms belonging to a set P of terms.

First we introduce a special case of the transformation rule which applies only to expressions containing no variables; the motivation for the general rule will then be clear. Following Robinson, we call expressions containing no variables *ground* expressions. Ground terms are either 0-degree function symbols (constants) or else terms of the form $f^{(n)}(a_1, \dots, a_n)$ where $a_1, \dots, a_n$ are ground terms; a literal or clause is a ground expression if all of its terms are ground terms. If $\mathcal{S}$ is a conjunctive form, the Herbrand expansion of

$\mathcal{S}$ is the set $H(\mathcal{S})$ of ground instances of the clauses of $\mathcal{S}$, and this will be infinite if $\mathcal{S}$ is not a ground expression and contains at least one n -placed function symbol, $n > 0$. Herbrand's theorem may be stated in the following form:

Herbrand's Theorem. *If $\mathcal{S}$ is a finite conjunction of clauses, and H is its Herbrand domain, then $\mathcal{S}$ is unsatisfiable if and only if some finite subset of $H(\mathcal{S})$ is inconsistent.*

Thus in order to test the unsatisfiability of a conjunctive form $\mathcal{S}$, it is (in theory) only necessary to test the inconsistency of a finite conjunction of the ground clauses in $H(\mathcal{S})$. One method of testing the inconsistency of a conjunction of propositional clauses (or ground clauses) is the well-known method of complementary literal elimination which Robinson calls *Ground Resolution*.

Definition. A *ground resolvent* of two ground clauses C, D is a third ground clause E obtained as follows: if there is a complementary pair of literals, L in C and $\sim L$ in D , then E is the disjunction of the set of literals,

$$(C - L) \cup (D - \sim L).$$

Notice that any pair of clauses has only finitely many resolvents, and that a resolvent of two clauses contains only literals already in the starting clauses. Hence, if one starts with a finite set of ground clauses and iterates the procedure of adding to the set of clauses so far obtained all resolvents of those clauses, the set obtained after some finite number of steps will be closed under the operation of taking resolvents.

Notation. If $\mathcal{S}$ is a set of ground clauses, $R(\mathcal{S})$ denotes the union of $\mathcal{S}$ and all ground resolvents of pairs of members of $\mathcal{S}$;

$$R^{n+1}(\mathcal{S}) = R(R^n(\mathcal{S})).$$

Now the conjunction of a set of ground clauses is consistent if it is possible to choose a ground literal from each of the clauses without being forced to choose a complementary pair of literals (i.e., a pair $L, \sim L$). Otherwise the conjunction is contradictory (truth-functionally inconsistent). We shall call a set of ground literals which does *not* contain a complementary pair a *model*. Thus we may say that a conjunction of ground clauses is consistent if and only if there is a model containing a member of each clause. Under this definition the empty clause, ϕ , is inconsistent, and so is any conjunction containing ϕ .

Theorem 1 (Robinson). *If $\mathcal{S}$ is a finite conjunction of ground clauses, then $\mathcal{S}$ is inconsistent if and only if $\phi \in R^n(\mathcal{S})$ for some $n \geq 0$.*

Proof. If E is a resolvent of clauses C and D , then any model containing a literal from each of C and D also contains a literal from E (since E is obtained by eliminating exactly *one* complementary pair of literals). Hence a model for $\mathcal{S}$ is also a model for $R^n(\mathcal{S})$, $n = 1, 2, \dots$. Therefore, if for some n , $\phi \in R^n(\mathcal{S})$, there is no model for $\mathcal{S}$.

THEOREM PROVING

On the other hand, if ϕ does not belong to $R^n(\mathcal{S})$ for any n , it is possible to construct a model for $\mathcal{S}$ as follows.

Let $L_1, L_2, \dots, L_k$ be the atomic formulae which occur in $\mathcal{S}$ or whose negations occur in $\mathcal{S}$. Let m be such that $R^{m+1}(\mathcal{S}) = R^m(\mathcal{S})$, and let C be any clause in $R^m(\mathcal{S})$. Then define a sequence $M_1, M_2, \dots, M_k$ of literals such that each M_i is either L_i or $\sim L_i$ according to the rule,

$$M_{i+1} = \begin{cases} L_{i+1} & \text{if no clause } C \text{ has all of its literals in the set} \\ & \{\sim M_1, \sim M_2, \dots, \sim M_i, \sim L_{i+1}\} \\ \sim L_{i+1} & \text{otherwise} \end{cases}$$

Let M be the model $\{M_1, M_2, \dots, M_k\}$.

Now assume that no clause is contained in $\{\sim M_1, \dots, \sim M_i\}$ but that $C \subseteq \{\sim M_1, \dots, \sim M_{i+1}\}$. Then $\sim M_{i+1} \in C$. By the construction $M_{i+1} = \sim L_{i+1}$ (hence $L_{i+1} \in C$) and this implies that there is a clause $D \subseteq \{\sim M_1, \dots, \sim M_i, \sim L_{i+1}\}$. Under the assumption above, $\sim L_{i+1} \in D$; but then the resolvent $(C - L_{i+1}) \cup (D - \sim L_{i+1})$ belongs to $R^m(\mathcal{S})$ and is contained in $\{\sim M_1, \dots, \sim M_i\}$, contradiction.

Hence if no clause is contained in $\{\sim M_1, \dots, \sim M_i\}$, the same is true of $\{\sim M_1, \dots, \sim M_{i+1}\}$. But a similar argument shows that no clause is contained in $\{\sim M_1\}$ since $\phi \in R^m(\mathcal{S})$. Therefore by induction M contains a literal from each clause in $R^m(\mathcal{S})$. Q.E.D.

The set $\mathcal{S}$ of ground clauses is therefore inconsistent if and only if the empty clause ϕ belongs to the set of ground clauses, $R^n(\mathcal{S})$, where $R^{n+1}(\mathcal{S}) = R^n(\mathcal{S})$; to test for inconsistency, one starts with $\mathcal{S}$ and generates $R(\mathcal{S})$, $R^2(\mathcal{S})$, $\dots$ until either the empty clause or closure is obtained.

As was mentioned in the previous section (p. 49), the problem with a procedure of the above type, depending on testing subsets of $H(\mathcal{S})$, is not the time taken to test a subset (in the case when $\mathcal{S}$ is itself relatively simple) but that if $H(\mathcal{S})$ is generated according to the increasing complexity of the ground terms in the instances, a prohibitively large number of instances may have to be generated before an inconsistent conjunction arises. If $\mathcal{S}$ is simple, then a small inconsistent sub-conjunction can be found. The problem therefore is to avoid generating the consistent instances. One way to do this is to *predict* which instances of two clauses will contain complementary literals (below we call this a 'clash') using the information we have at hand about the way in which the terms of the Herbrand domain are constructed. We have in mind an argument of the following sort. Suppose the clauses of $\mathcal{S}$ are

- (1) $P(x, e, x)$
- (2) $\sim P(y, z, v) \vee \sim P(y, v, w) \vee P(e, z, w)$
- (3) $P(a, f(u, v), e)$
- (4) $\sim P(e, f(f(b, c), a), a)$

where a, b, c, e are constants.

Consider any instance of (1); this will 'clash' with any instance of (2) in

which v is replaced by e , and y and w are replaced by the term which replaces x in (1), for such an instance will have the form

$$\sim P(x, z, e) \vee \sim P(x, e, x) \vee P(e, z, x).$$

Thus there will be a family of resolvents of instances of (1) and (2) of the form

$$(5) \sim P(x, z, e) \vee P(e, z, x).$$

Some of these clauses will 'clash' with instances of (3); namely those instances of (5) of the form

$$\sim P(a, f(u, v), e) \vee P(e, f(u, v), a),$$

the resolvents being

(6) $P(e, f(u, v), a)$. This family contains the clause

(7) $P(e, f(f(b, c), a), a)$ which clearly contradicts (4). Thus large enough segments of $H(\mathcal{S})$ are contradictory; if $H(\mathcal{S})$ is generated by the usual method, the first contradictory segment would contain between 4^8 and 20^8 clauses.

Such arguments involve applying a succession of substitutions of terms (not necessarily ground terms) for variables, each substitution being chosen so that the result of applying it to two clauses is that some of the literals in the first clause become equal (or collapse) to a literal L and some of the literals in the second clause collapse to $\sim L$. In order to formalise these arguments, we need to introduce three simple concepts: (i) the operation of composition of substitutions, (ii) the concept of a 'unifying' substitution, and (iii) the concept of a simplest substitution having a given property.

Definition. Let θ, ψ be substitutions.

The *composition* of θ and ψ (denoted by $\theta\psi$) is the substitution defined as follows:

$$\begin{aligned} \text{Let } \theta &= \{(t_1, u_1), \dots, (t_n, u_n)\}, \\ \theta' &= \{(t_1\psi, u_1), \dots, (t_n\psi, u_n)\}, \text{ such that } t_i\psi \neq u_i, \end{aligned}$$

and let ψ' consist of those pairs of ψ whose second elements do not occur among the variables $u_1, \dots, u_n$ of θ , say

$$\psi' = \{(s_1, v_1), \dots, (s_k, v_k)\}.$$

Then

$$\theta\psi = \theta\theta' \cup \psi'.$$

Remarks. It is clear from this definition that the result of applying θ and ψ successively to a well-formed expression C is the same as applying $\theta\psi$, i.e.,

$$(C\theta)\psi = C\theta\psi.$$

Further, composition of substitutions is associative,

$$(\theta\sigma)\psi = \theta(\sigma\psi).$$

Definition. Let $\mathcal{L} = \{L_1, L_2, \dots, L_n\}$ be a set of literals. $\mathcal{L}$ is *unified* (Robinson's terminology) by σ if

$$L_1\sigma = L_2\sigma = \dots = L_n\sigma$$

THEOREM PROVING

Remark. If σ unifies L , so does $\sigma\tau$ where τ is any substitution.

Definition. θ is a *simplest substitution* satisfying a condition C if for any σ satisfying C there is a λ such that

$$\sigma = \theta\lambda$$

Remark. Any pair of simplest substitutions (satisfying a given condition) differ by a change of variables. For, if $\theta = \sigma\lambda$ and $\sigma = \theta\psi$, it is easy to see that θ and σ must be of the form

$$\begin{aligned}\theta &= \{(s_1, u_1), \dots, (s_n, u_n)\} \text{ and} \\ \sigma &= \{(s_1\psi, u_1), \dots, (s_n\psi, u_n), (t_1, v_1) \dots (t_m, v_m)\}\end{aligned}$$

where $s_i = s_i\psi\lambda$ or $s_i\psi = u_i$, and $t_i\lambda = v_i$, which can only happen if ψ substitutes variables for variables.

In many cases there will be no simplest substitution satisfying the given condition. However, we shall show that if a set $\mathcal{L}$ of literals is unifiable, then there is a simplest substitution unifying $\mathcal{L}$ and, moreover, that there is an algorithm for finding such a substitution. In the definition below we need to pick one of the simplest substitutions unifying a set of literals; it does not matter which one, but that only one is chosen. We could for example choose the earliest one in the lexical ordering of sequences of well-formed expressions by considering the pairs in a substitution to be arranged as a sequence in the order of the second elements. We shall therefore assume that we have a method for choosing a unique simplest substitution unifying a given unifiable set of literals. It is also necessary to fix upon a method of changing the variables in a pair of clauses so that no variable occurs in both; it is not essential that this be the method given below.

We now extend the concept of a resolvent of two clauses to the case where the clauses are not necessarily ground clauses.

Definition. A resolvent of two clauses C_1 and C_2 is a third clause D obtained as follows:

(i) If $v_1, \dots, v_m$ are the variables of C_2 , and the highest variable of C_1 in the lexical order is u_k , let $\theta = \{(u_{k+1}, v_1), \dots, (u_{k+m}, v_m)\}$. (None of the variables in $C_2\theta$ occurs in C_1 .)

(ii) If there is a pair of sets of literals, $\mathcal{L} = \{L_1, \dots, L_k\}$ and $\mathcal{M} = \{M_1, \dots, M_n\}$ such that $\mathcal{L} \subseteq C_1$, and $\mathcal{M} \subseteq C_2$ and the set $\{L_1, \dots, L_k, \sim M_1\theta, \dots, \sim M_n\theta\}$ is unifiable, let σ_0 be the chosen simplest unifying substitution so that $\mathcal{L}\sigma_0$ and $\mathcal{M}\theta\sigma_0$ are complementary literals; then D is the disjunction of the literals,

$$(C_1 - \mathcal{L})\sigma_0 \cup (C_2 - \mathcal{M})\theta\sigma_0$$

Remarks. (i) Any pair of clauses have (at most) a finite number of resolvents because there are only finitely many pairs of sets of literals, $\mathcal{L}, \mathcal{M}$, and for each pair at most one substitution, σ_0 . (ii) The resolvents of a pair of ground clauses are the ground resolvents.

The transformation rule or rule of inference for our system of logic, which Robinson calls the Resolution Principle, is the following:

Resolution Principle. From any two clauses C_1 and C_2 , infer a resolvent of C_1 and C_2 .

Finally, a 'proof' or, more accurately, a refutation of a conjunctive form $\mathcal{S}$ is a sequence $C_1, C_2, \dots, C_n$ of clauses such that each clause either belongs to $\mathcal{S}$ or is inferred from two earlier clauses by a resolution, and $C_n = \phi$.

Our system of logic is now completely specified. We prove that it is consistent and complete, i.e., $\mathcal{S}$ is unsatisfiable if and only if there is a refutation of $\mathcal{S}$.

As before, let $R(\mathcal{S})$ denote the set of all clauses in $\mathcal{S}$ and all resolvents of pairs of clauses in $\mathcal{S}$; $R^{n+1}(\mathcal{S}) = R(R^n(\mathcal{S}))$. There is a refutation of $\mathcal{S}$ if and only if $\phi \in R^n(\mathcal{S})$ for some n .

Remark. If σ_0 is a simplest substitution unifying a set $\mathcal{L}$ of literals, then all of the function symbols of the terms of σ_0 occur in the terms of $\mathcal{L}$. This will be proved later.

Theorem 2. If $\phi \in R^n(\mathcal{S})$ for some n , then $\mathcal{S}$ is unsatisfiable.

Proof. Let $\mathcal{R}$ denote resolution in the extended sense that in the definition of resolvent above, $\mathcal{L}$ or $\mathcal{M}$ may be empty. Thus $R(\mathcal{S}) \subseteq \mathcal{R}(\mathcal{S})$. This in turn imposes a corresponding trivial extension on ground resolution in which additional resolvents of the form $A \cup B$ or $A \cup (B - L)$ are introduced. These are already redundant and it is easy to see in the case when $\mathcal{S}$ is a set of ground clauses, that if $\phi \in \mathcal{R}^n(\mathcal{S})$ then $\phi \in R^n(\mathcal{S})$. We make this extension only to facilitate the following proof.[†]

Let H be the Herbrand domain of $\mathcal{S}$.

First we prove $H(\mathcal{R}(\mathcal{S})) \subseteq \mathcal{R}(H(\mathcal{S}))$. (1)

If $C \in \mathcal{S}$ then clearly $H(C) \subseteq \mathcal{R}(H(\mathcal{S}))$.

Suppose C is a resolvent (in the extended sense).

Let $C = (A - \mathcal{L})\sigma_0 \cup (B - \mathcal{M})\theta\sigma_0$ where $A, B \in \mathcal{S}$ and either $\mathcal{L}\sigma_0, \mathcal{M}\theta\sigma_0$ are complementary literals or at least one of $\mathcal{L}, \mathcal{M}$ is empty.

For any substitution τ ,

$$\begin{aligned} C\tau &= (A - \mathcal{L})\sigma_0\tau \cup (B - \mathcal{M})\theta\sigma_0\tau \\ &= (A\sigma_0\tau - \mathcal{L}'\sigma_0\tau) \cup (B\theta\sigma_0\tau - \mathcal{M}'\theta\sigma_0\tau) \end{aligned}$$

where $\mathcal{L}' = \mathcal{L}$ or $\mathcal{L}' = \phi$ and $\mathcal{M}' = \mathcal{M}$ or $\mathcal{M}' = \phi$

depending on whether $\mathcal{L}$ contains all literals L in A , such that $L\sigma_0\tau = \mathcal{L}\sigma_0\tau$ and $\mathcal{M}$ contains all literals M in B such that $M\theta\sigma_0\tau = \mathcal{M}\theta\sigma_0\tau$.

Let τ be a substitution from H such that $C\tau \in H(\mathcal{R}(\mathcal{S}))$. Then there is a τ' such that $\tau \subseteq \tau'$, $C\tau = C\tau'$, and $A\sigma_0\tau', B\theta\sigma_0\tau' \in H(\mathcal{S})$. (Essentially, τ' substitutes constants from H for any variables remaining in the terms of $A\sigma_0\tau$ and $B\theta\sigma_0\tau$; by the remark above, $\sigma_0\tau'$ and $\theta\sigma_0\tau'$ are also substitutions from H .)

Either $\mathcal{L}'\sigma_0\tau' = \sim\mathcal{M}'\theta\sigma_0\tau'$ or one of $\mathcal{L}', \mathcal{M}'$ is empty.

Therefore $C\tau \in \mathcal{R}(H(\mathcal{S}))$.

[†] Resolution does in fact generate such redundant clauses, e.g., if $\mathcal{S}$ is the two-clause set, $\{(P(x, x), P(x, a)), (\sim P(x, x), \sim P(x, a))\}$, then $H = \{a\}$ and $R(H(\mathcal{S})) = \{(P(a, a)), (\sim P(a, a)), \phi\}$ whereas $H(R(\mathcal{S})) = \{(P(a, a)), (\sim P(a, a)), (P(a, a), \sim P(a, a)), \phi\}$.

THEOREM PROVING

By the above remark, the function symbols in $\mathcal{R}^n(\mathcal{S})$ already occur in $\mathcal{S}$ and hence H is also the Herbrand domain of $\mathcal{R}^n(\mathcal{S})$.

If we assume $H(\mathcal{R}^n(\mathcal{S})) \subseteq \mathcal{R}^n(H(\mathcal{S}))$,
then $H(\mathcal{R}^{n+1}(\mathcal{S})) \subseteq \mathcal{R}(H(\mathcal{R}^n(\mathcal{S})))$ by (1),

$$\subseteq \mathcal{R}(\mathcal{R}^n(H(\mathcal{S}))).$$

Therefore $H(\mathcal{R}^n(\mathcal{S})) \subseteq \mathcal{R}^n(H(\mathcal{S}))$ for all n , by induction.

If $\phi \in \mathcal{R}^n(\mathcal{S})$ then $\phi \in H(\mathcal{R}^n(\mathcal{S}))$; by the above result $\phi \in \mathcal{R}^n(H(\mathcal{S}))$ and hence $\phi \in \mathcal{R}^n(H(\mathcal{S}))$. This implies that there is a finite subset $Q \subseteq H(\mathcal{S})$ containing at most 2^n clauses such that $\phi \in \mathcal{R}^n(Q)$. Q is inconsistent (Theorem 1), and therefore $\mathcal{S}$ is unsatisfiable (Herbrand's Theorem). Q.E.D.

Theorem 3 (Robinson). *If $\mathcal{S}$ is unsatisfiable then $\phi \in \mathcal{R}^n(\mathcal{S})$ for some n .*

Proof. We prove first that if $P \subseteq H$ then $R(P(\mathcal{S})) \subseteq P(R(\mathcal{S}))$. (1)

Suppose $C \in R(P(\mathcal{S}))$.

If $C \in P(\mathcal{S})$ then clearly $C \in P(R(\mathcal{S}))$.

If C is a resolvent, then

$$C = (A\alpha - L\alpha) \cup (B\beta - M\beta)$$

where $A, B \in \mathcal{S}$, $A\alpha, B\beta \in P(\mathcal{S})$, and $L\alpha, M\beta$ are complementary literals.

Let

$$\alpha = \{(s_1, u_1), \dots, (s_k, u_k)\} \text{ and}$$

$$\beta = \{(t_1, v_1), \dots, (t_m, v_m)\}$$

where $u_1, \dots, u_k$ are all the variables of A and $v_1, \dots, v_m$ are all the variables of B . Let θ be the change of variables, $\theta = \{(u_{k+1}, v_1), \dots, (u_{k+m}, v_m)\}$ and let $\sigma = \{(s_1, u_1), \dots, (s_k, u_k), (t_1, u_{k+1}), \dots, (t_m, u_{k+m})\}$. Then $A\sigma = A\alpha$, $B\theta\sigma = B\beta$, and $C = (A - \mathcal{L})\sigma \cup (B - \mathcal{M})\theta\sigma$ where $\mathcal{L}\sigma = L\alpha$ and $\mathcal{M}\theta\sigma = M\alpha$. Furthermore, since σ unifies the set $\mathcal{L} \cup \sim \mathcal{M}\theta$ there is a chosen simplest unifying substitution σ_0 and a resolvent C' of A and B , where $C' = (A - \mathcal{L})\sigma_0 \cup (B - \mathcal{M})\theta\sigma_0$. Let $\sigma = \sigma_0\lambda$. Then $C = C'\lambda$. Thus $C \in P(R(\mathcal{S}))$.

The general form of (1) now follows by induction:

Assume $R^n(P(\mathcal{S})) \subseteq P(R^n(\mathcal{S}))$

Then

$$\begin{aligned} R^{n+1}(P(\mathcal{S})) &= R(R^n(P(\mathcal{S}))) \\ &\subseteq R(P(R^n(\mathcal{S}))) \\ &\subseteq P(R^{n+1}(\mathcal{S})) \text{ by (1)} \end{aligned}$$

If $\mathcal{S}$ is unsatisfiable, a finite subset of $H(\mathcal{S})$ is inconsistent. Suppose this is $P(\mathcal{S})$. Then $\phi \in R^n(P(\mathcal{S}))$ for some n (Theorem 1). Therefore $\phi \in P(R^n(\mathcal{S}))$.

This implies $\phi \in \mathcal{R}^n(\mathcal{S})$ since substituting terms for variables cannot reduce a non-empty clause to the empty clause. Q.E.D.

Finally we must show that there is an algorithm for choosing a simplest substitution unifying a set of literals. An algorithm is given by Robinson (1965), and this seems to be as straightforward and efficient as one can hope for. This procedure starts with the empty substitution and builds up step by step a simplest σ_0 which unifies the set $\mathcal{L}$ of literals. If at the k th step

the substitution so far obtained is σ_k , and the literals $L_1\sigma_k, \dots, L_n\sigma_k$ in $\mathcal{L}\sigma_k$ are not all equal, the procedure changes σ_k on the basis of a 'disagreement list' containing the first well-formed expression in each $L_i\sigma_k$ which needs to be changed. Consider each $L_i\sigma_k$ to be a sequence of symbols. Define the *disagreement set* of $\mathcal{L}\sigma_k$ to be the set of all well-formed sub-expressions of the literals in $\mathcal{L}\sigma_k$ which begin at the first symbol position at which not all the literals have the same symbol. This set is either $\mathcal{L}$ itself (in which case $\mathcal{L}$ is not unifiable) or else it contains a term or sub-term from each literal in $\mathcal{L}\sigma_k$ (e.g., the disagreement set of $\{P(x, f(x, g(y))), v\}, P(x, f(x, z), w)\}$ is $\{z, g(y)\}$).

Robinson's unification procedure is as follows:

- (1) Set $\sigma_1 = \phi$ and go to (2).
- (2) If the elements of $\mathcal{L}\sigma_k$ are all equal, set $\sigma_0 = \sigma_k$ and stop; otherwise go to (3).
- (3) Let s_k, t_k be the two earliest expressions in the lexical ordering of the disagreement set of $\mathcal{L}\sigma_k$; if s_k is a variable and does not occur in t_k , set $\sigma_{k+1} = \sigma_k\{(t_k, s_k)\}$ and go to (2); otherwise stop.

Since $\mathcal{L}\sigma_{k+1}$ contains one less variable than $\mathcal{L}\sigma_k$, the procedure must stop in at most m steps if $\mathcal{L}$ has m variables. Also it is clear that if the procedure stops at stage (2), σ_0 is uniquely determined and its terms contain only function symbols occurring in $\mathcal{L}$. It remains for us to show that if $\mathcal{L}$ is unifiable then σ_0 is defined and is a simplest unifying substitution.

Theorem 4 (Robinson). *If λ is any substitution unifying set $\mathcal{L}$ of literals, then the unification procedure applied to $\mathcal{L}$ stops at stage (2) and $\lambda = \sigma_0\tau$ for some τ .*

Proof. By induction on the number of steps taken by the procedure before it stops. Assume that at the k th step the procedure is at stage (2) and there is a τ_k such that $\lambda = \sigma_k\tau_k$. (This is clearly true at the first step since $\lambda = \phi\lambda$.) There are two cases. If σ_k unifies $\mathcal{L}$ then $\sigma_0 = \sigma_k$ and $\lambda = \sigma_0\tau_k$. Otherwise the disagreement set of $\mathcal{L}\sigma_k$ contains at least two expressions and these have different first symbols. Now $\sigma_k\tau_k$ unifies $\mathcal{L}$. Therefore τ_k unifies this disagreement set. But two expressions with different first symbols cannot be unified unless one of the expressions is a variable which does not occur in the other. Since variables are earliest in the lexical ordering, s_k and t_k satisfy the conditions of stage (3) so that the procedure does not stop there, and $\sigma_{k+1} = \sigma_k\{(t_k, s_k)\}$.

Now let τ_{k+1} be obtained from τ_k by deleting any pair containing s_k as a second element. $\tau_{k+1} = \tau_k - \{(s_k\tau_k, s_k)\}$.

Then

$$\begin{aligned}\tau_k &= \{(s_k\tau_k, s_k)\} \cup \tau_{k+1} \\ &= \{(t_k\tau_k, s_k)\} \cup \tau_{k+1}\end{aligned}$$

since τ_k unifies the disagreement set,

$$= \{(t_k\tau_{k+1}, s_k)\} \cup \tau_{k+1}$$

THEOREM PROVING

since s_k does not occur in t_k ,

$$= \{(t_k, s_k)\}_{\tau_{k+1}}$$

Therefore $\lambda = \sigma_{k+1}\tau_{k+1}$ and the procedure returns to stage (2). Q.E.D.

It is a simple matter to construct examples showing that the choice of a simplest substitution in the definition of resolution is crucial for the completeness of the system, though not for consistency. However it is likely that the definition may be restricted to 'pairwise' resolution in which the sets $\mathcal{L}$ and $\mathcal{M}$ each contain a single literal; the completeness proof should go through as before with minor complications.

As it stands, the basic resolution procedure of repeatedly computing resolvents, resolvents of resolvents, and so on, until the empty clause occurs, is still very inefficient. Some strategies for choosing more likely inferences first, and for cutting down redundant inferences, are easy to formulate. For example, one would naturally give preference to resolvents of 'short' clauses; the only guarantee that a resolvent is shorter than its parent clauses is that one of the parents is a unit clause. Again, some simple strategies will eliminate inferences from some of the consistent subsets of the starting set of clauses. Such strategies have been successfully used on simple algebraic theorems (see Wos, Carson & Robinson 1964, 1965), but it must be mentioned that in these cases the starting set contains two or three unit clauses. In more complex cases, more involved strategies and probably heuristic principles (which will destroy the completeness of the system) will have to be found.

One line of attack is to use the fact that if there is a deduction of ϕ from $\mathcal{S}$, then there is a deduction which denies a possible model M for $H(\mathcal{S})$ by using a clause satisfied by M as a parent for each sub-deduction. Using this, Robinson (to be published) has shown that in certain cases the procedure need only be applied to a subset of $R^n(\mathcal{S})$. It is not yet known how far such improvements to the basic resolution procedure can be taken and we can only state that much theoretical and experimental work needs to be done.

REFERENCES

- Ackermann, W. (1954). *Solvable cases of the decision problem*. Amsterdam: North-Holland.
- Chinlund, T. J., Davis, M., Hinman, P. G., & McIlroy, D. (1964). *Theorem-proving by matching*. Bell Laboratories. Spring 1964.
- Davis, M., ed. (1965). *The undecidable*. New York: Raven.
- Davis, M., Logemann, G. & Loveland, D. (1962). A machine program for theorem-proving. *Communs Ass. comput. Mach.*, 5, 394-397.

- Davis, M., & Putnam, H. (1960). A computing procedure for quantification theory. *J. Ass. comput. Mach.*, 7, 201-215.
- Gilmore, P. C. (1960). A proof method for quantification theory. *IBM JI Res. Dev.*, 4, 28-35.
- Herbrand, J. (1930). Recherches sur la théorie de la démonstration. *Pr. Tow. nauk. warsz. (b)*, 33.
- Hilbert, D., & Ackermann, W. (1950). *Principles of mathematical logic*. New York: Chelsea.
- Newell, A., Shaw, J. C., & Simon, H. A. (1957). Empirical explorations of the logic theory machine. *Proc. west. jt Computer Conf.*, pp. 218-239.
- Popplestone, R. J. (1966). Pp. 31-46 of this volume.
- Prawitz, D. (1960). An improved proof procedure. *Theoria.*, 26, 102-139.
- Quine, W. V. O. (1955). A proof procedure for quantification theory. *J. of Sym. Logic*, 20, 141-149.
- Robinson, J. A. (1963). Theorem-proving on the computer. *J. Ass. comput. Mach.*, 10, 163-174.
- Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *J. Ass. comput. Mach.*, 12, 23-41.
- Robinson, J. A. Automatic deduction with hyper-resolution. (To be published.)
- Wang, H. (1960a). Towards mechanical mathematics. *IBM JI Res. Dev.*, 4, 2-22.
- Wang, H. (1960b). Proving theorems by pattern recognition—I. *Communs Ass. comput. Mach.*, 3, 220-234.
- Wang, H. (1960c). Proving theorems by pattern recognition—II. *Bell Syst. tech. J.*, 40, 1-41.
- Wos, L., Carson, D. F., & Robinson, J. A. (1964). The unit preference strategy in theorem proving. *AFIPS Conference proceedings—1964 Fall Joint Computer Conference*, 26, Pt 1, 615-621. Washington: Spartan.
- Wos, L., Carson, D. F. & Robinson, G. A. (1965)., Efficiency and completeness of the set of support strategy in theorem proving. *J. Ass. comput. Mach.*, 12, 536-541.



MACHINE LEARNING AND HEURISTIC PROGRAMMING

5

TREE-SEARCHING METHODS WITH AN APPLICATION TO A NETWORK DESIGN PROBLEM

R. M. BURSTALL

EXPERIMENTAL PROGRAMMING UNIT
UNIVERSITY OF EDINBURGH

SEARCH TREES

We will talk about problems with the following characteristics:

- (i) We could recognise a solution to the problem if given one.
- (ii) The solution is known to lie in some finite set of objects which is nevertheless too large to enumerate in practice.
- (iii) We have no straightforward means of calculating the solution or solutions, without searching through at least part of the set of objects.

Let us call the set of objects which is known to contain the solutions the 'candidates'. We include cases where there is more than one solution or where an optimal solution is required.

Some examples are:

- (i) In playing chess there are only a finite number of possible strategies (candidates) but the number is far too large to enumerate.
- (ii) In industrial scheduling we might have to assign some jobs to some machines to satisfy certain conditions. Any assignment is a candidate and there are a finite, but usually large, number of assignments.
- (iii) Given a set of points and a table of distances between them we might want to find the shortest route passing through each point just once (the 'Travelling Salesman' problem). The candidates are the $(n-1)!$ permutations of the points omitting the starting point.

In this section we will describe in an abstract way two approaches to problems of this type. We will give examples of their use, first in 'Some problems about sets' (p. 69 some examples of special search algorithms for problems about choosing subsets of a set, and then in 'An electricity

network design problem' (p. 75) applications of the approaches in the solution of a network design problem. Although both the approaches have often been used before, the discussion may help to clarify those features common to the different applications.

Our two approaches are both search techniques (partial enumeration techniques). The first we will call 'the method of neighbours' and the second 'the method of decision trees'. We claim no special merit for these names, but they will serve for the present.

In our first method we define a relation called neighbourhood between pairs of candidates. This should have the property that from any candidate, or at least from some known starting set of candidates, it should be possible to get to any candidate which might be a solution by means of a chain of candidates each of which is a neighbour of the last. For example, if the candidates were permutations the neighbours of a permutation might be all permutations obtainable from it by interchanging a pair of elements. Beginning with the candidate or candidates of the starting set we may take all neighbours of each, then all their neighbours, etc.

In order to save computing time we may not wish to explore all neighbours, or at least if we do explore them all we wish to do it in an order which is likely to bring the desired solutions more quickly. To eliminate some neighbours or to choose the order we may use heuristic methods, i.e., methods which are supposed to be helpful but not guaranteed to be so, or we may use some properties of the candidates, if we know any useful ones, to rule out certain candidates or sets of candidates. A common device is to assign a value to each candidate by means of an evaluation function and explore only neighbours with better values, or at least explore these first. In the extreme case we take only the best valued neighbour and if this has a value worse than the last candidate we simply give up, or start at random somewhere else (Reiter & Sherman 1963). We may call this the linear case. If on the other hand we sometimes take more than one neighbour we will be searching a tree of candidates (compare the contribution to these proceedings by J. E. Doran). Another slightly different method is to define more than one kind of neighbourhood relation and then decide which kind of neighbour is likely to be more promising. For example, in the case given above, permuting two elements which lie far apart might be more favourable for some reason. If there are a lot of neighbours this may save us generating them all and evaluating the merit of each.

Our second method is to construct a decision tree. It is similar to, but rather more general than, the method of 'backtrack programming' which has just been reviewed by Golomb & Baumert (1965). In this case we start off with an open mind, considering any candidate as a possible solution and then make the hypothesis that the solution satisfies some condition. We must of course remember to consider also the alternative that it fails to satisfy the hypothetical condition (or if there are many solutions we must consider in turn both those satisfying and those not satisfying it). For each

case we then impose a further condition, considering also the opposite case and so on. If we go on long enough, the accumulated conditions will have specified a unique candidate, in which case we look to see whether it is a solution; or possibly we can see from them that no candidate satisfying the conditions could possibly be a solution, in which case we can terminate this branch of the search. In general we will branch not just into two (a condition holding or failing) but into any number of mutually exclusive conditions.

Instead of talking about conditions it may be easier to talk about the corresponding sets of candidates, saying that each node of the tree corresponds to a set of candidates. The initial node, where we impose no conditions, corresponds to the set of all candidates. If a node corresponds to a set S of candidates then either S is a unit set or the node has successors which correspond to sets $S_1, S_2, \dots, S_n$ forming a partition of S , i.e.,

$$\begin{aligned} S_1 \cup S_2 \cup \dots \cup S_n &= S && \text{(they cover } S) \\ S_i \cap S_j &= \text{nil} && \text{for all } i, j \text{ (they are disjoint)} \end{aligned}$$

We demand that they cover S so as not to ignore any possible solutions. We demand that they are disjoint so that we can be sure not to waste our time considering the same set (i.e., $S_i \cap S_j$ if this is non-empty), more than once. Disjointness is sometimes very difficult to arrange without laborious checking or carrying a great deal of information and it may be the major problem in defining a suitable decision tree. The dilemma is that if we characterise the sets S_i too loosely they will not be disjoint but if we do it too restrictively they will not cover S . This is reminiscent of the difficulties met in choosing an axiom set for some mathematical system; if the axioms are too loose we may be able to prove contradictions, if they are too restrictive we may not be able to prove statements which we would like to have as theorems.

Let us take a simple example to make this more concrete. Suppose that four players a, b, c and d have taken part in a championship with the following results

$$\begin{array}{lll} d \text{ beat } a & a \text{ beat } b & c \text{ beat } b \\ c \text{ beat } d & b \text{ beat } d & a \text{ beat } c. \end{array}$$

We wish to rank the four players in the sequence with the smallest number of discrepancies, defining a discrepancy as: x precedes y but y beat x . The candidates are all possible permutations, and we may construct the decision tree shown in Fig. 1, partitioning the set of all permutations into those in which d precedes a ($d > a$) and those in which $a > d$, then continuing with other similar partitions, until we eventually partition into unit sets.

Having specified a set of candidates S at some node it may be possible by looking at its defining conditions to reduce it to a set S' such that $S' \subset S$ and S' contains all solutions in S . We then proceed by partitioning only S' . In the extreme case we can reduce S' to the empty set and stop exploring the branch altogether. To do this for the tree of permutations in Fig. 1 we must agree to explore it in a certain sequence, e.g., always taking the upper

branch first. If we then keep a record of the best sequence found to date and its number of discrepancies we can frequently reject a whole sub-tree by noticing that *any* permutation in the set corresponding to this sub-tree must have more discrepancies than the best to date. The circled numbers

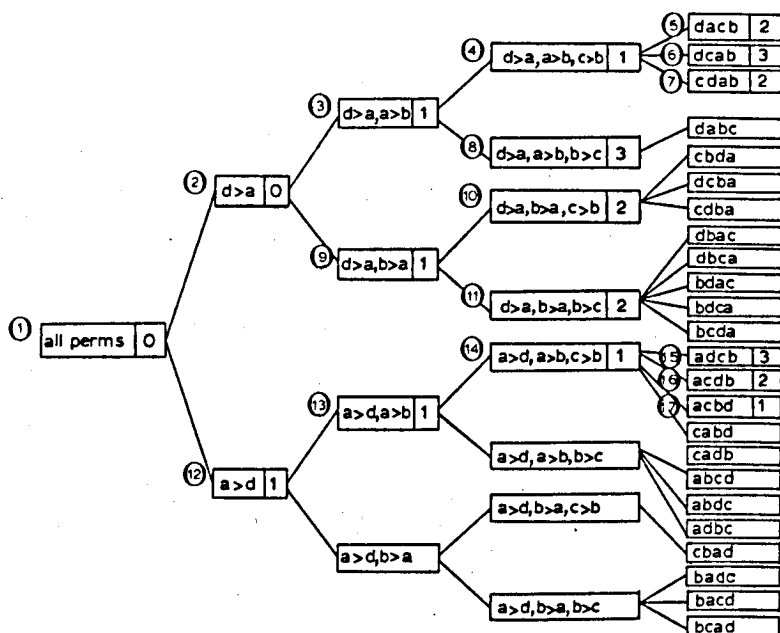


FIG. 1. Decision tree for ranking problem. The numbers in circles show the sequence in which the tree is explored. The numbers in the boxes show the minimum possible number of discrepancies, or the exact number if the box is a tip of the tree.

show the sequence of exploration, unnumbered boxes being ruled out as just explained. The number to the right of each box shows the least number of discrepancies which could possibly occur in any permutation of the set corresponding to that box.

This is an example of a logical exclusion rule which enables us to rule out certain parts of the tree without sacrificing any chances of getting the desired solution. We may also use heuristic exclusion rules, and heuristic principles may also be used to guide us in choosing the sequence of exploration. Another paper in this collection, by J. E. Doran, discusses in detail a particular class of heuristic sequencing rules.

It is often important to choose a useful definition of the decision tree for a given problem, i.e., a useful method of partitioning the candidates. For example, in the permutation problem, instead of the partitioning given above we could have first partitioned according to the first letter of the permutation (a 4-way split), and then according to the second letter (a 3-way split) and so on. This might have led us to some more powerful exclusion rule and reduced the amount of exploration still further. Some problems such as board games with a sequence of moves have a natural tree structure; specify

BURSTALL

the first move, then given this specify the second move and so on. Before basing the algorithm exclusively on this it may be worth examining other decision tree structures which might be imposed, e.g., specifying that certain states will or will not be reached, or specifying the last move first.

Before leaving the permutation example we may note that it would also be amenable to our first method, the method of neighbours. For example, we could define the neighbours of a permutation as all permutations which can be obtained from it by exchanging a pair of adjacent elements. Thus from 'abcd' (2 discrepancies) we could move to any of its neighbours 'bacd' (3 discrepancies), 'acbd' (1 discrepancy) or 'abdc' (3 discrepancies). The best of these being 'acbd' we examine its neighbours, and so on until we fail to secure any improvement. Notice that in this method we would not know whether we had found the optimal solution, whereas using the decision tree method we would be sure of this, an important advantage.

Decision trees for optimising problems which use the current best value to exclude parts of the tree, on the ground that they cannot contain solutions any better than this, have been called 'Branch and Bound' algorithms (Little, Murty, Sweeney & Karel 1963). Little and his colleagues give an algorithm for the 'Travelling Salesman' problem (shortest route through a number of points, visiting each point just once). A number of people have used the method of neighbours for this problem (Reiter & Sherman 1963). The paper by Reiter and Sherman gives a discussion of the general method. Amongst other applications of the decision tree method we may mention assembly line balancing (Tonge 1960), scheduling (Lomnicki 1965), job-sequencing (Burstall 1966b) and solving integer inequalities (Burstall 1965b).

SOME PROBLEMS ABOUT SETS

Successor families

We will now show how the decision tree method can be used for some problems about choosing sets of things. A natural way of going about such problems is to choose the elements of the set one at a time. This is a particular way of imposing a decision tree structure on the problem. We will investigate some problems where it is possible to construct such a decision tree satisfying the essential conditions of the previous section:

(i) $S_1 \cup S_2 \dots \cup S_k = S$ and (ii) $S_i \cap S_j = \text{nil}$.

We suppose that we are given a set U and the solution to our problem is a family of subsets of U . We have a means of recognising whether a given subset belongs to the family or not, and the problem can be solved by generating all subsets of U and testing each with our recogniser. However, the number of subsets to be tested grows rapidly with the number of elements in U and may become hopelessly uneconomic, so we seek a means of generating only the ones belonging to the required family. In some cases we can do this quite easily, namely, cases where we can find a generating rule such that each set of the family (except the empty set) can be obtained by

adding a point to some other set of the family. We will call such a family a 'successor family'. Of course, not all problems about families of sets can be characterised in this way. We give below a more formal definition of 'successor family' and five examples of problems which can be formulated in this way. This definition is *constructive*, allowing us to find an algorithm for building up the required sets, as opposed to a merely *predicative* definition which allows us to recognise the required sets.

However, although the constructive definition can easily be turned into an algorithm we may seek a more efficient algorithm. The difficulty in building up sets of the family element by element is to avoid the twin dangers of omitting a set and of generating the same set twice. We can avoid omissions by considering all possible elements which could be added. But to avoid duplication of sets we will generally have to keep some sort of list of the sets already encountered and test each new set against this list, an expensive business in both computing time and storage. Now under certain conditions we can manage without doing this testing, and we will describe a decision tree algorithm for generating successor families which works quickly and uses very little storage.

Under certain further conditions this algorithm which calls itself recursively twice is equivalent to one with only one recursive call and a loop. It is this latter algorithm which seems to be more obvious when first approaching problems of this type. However, it is a trap (into which I fell at one time) because although plausible it fails for some cases and it is not easy to see at first exactly what has gone wrong.

We will give a proof that the more general decision tree algorithm does generate the sets of a successor family. This may be more convincing than computing results alone, and it may also be of interest in connection with the paper in these proceedings by D. C. Cooper, in which he discusses some proofs that programs are correct.

Let us now return to the definition of successor families of subsets of the given set U .

We will use lower-case letters for elements of U , upper-case letters for subsets of U and script letters for families of subsets of U (we say 'families of subsets' rather than 'sets of subsets' in the hope that it is less confusing). We use '*nil*' for the empty set. Given a function ϕ from subsets of U to subsets of U such that $\phi(S) \cap S = \text{nil}$ we say that a set P is an *immediate successor* of R ($R < P$) if $P = R \cup \{t\}$ for some $t \in \phi(R)$. We say that Q is a *successor* of R ($R \leq Q$) if either $Q = R$ or there is a P such that $R < P$ and $P \leq Q$. We now say that $\mathcal{F}$ is a *successor family* generated by ϕ if

- (i) $\text{nil} \in \mathcal{F}$;
- (ii) for all non-empty $P \in \mathcal{F}$ there is an $R \in \mathcal{F}$ such that $R < P$, i.e., every set in the family is empty or it is the successor of another set in the family.

We say that a successor family $\mathcal{F}$ is 'accessible' if whenever $P \subseteq Q$ and $P \in \mathcal{F}$ and $Q \in \mathcal{F}$ then $P \leq Q$. The decision tree algorithm will apply only to accessible successor families. Accessibility just means that any set of the

family can be derived from any of its subsets which are in the family by successively adding elements.

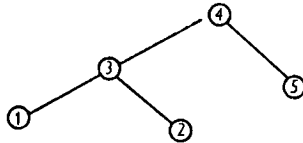
We will now give some examples of accessible successor families.

Example 1: Cost function

Suppose C is a function which assigns a cost to a set R and is monotonically non-decreasing, i.e., if $P \subseteq R$ then $C(P) \leq C(R)$, and we wish to generate all sets with cost less than α . We set $\phi(P) = \{t \in U - R \mid C(R \cup \{t\}) \leq \alpha\}$ and the successor family is just the family of sets required. More generally, if A is a predicate meaning acceptable, and implies $P \subseteq R$ and $A(R) \wedge C(P)$, then we can generate all acceptable sets by taking $\phi(R) = \{t \in U - R \mid A(R \cup \{t\})\}$.

Example 2: Connected subsets of a graph

Suppose that Γ is a symmetric graph over U , i.e., there is a relation $\gamma(r, t)$ between members of U such that $\gamma(r, t)$ implies $\gamma(t, r)$. We say that $S \subseteq U$ is connected if either it is empty or it cannot be partitioned into two sets S_1 and S_2 such that for all $s_1 \in S_1$ and $s_2 \in S_2$ not $\gamma(s_1, s_2)$, e.g., in the graph below $\{1, 2, 3\}$ is connected but $\{1, 2, 4\}$ is not.



The problem is to generate all the connected subsets of Γ . We do this by taking

$$\phi(R) = \{t \in U - R \mid R = \text{nil} \vee \exists r \in R [\gamma(r, t)]\}$$

We will use this example in the 'Electricity network design problem, (p. 75).

Example 3: Strongly connected subsets of a graph

Suppose Γ as in the previous example, we call a subset S strongly connected if it is empty or for all $s \in S$ and $t \in S$, $\gamma(s, t)$. To generate all the strongly connected subsets of Γ we take

$$\phi(R) = \{t \in U - R \mid R = \text{nil} \vee \forall r \in R [\gamma(r, t)]\}$$

Example 4: Non-redundant covers of a set

Let σ be a set and U be a family of subsets of σ . We define

$$\text{sumset}(S) = \{\alpha \mid \exists s \in S [\alpha \in s]\}$$

i.e., the members of members of S .

We say that S is a *cover* of σ if $\sigma \subseteq \text{sumset}(S)$, and that S is a *non-redundant cover* of σ if no proper subset of S is a cover of σ . We wish to generate all

the non-redundant covers of σ . All such sets (and some that are not covers) are contained in the successor family with respect to

$$\phi(R) = \{t \in U - R \mid \text{sumset}(R) \subset \text{sumset}(R \cup \{t\})\}$$

For an example of the use of an algorithm similar to this in a job sequencing problem see Burstall (1965b).

Example 5: Linear combinations of points

Let U be a set of points in n -space. We say that a point $q \in U$ is between points p and $r \in U$, '*Between* (p, q, r)', if q is in the line segment joining p and r . Given a basis set of points $B \subseteq U$ we say that a set $R \subseteq U - B$ is derived from B if any point in R is between two other points in $R \cup B$. To obtain the family of sets derived from B we take

$$\phi(R) = \{t \in U - R - B \mid \exists s \in R \cup B \exists u \in R \cup B [\text{Between}(s, t, u)]\}$$

Between is a particular three-place relation, but the method generalises quite easily to any other three-place relation and, indeed, to any k -place relation.

The Algorithm

What is the connection between successor families and decision trees? How can we use it to get an algorithm?

Suppose that we have already chosen some elements of U , say a set R , and wish to add further elements. A natural approach would be to associate a node of the decision tree with R . We could say that it corresponds to that set of candidates (family of subsets of U) defined as $\mathcal{F} = \{P \subseteq U \mid R \subseteq P\}$. However, this is not good enough since two different sets R and R' , would have overlapping families, both containing $R \cup R'$, for example, and this contradicts the condition of 'successor families': $S_i \cap S_j = \text{nil}$. The trick is to associate with a node a pair of disjoint sets R and S , R being the elements of U we have chosen and S being the elements of U which we may choose. We then associate with the node labelled ' (R, S) ' the set of candidates $h(R, S) = \{P \subseteq U \mid R \subseteq P \wedge P \subseteq R \cup S\}$, i.e., sets obtained by adding to R elements of S . Now if we choose some element $t \in S$ we may either add it to R or not. This partitions $h(R, S)$ into two subsets:

$$h(R \cup \{t\}, S - \{t\}) \text{ i.e. } \{P \subseteq U \mid R \cup \{t\} \subseteq P \wedge P \subseteq [R \cup \{t\}] \cup [S - \{t\}]\}$$

and $h(R, S - \{t\}) \text{ i.e. } \{P \subseteq U \mid R \subseteq P \wedge P \subseteq R \cup [S - \{t\}]\}$

It is easy to verify that these sets are disjoint and that their union is $h(R, S)$.

Now $h(\text{nil}, U)$ contains all subsets of U and $h(R, \text{nil})$ contains just the set R so that the function h defines the nodes of a complete decision tree.

Note, however, that we have defined h *predicatively* by a property of the sets belonging to it. This is simple on paper but computationally inefficient. The equivalent constructive definition is (see Appendix 1 for notation):

$$\begin{aligned} g(R, S) &= \text{null}(S) \rightarrow \{R\}, \\ &[\text{let } t = \text{any}(S); \\ &g(R \cup \{t\}, S - \{t\}) \cup g(R, S - \{t\})] \end{aligned}$$

Now we can see how the successor families can be used. Instead of choosing any element $t \in S$ we can restrict ourselves to choosing an element $t \in \phi(R) \cap S$ since this is the way to get the successors of R . Does it matter which element t we choose? If we are dealing with an *accessible* successor family the answer is no; this is precisely why we introduced the idea of accessibility. To put it another way, provided the successor family is accessible we can build up any member of it element by element *in any order*. Before justifying these assertions we will present our algorithm as a recursive function (see Appendix 1 for notation). We define a function $f(U, \phi)$, whose value is the accessible successor family over U with respect to ϕ . The definition uses an auxiliary function $g(R, S)$.

let $f(U, \phi) = g(\text{nil}, U)$
 where $\text{rec } g(R, S) = [\text{let } T = S \cap \phi(R);$
 $\text{null}(T) \rightarrow \{R\},$
 $[\text{let } t = \text{any}(T);$
 $g(R \cup \{t\}, S - \{t\}) \cup g(R, S - \{t\})]]$

Theorem

If $\mathcal{F}$ is the successor family over U with respect to ϕ and $\mathcal{F}$ is accessible then $f(U, \phi) = \mathcal{F}$.

Proof of theorem

We first of all note that $g(R, S)$ is defined for all R and S . If S has no elements then for any R , $g(R, S) = \{R\}$ since T is empty. If S is not empty then either T is empty, in which case $g(R, S) = \{R\}$ or $g(R, S)$ is defined in terms of $S - \{t\}$ which has fewer elements than S . Since the 'function' *any* is supposed to pick any element of T it is not immediately clear that the value of $g(R, S)$ is independent of the element chosen. However, this is in fact the case as can be seen from a lemma which we now prove:

Lemma: If $R \in \mathcal{F}$ and $R \cap S = \text{nil}$ then $g(R, S) = h(R, S) \cap \mathcal{F}$, where as above

$$h(R, S) = \{P \subseteq U \mid R \subseteq P \wedge P \subseteq R \cup S\}.$$

Proof of lemma: Suppose $R \in \mathcal{F}$ and $R \cap S = \text{nil}$.

Case (i). Suppose $S \cap \phi(R) = \text{nil}$. Then $g(R, S) = \{R\}$ by the definition of g .

Now $R \in \mathcal{F}$ by hypothesis and $R \in h(R, S)$ by the definition of $h(R, S)$. Therefore $\{R\} \subseteq h(R, S) \cap \mathcal{F}$.

We now show that $h(R, S) \cap \mathcal{F} \subseteq \{R\}$.

Suppose $P \in h(R, S) \cap \mathcal{F}$. Then $R \subseteq P \subseteq R \cup S$ by definition of h .

But since $\mathcal{F}$ is accessible, $R \subseteq P$ implies $R \ll P$, i.e., $P = R$ or else there is a Q such that $R < Q \ll P$.

But by definition of immediate successor $Q = R \cup \{t\}$ where $t \in \phi(R)$.

It also follows easily from the definition of successor that $Q \ll P$ implies $Q \subseteq P$ which gives $Q \subseteq R \cup S$ and hence since $R \cap S = \text{nil}$, $t \in S$.

We have that $t \in S \cap \phi(R)$ but this is empty by hypothesis, therefore there is no such Q and hence if $P \neq R$ then $P \notin h(R, S) \cap \mathcal{F}$, i.e., $h(R, S) \cap \mathcal{F} \subseteq \{R\}$.

We now have $h(R, S) \cap \mathcal{F} = \{R\} = g(R, S)$.

Case (ii). Suppose $S \cap \phi(R)$ is not empty. We now prove the lemma by induction on the number of elements in S .

If S is empty then $S \cap \phi(R)$ is empty and Case (i) above applies.

We now show that the lemma holds for $g(R, S)$ for any R , assuming as induction hypothesis that it holds for $g(R', S - \{t\})$ for any R' and any $t \in S$.

Suppose $t \in S \cap \phi(R)$, $R \in \mathcal{F}$ and $S \cap R = \text{nil}$.

Then $g(R, S) = g(R \cup \{t\}, S - \{t\}) \cup g(R, S - \{t\})$.

But for both these g expressions the conditions of the induction hypothesis hold, i.e., $R \cup \{t\} \in \mathcal{F}$ since $R < R \cup \{t\}$, and $[R \cup \{t\}] \cap [S - \{t\}] = \text{nil}$, also $R \in \mathcal{F}$ and $R \cap [S - \{t\}] = \text{nil}$.

Therefore

$$\begin{aligned} g(R, S) &= [h(R \cup \{t\}, S - \{t\}) \cap \mathcal{F}] \cup [h(R, S - \{t\}) \cap \mathcal{F}] \\ &= [h(R \cup \{t\}, S - \{t\}) \cup h(R, S - \{t\})] \cap \mathcal{F} \\ &= h(R, S) \cap \mathcal{F} \end{aligned}$$

by the identity mentioned when h was first defined.

This completes the proof of the lemma.

Now we had $f(U, \phi) = g(\text{nil}, U)$.

But this is $h(\text{nil}, U) \cap \mathcal{F}$ and we observed before that $h(\text{nil}, U)$ is the set of all subsets of U and hence $h(\text{nil}, U) \cap \mathcal{F} = \mathcal{F}$.

This concludes the proof.

In all the five examples given above the solutions could be characterised as an accessible successor family with respect to some stated ϕ , so we can use the function f to generate the solutions to any of these examples. In Appendix 2 we give an Algol program corresponding to f . We also give a slightly simpler Algol program corresponding to a variant of f , which may be used when a certain extra condition holds, namely: if $P \subseteq R$ then $\phi(R) \subseteq \phi(P)$. This condition holds for Examples 1, 3 and 4, but it does not hold in general for Examples 2 or 5.

We can make a useful generalisation of the function f to deal with what we may call 'collections' or unordered lists. These are like sets in having no order of the elements, but elements are allowed to occur more than once, e.g., $\{a, b, b, d, d\}$ is a collection and the same as $\{b, a, d, d, b\}$. Another representation would be $\langle 1, 2, 0, 2 \rangle$ associated with the alphabet $\langle a, b, c, d \rangle$. We assume operations analogous to unit set and union in the natural way, e.g., $\{a, b, b\} \cup \{a\} = \{a, a, b, b\}$. Then the analogue of $f(U, \phi)$ is $f'(U, \phi)$ where f' is a new function, U is a set and ϕ is a successor function carrying a collection into a set; f' produces a set of collections. The definition of f' is formally the same as that of f except that the term $g(R \cup \{t\}, S - \{t\})$ is replaced by $g(R \cup \{t\}, S)$ since t may be added more than once to the collection R . Example 1 above carries over in a natural way to collections and

we shall use this in the 'Electricity network design problem'. Example 4 also carries over to the case where σ is a collection.

AN ELECTRICITY NETWORK DESIGN PROBLEM

The problem

As an example of the use of tree-searching methods I have developed an algorithm for designing electricity distribution networks. This research was assisted by discussions with engineers of the Central Electricity Generating Board, particularly U. G. Knight of the H.Q. System Planning Branch. For a more detailed account see Burstall (1965*a*).

This work is concerned with the design of national grid and city supply networks, a complex task in which computers at present play only an ancillary role. My program is able to create designs to meet the fundamental loading and security requirements. These designs can then be scrutinised by engineers and tested further by other methods. A number of problems have been run on an Elliott 503 computer with encouraging results.

The aim is to design a network which will supply a number of points consuming power from other points which produce it, by specifying the number of standard circuits to be laid down between each pair of points, usually between 0 and 6 circuits. The points may be individual stations or they may be whole areas considered as an approximation to be single points. The network must also have a very high probability of functioning even if some circuits break or some points consume more than the expected amount. This security requirement poses a difficult problem in probability theory. I have followed a method used by the CEGB. This is to specify the number of circuits required by each subset of the points which is a net consumer of power, using a simple formula based on the estimated power requirements of the individual points in the set and including a safety margin. This gives a constraint for each consumer subset.

We are given an estimate of the capital cost per circuit between each pair of points and asked to produce a solution at minimum cost. In practice we seek only to produce solutions close to this minimum.

For computational convenience we neglect losses and let one of the points be a slack. We assign to this slack point consumption or production so that the total consumption and total production just balance. This means that half of the subsets will be net consumers of power and their complements net producers. In what follows we will by convention associate the constraint with either the consumer set or its complement, whichever does not contain the slack point.

We denote the slack point by 0 and the set of all other points by U . If S is a subset of U we write S^0 for $S \cup \{0\}$.

Let x_{ij} be the number of circuits between points i and j and c_{ij} be the cost of each circuit between i and j .

Let $g(s)$ be the number of circuits required by the set S , i.e., circuits between S and $U^0 - S$.

Then the constraints are

$$\sum_{i \in S} \sum_{j \in U^* - S} x_{ij} \geq g(S) \quad \text{for each } S \subseteq U \quad (1)$$

$$x_{ij} \geq 0 \quad \text{for each } i \in U, j \in U \quad (2)$$

The total cost to be minimised is

$$Z = \sum_{i,j \in U} c_{ij} x_{ij}$$

In some practical cases a non-linear cost function occurs. The program can treat this but we do not discuss it here.

This is an integer linear programming problem and has been tackled in this way by U. G. Knight of CEGB who first formulated it in these terms. Using an IBM 7094 and including only a sample of the constraints he has successfully tackled some problems with 9 points; on another problem of this size, however, the integer linear programming program seemed to be making no progress and was stopped after about 5 minutes. The limitations of the integer linear programming method should be apparent from the following table:

Number of Points	Number of Variables	Number of Constraints
5	10	15
10	45	511
15	105	16 383
20	190	524 287

(The number of constraints does not include the non-negativity constraints.)

The computer program using tree search techniques has been run successfully on programs up to 16 points in size (32 767 constraints), for which it takes 2 to 3 hours on an Elliott 503 (7.5 microseconds for fixed point add and similar simple operations). On a 9-point (255 constraint) problem it obtained the same solution as integer linear programming. For the larger problems no exact standard of comparison was available but the results were quite consistent when repeated with variations in the strategy, and they agreed well with the kind of network previously arrived at by engineers using intuitive design methods.

The optimising method

The difficulty of the problem lies mainly in recognising whether a solution is feasible. In an ordinary network flow problem we only have to check that each consumer point has enough circuits into it with sufficient total flow to satisfy its demand. In our problem, however, each consumer *subset* of points has to have enough circuits. Instead of checking say $\frac{1}{2}n$ points we must check $\frac{1}{2}2^n$ subsets. A solution can look very convincing until a careful and tedious check uncovers a single deficient subset. On problems with 15 or so points even to check 100 per cent of the subsets is virtually

impossible without a computer; the engineer must use his intuition to diagnose any weak spots. He supports this by some analytical checks.

The basic method which we use is the one we have called the method of neighbouring solutions. We start with a feasible solution, either one found by hand experimentation or one formed by joining each point to the slack with a sufficient number of circuits (mostly a very expensive solution but one which can easily be shown to be feasible). We then find a succession of solutions, each one feasible and cheaper than the previous one, until further efforts yield no more improvement. Each solution is a neighbour of the previous one in the following sense: it is obtained from it by adding 0, 1 or 2 circuits and deleting some other circuits.

Even with as few as 10 points in the problem it is utterly impracticable to examine all the neighbours of a given solution. For n points there are $\frac{1}{2}n(n-1)$ pairs of points and therefore $\frac{1}{2}[\frac{1}{2}n(n-1)][\frac{1}{2}n(n-1)+1]$ ways of adding 2 circuits. Each of these could lead to several deletions, say half a dozen different combinations of deletions to be conservative. For each of these we must check 2^{n-1} subsets for feasibility. The total number of subsets to be checked is thus of the order of $6n^4 2^{n-4}$. If $n=10$ this gives nearly 4 million subsets, if $n=15$ it gives about 600 million (the 503 takes a milli-second or so to check a subset). All this is just to make one slight improvement in the solution!

Consequently we decide to look at a limited selection of the neighbours and try to simplify the checking process so as not to go through the whole business afresh for each neighbour. To do this we choose some subset of U , say F , which we will call the 'focus set', since we intend to focus attention on it for the time being. We restrict the neighbouring solutions which we consider to those which involve adding or deleting circuits within F^0 , i.e., only involving points of the focus set or the slack point. We take the best of these neighbours and repeat the process considering its neighbours. When no improvements occur we choose another focus set, and so on until a sufficient number of focus sets have been tried to cover each point several times over and no further improvement seems likely. By doing this we can also simplify the checking by setting up a small number of special constraints for the set F , which, so to speak, synopsis all the constraints of the whole problem. Once set up, these enable us to make changes at will within F^0 disregarding the constraints of the whole problem and checking only these special constraints. This idea is crucial for the success of the program. We call it 'constraint mapping'. If U^0 has n points and F^0 has k we can temporarily replace the 2^{n-1} constraints associated with subsets of U by 2^{k-1} constraints associated with subsets of F , and these are all we need to worry about for additions and deletions within F^0 .

We now show how these special constraints are to be set up. Suppose that x_{ij} is the current solution. We first define a quantity for each $T \subseteq U$

$$spare(T, U) = \sum_{i \in T} \sum_{j \in (U^0 - T)} x_{ij} - g(T)$$

This is the number of circuits that T has over and above its requirement. For feasibility $\text{spare}(S, U) \geq 0$ for all S (by equation (1) above).

We now associate with each subset $S \subseteq F$ a family of sets,

$$\mathcal{F}(S, F, U) = \{T \mid T \subseteq U \text{ and } T \cap F = S\},$$

i.e., sets consisting of S together with some other points outside the focus set.

Notice that if we delete a circuit which joins a point in S to a point in $F^0 - S$ then for each $T \in \mathcal{F}(S, F, U)$, $\text{spare}(T, U)$ decreases by 1. This is the reason for defining the family $\mathcal{F}$, all sets in it are equally affected by changes within the focus set. To find how many such circuits we can safely delete we define

$$\text{spare}(S, F) = \min_{T \in \mathcal{F}(S, F, U)} \text{spare}(T, U)$$

So long as $\text{spare}(S, F) > 0$ we have not introduced an infeasibility by such deletions. Similarly adding a circuit from a point in S to a point in $F^0 - S$ gives an extra circuit to each $T \in \mathcal{F}(S, F, U)$ and hence increases $\text{spare}(S, F)$ by 1.

We see now that we can add or delete circuits within F^0 and it is sufficient to watch the effect on $\text{spare}(S, F)$ for each $S \subseteq F$.

More formally if $\text{spare}(S, F)$ is calculated from x_{ij} and we change x_{ij} to $x_{ij} + \delta x_{ij}$, keeping $\delta x_{ij} = 0$ unless $i, j \in F^0$, feasibility is maintained if and only if

$$\sum_{i \in S} \sum_{j \in F^0 - S} \delta x_{ij} \geq -\text{spare}(S, F) \text{ for all } S \subseteq F \quad (3)$$

To maintain non-negativity of x_{ij} we also need

$$\delta x_{ij} \geq -x_{ij} \text{ for all } i, j \in \text{pairs}(F^0) \quad (4)$$

The increase in cost is

$$\delta Z = \sum_{i, j \in F^0} c_{ij} \delta x_{ij}$$

To find the best neighbour we minimise this cost, hoping for a negative value, i.e., an improvement.

These constraints again define an integer linear programming problem. In fact we restrict ourselves to changes involving not more than two additions and tackle the problem by a decision tree method. Since we are on a subproblem even a guaranteed optimal solution to the subproblem would not be sure to give an optimal solution to the whole problem, hence our preference for the less powerful but we hope faster method. Another reason is that we feel the decision tree method is more flexible in dealing with extra *ad hoc* constraints or complications in the cost function. Having got a better solution with not more than two additions we try again, and again, until the possibilities of the focus set are exhausted. We must then set up the constraints for another focus set and continue.

Using decision trees to find the best neighbour

To find the best neighbour, restricting ourselves to changes within the focus set, we use one of the algorithms described in 'Some problems about

sets' (p. 69). The first example in that section gave a method for generating a family of sets which each had a property of 'acceptability'. Now for 'sets' read 'collections' and for 'acceptability' read 'feasibility'; we can apply the algorithms to our current problem.

If $+ij$ means add a circuit between i and j , and $-ij$ means delete a circuit between i and j we can describe a combination of changes by a collection, e.g., $\{+12, +12, -13, -24, -15\}$. The solution before alteration, x_{ij} , corresponds to the empty collection.

We shall use italic bold type for symbols denoting collections to distinguish them from sets. If C is a collection of additions and deletions, we denote the collection of all additions in it by C^+ and that of all deletions in it by C^- . Thus $C = C^+ \cup C^-$. Similarly for sets of additions or deletions. We denote the set of all additions and deletions in F^0 by V , i.e., V contains two elements for each pair of points in F^0 , one with a $+$ and one with a $-$.

Now if we allow up to m additions (in practice $m=1$ or 2) we can characterise the set of possible changes by an accessible successor family with

$$\phi(R) = \text{size}(R^+) < m \rightarrow V^+, \{v \in V^- \mid \text{Feasible}(R \cup \{v\}) \wedge \forall v \in R^+\}$$

where *size* is a function which counts the number of elements in a collection and *feasible* is a predicate which takes the value true if its argument is a feasible collection of changes, i.e., it satisfies the constraint inequalities 3 and 4 above. If R contains less than m additions we may add any circuit, otherwise we can always delete any circuit provided that the deletion is feasible and we do not simply delete a circuit which has just been added (a waste of time).

We notice that ϕ has the property: if $P \subseteq R$ then $\phi(R) \subseteq \phi(P)$. This means that we can use the slightly simpler algorithm described in Appendix 2. Since a lot of the computation is taken up in testing for feasibility a further refinement is to keep a record of the successor elements $\phi(P)$ and pass this set on as an extra parameter of g ; since we know that $\phi(R) \subseteq \phi(P)$ we can determine $\phi(R)$ more quickly by ignoring all points not in $\phi(P)$. We mention this to show how the general algorithm can be improved by adapting it to the circumstances of the particular case.

Checking only connected subsets

It still happens that each time we start optimising a new focus set we have to set up the special constraints for its subsets by examining the constraints for every subset of U . This only happens say ten or twenty times instead of many thousands in the naïve method originally discussed, but it is still tedious. It is easy to prove that if the function g which specifies the number of circuits needed has the following property we need only check those subsets which are connected (in the sense of Example 3 of 'Some problems about sets' (p. 69):

$$g(S \cup T) \leq g(S) + g(T)$$

By the nature of the security rules this condition does in fact hold.

The argument is that if a subset P is not connected it must be divisible into two sets S and T such that there are no circuits between S and T . The number of circuits into P is then the sum of the numbers into S and into T ; but its requirement is less than or equal to the sum of the requirements of S and T if the above condition holds. Therefore to check P it is sufficient to check S and T . Repeating the argument if S or T is not connected we see in the end that it is sufficient to check only connected subsets.

Now using Example 3 of 'Some problems about sets' (p. 69) we know how to generate just the connected subsets of the symmetric graph defined by ' $\gamma(i, j)$ if and only if $x_{ij} > 0$ ', and we do not need to examine all the disconnected subsets. On a 16-point problem this saves a factor of about 10 in the checking process. The saving increases with the problem size.

Results obtained with the program

Design engineers supplied data for several design problems, and a number of other problems have been invented using random numbers to generate problems broadly similar to those supplied.

Some experimenting was done to find the effect of different methods of choosing the sequence of focus sets. One method was to choose them at random, another was to choose intuitively a sequence which seemed to be suitable bearing in mind the following criteria:

- (i) The points in a focus set should be fairly close together.
- (ii) Each point should occur at least twice at different stages in the optimising process.

TABLE 1

Comparison of costs for intuitive and random choice of focus sets

Job number	210	220	230	240	250	260
No. of points	10	9	11	8	8	9
Intuitive short run	620	528	632	593	506	502
Intuitive short run	620	513	635	593	506	522
Intuitive longer run	620	513	625	593	506	509
Random short run	636	513	631	591	525	509
Random short run	632	524	657	593	506	509
Random longer run	620	513	637	581	506	509
Average intuitive	620	518	631	593	506	511
Average random	629	517	642	588	512	509

Running times varied between 3 minutes and 22 minutes.

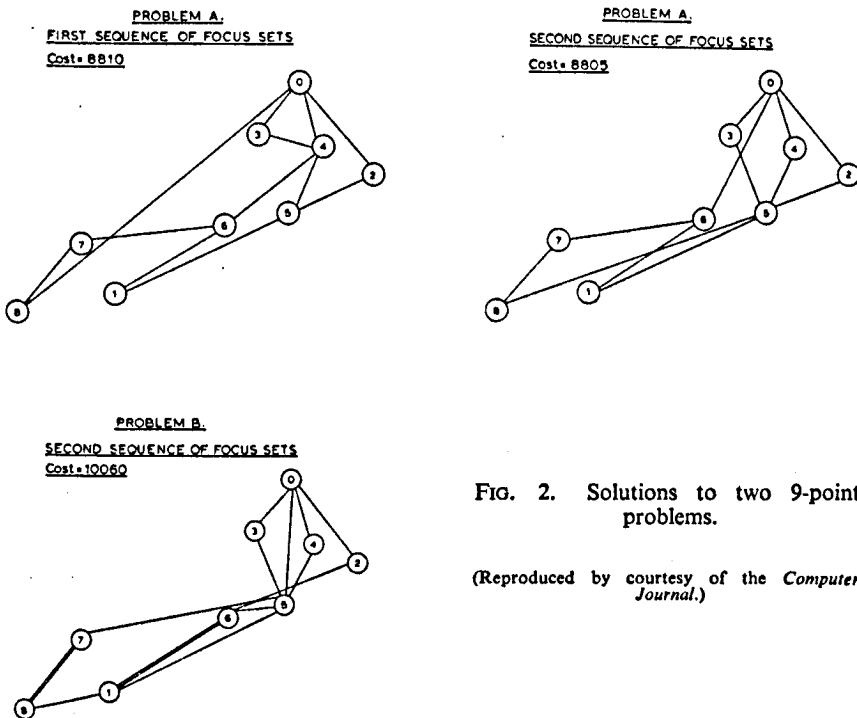
Table 1 shows the results of trying six problems invented randomly. In these problems only one point was a producer and the others were consumers of power. Three runs were done choosing focus sets intuitively, and three

BURSTALL

more choosing them at random. The focus sets had four or five points each. There is a slight indication that the intuitive selection was better than random on average but not enough to be of any significance.

A 9-point problem supplied by a design engineer was tried. He had run this by integer linear programming using two slightly different sets of consumption data. For the first data set, two runs were done using different sequences of intuitively chosen focus sets. The first produced a cost 0.5 per cent higher than the linear programming solution and the second obtained a solution identical with that obtained by linear programming. The times taken were 10 and 11 minutes. For the second data set only one run was done again using intuitively chosen focus sets and the results were identical with those found by linear programming. The time taken was 27 minutes. The results are shown in Fig. 2.

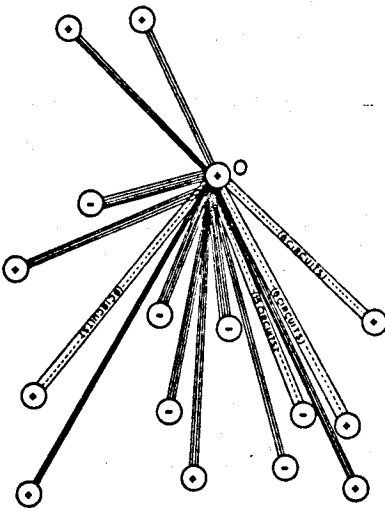
Fig. 3 shows the results of trying the program on a 16-point problem provided by system design engineers. Two runs were done using quite different intuitively chosen sets of focus points. Each point represents an area rather than an individual station. The results agreed to within 0.2 per cent in cost. No hand-produced solution was available for this set of data, but the solution was shown to engineers who remarked upon the similarity of the computer solution to their more detailed design studies for the same system. The times taken were 1 hour 48 minutes and 2 hours 53 minutes.



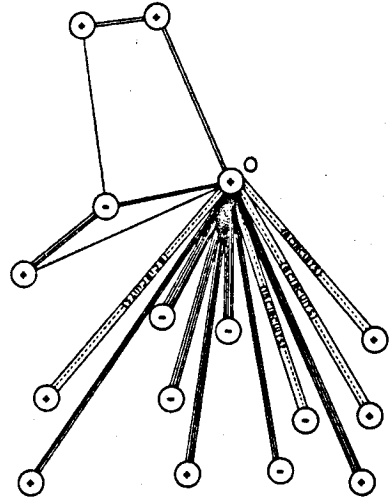
MACHINE LEARNING AND HEURISTIC PROGRAMMING

Work is now proceeding to evaluate the performance of the program further and to improve it in various ways to enable larger problems to be tackled.

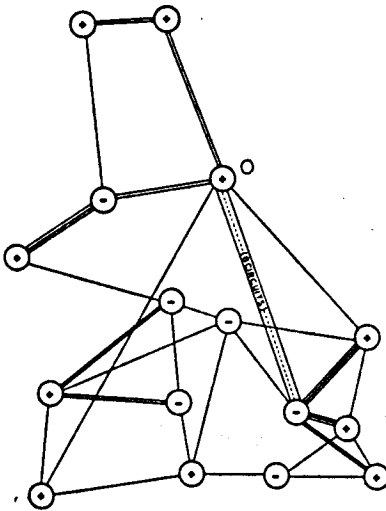
(a) Cost = 354.7 (initial solution)



(b) Cost = 331.4



(c) Cost = 124.0



(d) Cost = 111.6

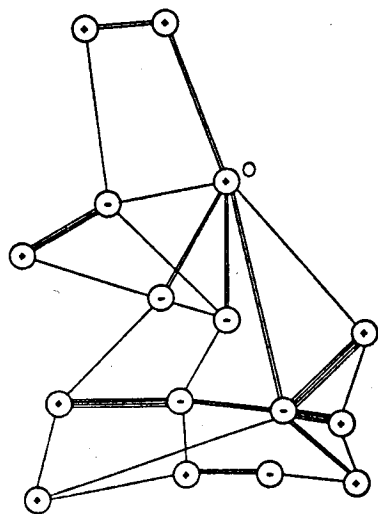


FIG. 3. Stages in the solution of a 16-point problem. — indicates a supply point, + indicates a demand point.

(Reproduced by courtesy of the *Computer Journal*.)

APPENDIX 1: NOTATION

The notation used for the recursive functions is broadly the same as in CPL (Barron, Buxton, Hartley, Nixon & Strachey 1963) or to a lesser extent that of the LISP meta-language (McCarthy 1960):

- (i) Conditional expressions $P \rightarrow X, Y$ are equivalent to the Algol expressions *if P then X else Y*.
- (ii) $\text{let } x = E; E'$ means that x has the value E in E'
 $\text{let } f(y) = E; E'$ means that f has the value $(\lambda y. E)$ in E' .
- (iii) E' where $x = E$ is the same as $\text{let } f(x) = E; E'$
 E' where $f(y) = E$ is the same as $\text{let } f(y) = E; E'$.
- (iv) Parentheses are used to enclose the parameters of functions and brackets for grouping terms.
- (v) $\{ \}, \cup, \cap, \in, -, \text{nil}$ have their usual set theory meanings.
- (vi) *null* (T) is true just if T is the empty set.
- (vii) *any* (T) is an arbitrary member of the set T .
- (viii) *rec.* in a function definition indicates recursion.

APPENDIX 2: ALGOL PROGRAMS

The following Algol program corresponds to the function f . The sets are represented by Boolean arrays. However the result is not a set of sets represented in this way, which would be wasteful of storage, but rather each set generated is presented as a Boolean array to the procedure *output* provided by the calling program to use the results. The parameter n is the dimension of U . The parameter function ϕ , is given the set R as a Boolean array and an element t ; it takes the value true if $t \in \phi(R)$. We have taken advantage of some features of the auxiliary function g to save space by making its parameters into non-locals.

```

procedure  $f(U, n, \phi)$  result: (output); value n;
  Boolean array  $U$ ; integer  $n$ ; Boolean procedure  $\phi$ ; procedure output;
  begin Boolean array  $R, S[1:n]$ ; integer  $i$ ;
    procedure  $g$ ;
      begin integer  $t$ ;
        for  $t := 1$  step 1 until  $n$  do
          if  $\phi(R, t) \wedge S[t]$  then goto  $Tnonnull$ ;
          output ( $R$ ); goto exit;
         $Tnonnull: S[t] := \text{false}; R[t] := \text{true}; g$ ;
         $R[t] := \text{false}; g; S[t] := \text{true};$ 
      exit: end g;
      for  $i := 1$  step 1 until  $n$  do begin  $R[i] := \text{false}; S[i] := U[i]$  end;
     $g$ 
  end  $f$ ;

```

In some cases the following condition may be satisfied:

$$\text{if } P \subseteq R \text{ then } \phi(R) \subseteq \phi(P)$$

This enables us to use instead of f the function f_1 defined by:

```

let  $f_1(U, \phi) = g(\text{nil}, U)$ 
where rec.  $g(R, S) = \text{null}(S) \rightarrow \{R\}$ ,
  [let  $t = \text{any}(S)$ ;
  [ $t \in \phi(R) \rightarrow g(R \cup \{t\}, S - \{t\}), \text{nil}] \cup g(R, S - \{t\})$ ]

```

This function has the characteristic that the elements of S can be deleted one at a time in any order we choose. This means that S can be represented in Algol by

an integer s with the convention that $i \in S$ if and only if $i > s$. This in turn enables us to replace one of the recursive calls by an iteration without copying any arrays.

```

procedure  $f1(U, n, phi)$  result: (output); value  $n$ ;
  Boolean array  $U$ ; integer  $n$ ; Boolean procedure  $phi$ ; procedure  $output$ ;
  begin Boolean array  $R[1 : n]$ ; integer  $i$ ;
    procedure  $g(s)$ ; value  $s$ ; integer  $s$ ;
    begin integer  $t$ ;
      for  $t := s$  step 1 until  $n$  do
        if  $phi(R \upharpoonright t) \wedge U[t]$  then begin  $R[t] := \text{true}$ ;  $g(t+1)$ ;
           $R[t] := \text{false}$ 
        end;
       $output(R)$ ;
    end  $g$ ;
    for  $i := 1$  step 1 until  $n$  do  $R[i] := \text{false}$ ;
   $g(1)$ 
end  $f1$ ;
  
```

The procedures for dealing with collections can be coded in a very similar manner, replacing Boolean arrays by integer arrays.

REFERENCES

- Barron, D. W., Buxton, J. N., Hartley, D. F., Nixon, E., & Strachey, C. (1963). The main features of CPL. *Comp. J.*, 6, 134-143.
- Burstall, R. M. (1965a). Computer design of electricity networks by a heuristic method. *Comp. J.* (in press).
- Burstall, R. M. (1965b). A tree searching method for solving integer linear inequalities. *Experimental Programming Report*. No. 10. Edinburgh: Experimental Programming Unit, University of Edinburgh.
- Burstall, R. M. (1966). A heuristic method for a job scheduling problem. *Opl. Res. Q.*, 17, 291-304.
- Golomb, S. W., & Baumert, L. D. (1965). Backtrack programming. *J. Ass. comp. Mach.*, 12, 516-524.
- Knight, U. G. W. (1960a). Logical design of electrical networks. *J. Instn elect. Engrs*, 6, 228-230.
- Knight, U. G. W. (1960b). The logical design of electrical networks using linear programming methods. *Proc. Instn elect. Engrs*, 107A, 306-319.
- Knight, U. G. W. (1963). Economic design and operation of power systems. *Elect. Rev., Lond.*, 92, 124-130; 92, 163-166.
- Little, D. D. C., Murty, K. G., Sweeney, D. W., & Karel, C. (1963). An algorithm for the travelling salesman problem. *Ops. Res.*, 116, 972-989.
- Lomnicki, Z. A. (1965). A 'branch-and-bound' algorithm for the exact solution of the three-machine scheduling problem. *Opl. Res. Q.*, 16, 89-100.
- McCarthy, J. (1960). Recursive functions of symbolic expressions. *Communs Ass. comput. Mach.*, 3, 184-195.
- Newell, A., Shaw, J. C., & Simon, H. A. (1959). Report on a general problem-solving program. *Proc. of the International Conference on Information Processing*. pp. 256-264. Paris: UNESCO House.

- Newell, A., & Simon, H. A. (1961). GPS, a program that simulates human thought, in *Lernende Automaten*. Munich: R. Oldenbourg KG.
- Reiter, S., & Sherman, G. R. (1963). Discrete optimising. *Institute for Quantitative Research in Economics and Management, Purdue University, Institute Paper No. 37*.
- Tonge, F. M. (1960). Summary of a heuristic line balancing procedure. *Management Science*, 7, 21-42.

6

EXPERIMENTS WITH A LEARNING COMPONENT IN A GO-MOKU PLAYING PROGRAM

E. W. ELCOCK and A. M. MURRAY
UNIVERSITY OF ABERDEEN

INTRODUCTION

This paper is a report on some preliminary work undertaken as part of a longer term study of the problems which arise in designing and implementing digital computer programs which 'learn'.

A program has been written which learns to play the board game 'Go-Moku' using a particular learning mechanism to be described later. The program is to be regarded as an experimental tool by means of which the particular learning mechanism can be investigated in some depth.

Go-Moku is a simple but not a trivial game with an intellectual content comparable with a game of draughts (checkers). Opinions have sometimes been expressed that there is nothing to be learnt (no pun intended!) by programming simple games. With this we disagree. Present knowledge of programming learning is such that it is useful to experiment with programs operating in a simple task environment. It is not so much what game the program learns as how it learns it. It is emphasised that the object of the present work is not to write a program which plays a difficult game better than anyone or anything has played it before, but to isolate and investigate particular aspects of a learning process which might be valid over a range of ill-structured problems. For the record, however, the current learning programs learn to play a good (basically defensive) game. The modifications currently being made to the program should give it a learning capacity to become unbeatable.

GO-MOKU: THE RULES AND AN EXAMPLE

For readers unfamiliar with the game of Go-Moku, the rules are given below. An example game is given to illustrate the kind of motivation that might lie behind move making. It is hoped that this will provide a background for discussion of the learning program which follows.

The game is played on a 19×19 square mesh (Fig. 1). Player $b(w)$ has a supply of indistinguishable black (white) pieces. The players take it in turns to play a piece on a lattice point. The winner is the first to complete a

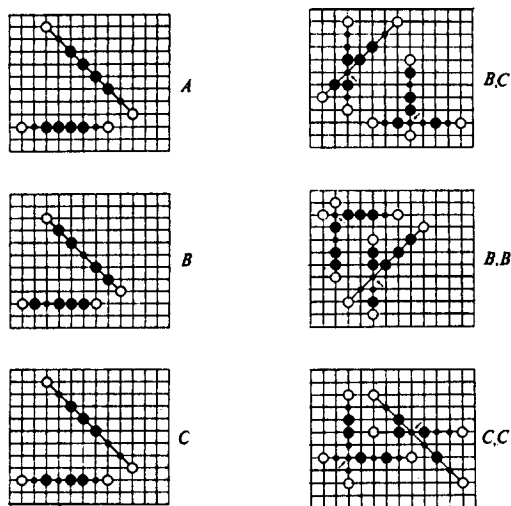


FIG. 1.

(horizontal, vertical or diagonal) line of five and only five adjacent pieces of his colour.

One or two elementary remarks might be useful. Pattern A (Fig. 1) is a winning pattern. It can be made into a five by adding a piece at either end: the opponent cannot kill both. It follows that patterns such as B and C are *forcing* patterns: the opponent must reply to them immediately. It follows that patterns such as B,B C,C and B,C are winning patterns with black to play. By playing moves such as those indicated he simultaneously creates two forcing patterns.

At a more advanced level of play the advantage of being first player is compensated for by adding a further rule, to the effect that the opening player may not play a move which is a component of two forcing patterns of the type C, i.e., moves of the kind indicated in the patterns C,C.

This additional rule has not been implemented in the work reported here. And now for some comments on the example game illustrated in Fig. 2.

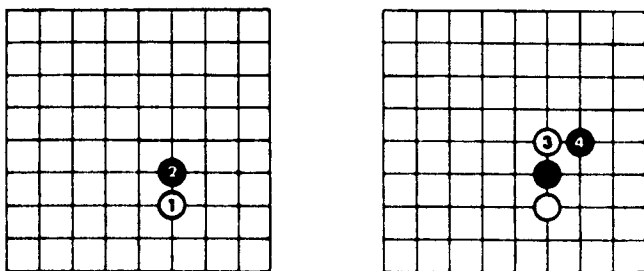


FIG. 2.

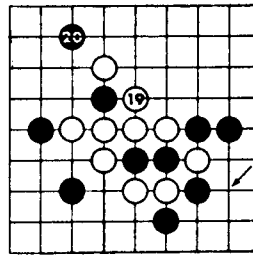
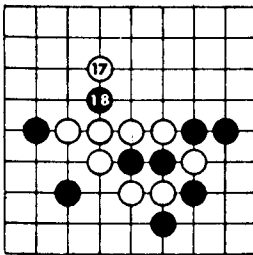
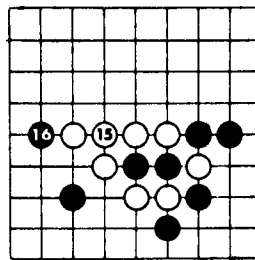
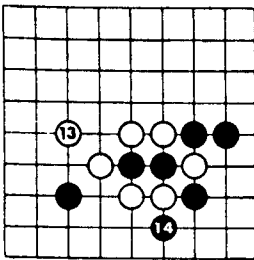
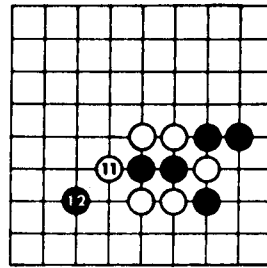
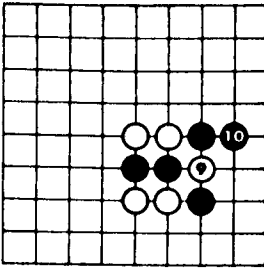
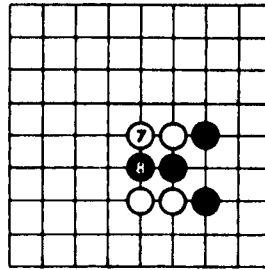
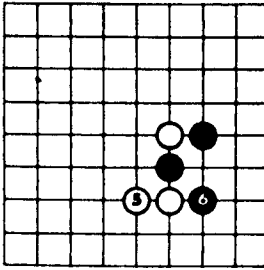


FIG. 2 (cont.)

The opening moves are characterised by the fact that it is usually dangerous to play too far away from one's opponent. White's moves 3, 5, 7 and 9 are motivated by a desire to kill his opponent's tentative line patterns whilst still preserving his own. The move 9, in fact, achieves this object. At this

point white has active open patterns along the two diagonal lines through 9, whilst black has no active open patterns.

Black now plays 10 to kill one of White's open patterns—a weak defensive move.

White plays 11 to create a very strong (as we shall see) group of forcing moves around the left margin of the played pieces.

Black (unwisely) ignores the threat and plays the (weak) development move 12.

White can now win by a sequence of forcing moves: 13 (making up a pattern of type C) forcing 14; 15 (type B) forcing 16; 17 (type CC) forcing 18; 19 (making up a winning pattern of type A) forcing 20, etc.

Informal description of the learning mechanism

We shall start by considering just what is involved in writing a program to play Go-Moku. Most of what follows is applicable to two-person board games generally, and many of the basic ideas are not new. Indeed, our debt to previous workers (particularly Shannon, Samuels, Newell, Simon and Shaw) is so pervasive that no attempt will be made to give detailed acknowledgements. As far as we know, however, the particular learning mechanism isolated and investigated in the present program, the method we have called 'backtrack analysis', is new.

At any stage of the game there is a large number of possible moves (all vacant lattice positions). The first problem is to describe the current board situation in such a way as to make possible:

- (i) adequate discrimination between possible moves;
- (ii) an ordering (evaluation) of discriminated moves.

A further general requirement on the description (and indeed *all* processing) is that it should be sensible in the demands its implementation makes on computer time. This (debatable) requirement was not made for financial (or social) reasons, but was one which we felt made sense in relation to the learning process as a whole: if the learning mechanism was not powerful enough for the learning program to meet this requirement, then it had no place in the immediate project.

Let us suppose that, as a result of our own experience of playing Go-Moku, we have formed certain ideas about the kinds of description which might be appropriate.

If we use a description which is sufficiently simple for the ranking of discriminated moves to be obvious, then it is a straightforward business to write a program to play on the basis of this ranking. We have, in fact, implemented such non-learning programs to play Go-Moku. The strength of a playing program, however, resides largely in the power of the descriptive machinery. If more powerful descriptions are used, allowing more subtle discriminations between possible moves, it becomes increasingly difficult and

sometimes quite impracticable to rank the discriminated moves by direct inspection, and consequently impossible to make use of the description at all in a non-learning program.

At this point it becomes imperative, if one is to progress, to extend the concept of the game-playing program to include an automatic generation of an ordering of discriminated moves. An immediately attractive way of doing this is to get the program to analyse its played games and obtain the required ordering as a result of its accumulated experience of the analyses. There are various ways in which the descriptive and analytic components of the program might be designed to make this possible. One outstandingly successful method is that used by A. L. Samuel in his now classic draughts-playing program. The method isolated and studied in the present work is a deductive backtrack analysis of the kind a human game-player might make. Informally described, it goes as follows:

A game between the learning program (LP) and an opponent is played to a win. In a sense the LP now asks itself: 'Where did I (or my opponent) go wrong?' The LP performs a backtracking *analysis* to find the critical board situation from which it sees the win as inevitable. The LP then attempts to describe the particular *abstracted* board situation in its formal descriptive language. If the formal language has been suitably designed, then this description will automatically *generalise* the abstracted board situation. Finally the ordering of discriminated moves is adjusted in the light of this analysis, abstraction and generalisation.

It should be emphasised that the component of the LP which generates the ordering is *not* tied to any one formal description of the current state of the board, except in the minor demands it makes on the way this description is presented during implementation.

If the primary task was simply to design a game-playing program, then the LP not only makes it possible to use a powerful descriptive language, but gives one the facility for experimenting with different descriptive languages in a search for the most powerful. It is hoped that these features of the present work will carry over into less trivial tasks, and that the deductive backtrack analysis will prove a useful component in other learning programs.

THE LEARNING PROGRAM

As indicated in 'Informal description of the learning mechanism' (p. 90), the aim is to isolate and study the backtrack analysis component (BAC) as a learning mechanism for generating an ordering of discriminated moves.

The BAC operates in the context of a formalism for describing possible moves in such a way as to allow a description of a particular move to be elaborated through a sequence of increasingly sophisticated levels of discrimination. This formalism will be defined and discussed in 'Description of possible moves' (below). 'Program play' (p. 94) discusses, briefly, how in a game played by the LP a move is selected using an ordered list of

described moves. 'Backtrack analysis' (p. 94) discusses in detail how this ordering is generated by the BAC.

Description of possible moves

A description of a class of possible moves is made up of a number of component sub-descriptions, referring to particular features. It is supposed that, for any one feature, the possible values of the feature can be ordered by a relation 'better than' ('contains'). For example, consider the (imprecisely) defined feature: 'the maximum number of played pieces of the same colour in a possible winning 5-pattern which includes the move'. The possible values of this feature are 1, 2, 3 or 4. The relation 'better than' might be taken as integer magnitude.

An ordering is now defined over the values of a description. If D' and D'' are two values of a description, the relation 'better than' holds between D' and D'' if and only if it holds between each component feature.

More formally: An n -description, D^n , of a class of possible moves, is an ordered n -tuple of components d_i , $1 \leq i \leq n$. For each i , $1 \leq i \leq n$, there is an ordering (written $\supset^i$, and loosely thought of as 'better than' or 'contains') defined over the set of values of d_i .

An ordering (written $\supset$ and again thought of as 'is better than') is defined between two descriptions D_α^m and D_β^n if and only if $m=n$, when

$$D_\alpha^n \supset D_\beta^n$$

if and only if

$$d_{\alpha 1} \supset^1 d_{\beta 1}, d_{\alpha 2} \supset^2 d_{\beta 2}, \dots, d_{\alpha n} \supset^n d_{\beta n}.$$

It should be noted that not $(D_\alpha^n \supset D_\beta^n)$ does not imply $D_\beta^n \supset D_\alpha^n$.

Both in the interests of giving flesh to this skeleton, and of providing background in terms of which the actual performance of a particular LP can be discussed later, an example sequence of actual descriptions used in an LP follows.

An actual description

Begin by defining:

A 7-pattern is the pattern of played pieces on a line of seven adjacent lattice points.

A black 7-pattern is an integer, N , where N has the value 'if an end point is black or an interior point is white then zero, else one plus number of black pieces' (i.e., zero if the interior 5 cannot be made into a winning black pattern else a measure of completion of the pattern).

A line pattern on a given line through a given vacant lattice point is the pattern of played pieces on the given line through the lattice point.

A black line pattern is an ordered pair of integers, $n:r$, where n has the value of maximum $[N]$ over the set of black 7-patterns for which the given vacant lattice point is an interior point, and r is the number of successive

1-translations of a 7-pattern that can be made keeping $N=N_{\max}$ (see examples).

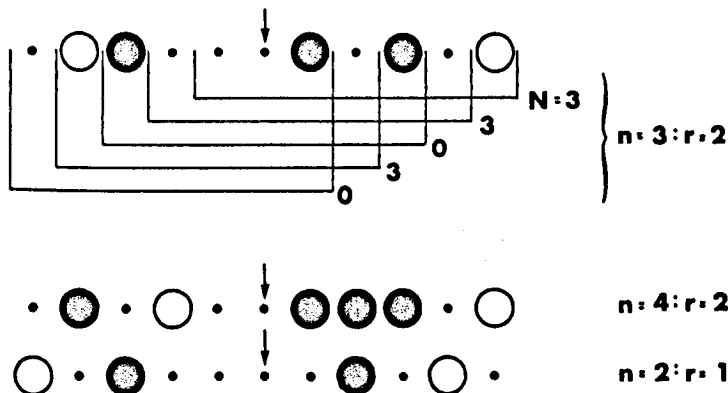


FIG. 3.

Fig. 3 shows some examples of line patterns: r gives a measure of the number of different winning patterns to which a piece played on the vacant lattice point, together with a largest group of already played pieces on the same line, can contribute.

If an ordering is defined over the set of ordered pairs n, r then a black line pattern is a description D^1 . We will take the ordering to be first by magnitude of the integer n and then by r . This description allows adequate discrimination of play on a given lattice point with regard to the given line. If and only if a move completes a winning pattern on the line then it has the description $5:1$. The description also gives a sensible ordering: a $4:2$ (a winning move) is 'better than' a $4:1$ (not a winning move); a $3:2$ is 'better than' a $2:3$.

There are four black line patterns through any vacant lattice point.

The level 1 description, D^1 , of a possible move is defined to be the best (see 'Description of possible moves' (p. 92)) black line pattern through the given vacant lattice point.

The level 1 description cannot give an adequate (sufficient) description of winning moves such as B,B C,C and B,C (Fig. 1) which depend on the pattern of played pieces on two lines intersecting in the considered lattice point. Such patterns can be described by extending the description of a possible move by another D^1 .

The level 2 description, D_2 , of a possible move is defined to be the second best line pattern through the given lattice point. The description $D_1 \times D_2$ (a D^2) is now capable of discriminating and ordering moves such as those of B,B, C,C and B,C (Fig. 1).

This kind of description can be and has been extended further but the above two levels provide an adequate background for the next two sections. More sophisticated levels of description will be given when needed.

Program Play

In a particular game, move selection is essentially governed by the LP's current list of subgoals. It is supposed that these subgoals are descriptions of moves from which a win is expected to be inevitable. The current list of subgoals has been generated in a way that will be discussed in detail in the next section. The list governs which of the described possible moves in the given game-state is finally selected for play. There is an implicit ordering of subgoals on the list according to the number of moves away from a win. This ordering is natural: it allows the most economical of a number of alternative winning moves to be selected. Equally important, it makes the choice between attacking and defensive moves straightforward; if black is to play and white has a higher level subgoal than black, then black must play (defensively) on the lattice point which gives white this higher subgoal.

Thus, the task is to select a move from a list of possible moves. Each move on the list is processed to find its description. The list of subgoals is now scanned by level and, for each subgoal, a comparison is made between the description of the subgoal and the described possible moves. The highest level acceptable attacking or necessary defensive move is selected.

Finally, there is a default move. If the search procedure does not find a move which achieves a subgoal then a default procedure is called to select a move. The default procedure may use quite different criteria or no criterion at all (e.g., selection using a pseudo-random number generator).

THE BACKTRACK ANALYSIS

A number of backtrack analysis components have been experimented with at various levels of complexity. One of the simpler, implemented originally as part of a feasibility study for the present LP, has formed a basis for later elaborations. We will begin with a discussion of this simple BAC and its limitations.

A simple BAC

This simple BAC works in the context of a fixed (static) description of possible moves. The flow is given in Fig. 4.

The BAC is only called when the LP has *lost* a game. This is important since it then follows that the loser's moves were played on the basis of the LP's current list of subgoals, and therefore need not be re-examined during the backtrack—they are the 'best' moves the LP could make. Removal of this restriction requires a more sophisticated BAC.

The BAC is entered with a list of played moves in reverse order, i.e., winning move at head of list. 'Level' refers to the ordering of subgoals on the list of subgoals by number of moves away from completion of the winning pattern. The level of the last subgoal played in the BA is held in 'old level'. When, in the course of the BA, a move which is not a subgoal

is encountered, the description is added to the list of subgoals at a level one below 'old level'—it being assumed that the new subgoal is a necessary and sufficient step in the formation of the higher one.

In general, this last assumption is clearly not justifiable. It is this requirement, that the description added to the subgoal list should be necessary and

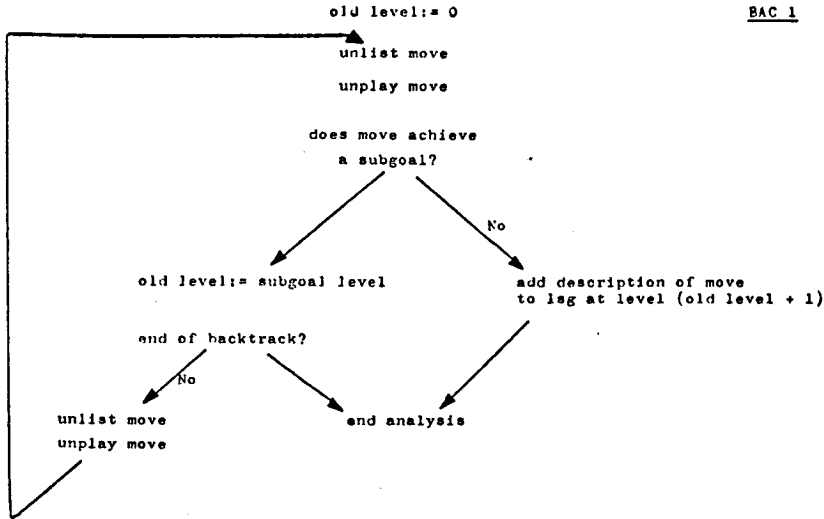


FIG. 4.

sufficient to ensure a transition to a higher subgoal, that lies at the heart of the design of a BAC. We will approach the problem through a sequence of elaborations of BAC 1; each elaboration going some way further to meet the requirement.

Some extracts from an actual teaching sequence with an LP using BAC 1 and 'random' play as a default move are given in the Appendix. These extracts are given both for their intrinsic interest and to provide a base-line from which to consider results obtained with later BACs.

BAC 2 and BAC 3: minimal descriptions

These BACs still operate in the context of a fixed description of possible moves. They go some way towards meeting the requirement that the descriptions of particular subgoals on the list of subgoals should be *necessary* (minimal).

Consider a played game in which a particular move achieves a new subgoal for which a *sufficient* description can be given. If D_{min} is the necessary (minimal) description of the subgoal, and D_{move} is the description of the move, then in general we have $D_{move} \supset D_{min}$ (D_{move} 'contains' D_{min}) but *not* $D_{move} = D_{min}$. We would like to have D_{min} added to the list of subgoals. BAC 1 adds D_{move} .

If D^1 , D^2 are descriptions such that $D^1 \supset D^2 \supset D_{\min}$ and $D^1 \neq D^2 \neq D_{\min}$ then, with BAC 1, if the sequence of teaching games is such that D^1 is encountered before D^2 , both D^1 and D^2 will be added to the list of subgoals. Both D^1 and D^2 describe the same subgoal and neither is a minimal description. Experiments with BAC 1 showed that this is what usually happens. Without attempting to give a precise explanation: it is a consequence of the fact that games in a teaching sequence tend to become less and less 'open' as the program learns.

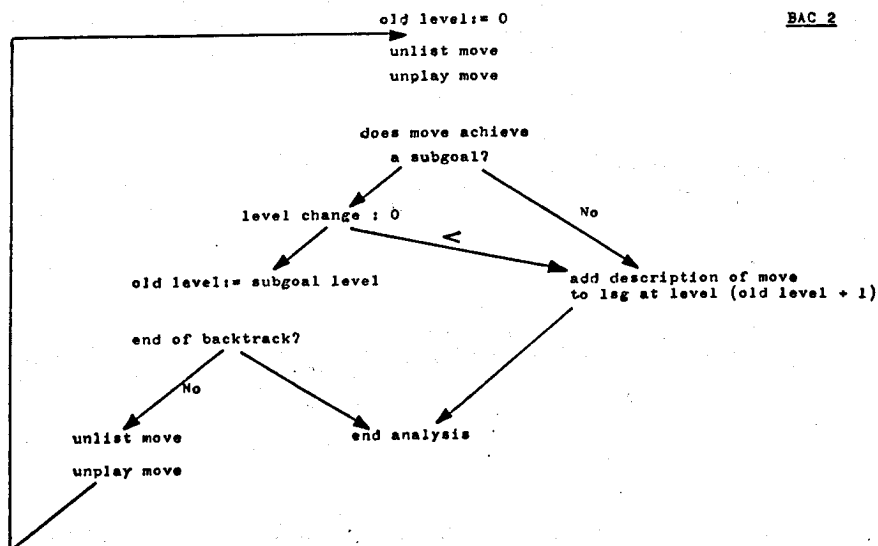


FIG. 5.

BAC 2 finds *minimal* descriptions by a straightforward modification of the action of the component, 'Add description of move to list of subgoals at level (old level + 1)', of BAC 1. Suppose the description D_{new} is to be added at a given level to the list of subgoals. D_{new} is added and, as part of the re-ordering of the list of subgoals, all those descriptions at the given level which 'contain' D_{new} are deleted. By this simple device, and over a sequence of teaching games, the descriptions on the list of subgoals approach minimal descriptions.

The experiments with BAC 1 also showed that, when subgoals are insufficiently described, their partial descriptions are often discovered for the first time in a BA at a level lower than their 'true' level. In BAC 2 if, on backtracking, the level change (old level—level + 1) between two successive subgoals in the backtrack sequence is found to be negative, then the list of subgoals is amended to bring the lower subgoal up to a level one below the first.

Finally, it is interesting to try to obtain more information from each BA and not simply end the analysis at the first re-organisation of the list of subgoals.

Consider the BA when the white move (n) is discovered as achieving a new subgoal. The list of subgoals is modified. Would the LP move ($n-1$) still make sense if LP had been playing from the new list of subgoals? Yes: if and only if $D_{LP}(n-1)$ 'contains' $D_W(n)$. Now this is most unlikely to happen for an LP whose default move selection is random. If, however, the default move selection is 'Play the best D^2 (own if equal black and white alternatives) on the board', then this situation can and frequently does occur. The modification of BAC 2 which makes this extended analysis possible is called BAC 3 and its flow is given in Fig. 6.

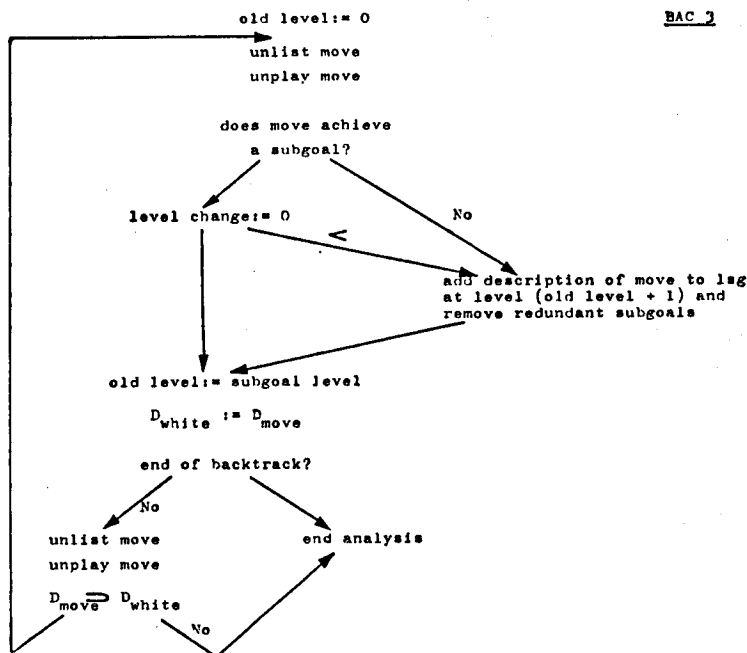


FIG. 6.

The selection of results given and discussed in the Appendix are from a teaching sequence with an LP using BAC 3 and a default move of the kind indicated above. It should be emphasised, however, that the final list of subgoals obtained in a teaching sequence is not significantly dependent on the default move selection.

BAC inadequate descriptions

Learning programs based on BAC 1, 2 or 3 all reach a stage at which they can learn nothing more from backtrack analyses of lost games. All minimal descriptions of subgoals which can be adequately described have been obtained. The opponent wins (*when he wins!*) by a move which achieved a described, but *inadequately* described, subgoal on the list of subgoals: the backtrack analysis cannot even recognise that there is a critical board situation, let alone resolve it.

The work going on at present is aimed at extending the BAC so that in a BA:

- (i) inadequate descriptions of subgoals are recognised as such and,
- (ii) following such recognition, it is optional to extend the formal descriptive language, in the light of the BA, in an attempt to make an adequate description of the new subgoal possible.

Recognition of inadequate descriptions

There are a number of ways by which the LP might detect whether or not a description is inadequate. We will indicate briefly two quite different approaches.

(i) When the LP is playing a game, it keeps a record of any of its own attacking moves played from its current list of subgoals (lsg). The BAC performs a similar analysis to BAC 3 but, in addition, now:

If the LP has lost, then the BAC looks at the record of own-moves which achieved an attacking subgoal. These subgoals were supposed to be winning subgoals. Their presence on the record indicates that the descriptions must be inadequate.

If the LP has won, then the BAC simply backtracks through the LP's moves, monitoring the sequence of subgoal levels. If a subgoal is encountered at a higher level than the previous subgoal in the backtrack, then this last subgoal is inadequately described, in that achieving it has not inevitably created a subgoal at a higher level.

(ii) Here the idea is that when, on backtracking, the BAC recognises a new subgoal, the new subgoal is subjected to an *immediate*, though not infallible, test for inadequacy. The test consists in generating a number of 'minimal' (in the sense of number of played pieces) examples of board situations which answer the description of the new subgoal. Starting from an example board situation, the BAC now plays alternate own and 'opponent's' moves, using the LP's current lsg to select both, to see if a win can be achieved in the number of moves implied by the level at which the new subgoal has been discovered. Depending on the results of this test for a number of 'randomly' generated examples, the new subgoal can be added to the lsg with the appropriate indication 'adequate' or 'inadequate'.

Method (i) has the advantage that its incorporation in the BAC is essentially independent of the particular descriptive language of the LP. It has the disadvantage that recognition of inadequacy may possibly not be obtained in the same game-analysis in which the new subgoal is discovered and added to the lsg. Indeed, recognition may be deferred several games, with undesirable implications for play and analysis in the intervening games.

With method (ii) the recognition of inadequacy can often be made immediately a new subgoal is discovered. On the other hand, a direct interpretation of method (ii) implies that each time the descriptive language is changed or extended, the processing unit in the BAC which generates 'random' examples has also to be changed or extended.

Both methods have been implemented and tested in the context of the simple descriptive language used for illustration in this paper. Both are being studied further in the context of more sophisticated descriptions (cf. 'Extension of inadequate descriptions' below).

A variant of method (ii) which overcomes the disadvantage of *direct* generation of 'examples' is also being investigated. A single minimal example is generated from the *particular* critical board situation met in the BA. Starting from this critical board situation, a piece 'x' on the board is 'unplayed'. If the description of the move which achieves the new subgoal is unchanged as a result of unplaying 'x' then 'x' is left off the board; else 'x' is replayed. This is done for all the pieces of the critical board situation. The pieces remaining on the board represent a minimal example of the described new subgoal. The example is now played to completion by the BAC as before. This variant of (ii), like (i), has the advantage that its incorporation in the BAC is independent of the particular descriptive language of the LP.

Extension of inadequate descriptions

BAC 1, 2 and 3 operate in the context of a *fixed* description of possible moves. With a BAC which is capable of recognising when a description is inadequate, it is desirable to be able to operate in the context of an *extendable* description. Thus, when the BAC indicates that a description of a new subgoal is inadequate, it should be possible, either immediately or at a convenient later 'break point', to add a new processing unit to the descriptive component; the BAC operates in what is essentially a conversational mode.

With a description scheme of this kind, it is interesting that the lsg can be used, not only in selecting a move from a list of promising moves, but also to control the amount of processing that goes into the actual description of particular promising moves. Briefly, in selecting a move from the list of possible moves, the moves on the list might first be described at the simplest level of description. During the comparison of these described moves with the descriptions on the lsg, contracted lists of 'promising' moves (moves whose simple descriptions contain the relevant component of an extended description of a subgoal) might be formed. These lists would be processed to the appropriate next level of description (which might be different for different lists) if and only if no subgoal is encountered at the simpler level of description. In this way the costly processing associated with obtaining sophisticated descriptions can be automatically controlled through the lsg. This is clearly feasible when the ordering of the lsg by (i) number of moves away from a win, and (ii) complexity of description, go hand and hand. The necessary control when this criterion is not satisfied is more difficult.

Work is in progress on a BAC which incorporates the components discussed in 'Recognition of inadequate descriptions' (p. 98) and which will be used to investigate the problems of operating in this kind of 'extendable description' mode.

ACKNOWLEDGEMENT

The work of the project has been supported in part by a grant from the SRC and the authors would like to take this opportunity of thanking the SRC for this assistance.

APPENDIX: EXTRACTS FROM RESULTS BAC 1 and 3

Note: the symbol * marks moves selected through use of the list of subgoals.

GO-MOKU 24 : 8 : 65 (BAC 1): Series 1

Newgame 1: current list of subgoals is: 5 : 0, 0 : 0

white begins: sequence of moves is:

8, 8 (8, 7) 9, 8 (6, 8) 10, 8 (7, 6) 11, 8 (12, 8*) 7, 8 wins

Game analysis is:

move 7, 8 achieves subgoal 5 : 0, 0 : 0 at level 1: backtrack

move 11, 8 does not achieve subgoal: description 4 : 1, 1 : 3 added to list of subgoals at level 2

Newgame 2: current list of subgoals is: 5 : 0, 0 : 0

4 : 1, 1 : 3

white begins: sequence of moves is:

8, 8 (7, 7) 9, 9 (9, 6) 10, 10 (6, 7) 11, 11 (12, 12*) 10, 9 (8, 7) 9, 7 (7, 6) 7, 9 (8, 9*)
6, 10 (5, 11*) 10, 6 wins

Game analysis is as follows:

move 10, 6 achieves subgoal 5 : 0, 0 : 0 at level 1: backtrack

move 6, 10 achieves subgoal 4 : 1, 1 : 3 at level 2: backtrack

move 7, 9 does not achieve subgoal: description 3 : 2, 3 : 1 added to list of subgoals at level 3

new list of subgoals is: 5 : 0, 0 : 0

4 : 1, 1 : 3

3 : 2, 3 : 1

.

Newgame 8: current list of subgoals is: 5 : 0, 0 : 0

4 : 1, 1 : 2 4 : 1, 1 : 3

3 : 2, 3 : 1

4 : 0, 2 : 1 3 : 2, 2 : 3

3 : 2, 2 : 2

2 : 3, 2 : 3

white begins: sequence of moves is:

8, 8 (8, 7) 9, 9 (9, 8*) 10, 10 (7, 7*) 11, 11 (12, 12*) 10, 9 (10, 8*) 10, 11 (10, 12*)
11, 9 (8, 9*) 9, 11 (8, 11*) 8, 12 (7, 13*) 12, 8 wins

Game analysis is:

.

Comments

Game 1. The LP plays 'at random' until it recognises (too late!) the winning move 12, 8. The LP plays 12, 8 but white wins with 7, 8. During the BA the LP recognises the move 11, 8 as creating a subgoal (the winning pattern of Fig. 1), and adds the description (4 : 1, 1 : 3) of this move to its list of subgoals.

Game 2. This game is longer than necessary. The sequence of moves up to the LP's move (12, 12*) are simply to illustrate that the simple win of game 1 cannot be repeated.

Game 8. The LP now has an extensive list of subgoals. White however wins by playing 11, 9 at move 13. After white plays 10, 11 at move 11, moves 11, 9 and 9, 11 are winning moves for white. The LP plays 10, 12. The reason is that all three moves achieve (inadequately) described subgoals at the same level (4) on the list of subgoals (lsg). The descriptive language would need to be extended before the BA could resolve such board situations and learn anything significant from this game. Note however:

- (i) how the program play is now almost entirely determined by the lsg (move selection from the lsg is indicated by the output of * after the move);
- (ii) the existence of non-minimal subgoals on the lsg. Thus, at level 2, 4 : 1, 1 : 3 4 : 1, 1 : 2 and neither is the minimal subgoal for this level (compare with results of BAC 3).

GO-MOKU 2 : 9 : 65 (BAC 3): Series 2

Newgame 2: current list of subgoals is: 5 : 0, 0 : 0
4 : 1, 2 : 0

white begins: sequence of moves is:

7, 7 (8, 7) 8, 6 (6, 8) 8, 8 (6, 6) 9, 9 (10, 10) 9, 5 (10, 4) 9, 7 (9, 6) 10, 8 (11, 9*)
10, 6 (11, 5*) 11, 7 (12, 6*) 12, 8 (8, 4*) 13, 9 wins

Game analysis is:

move 13, 9 achieves subgoal 5 : 0, 0 : 0 at level 1: backtrack

move 12, 8 achieves subgoal 4 : 1, 2 : 0 at level 2: backtrack

move 11, 7 does not achieve subgoal: description 3 : 2, 3 : 2 added to list of subgoals
at level 3: backtrack

move 10, 6 does not achieve subgoal: description 3 : 2, 2 : 3 added to list of subgoals
at level 4

new list of subgoals is: 5 : 0, 0 : 0
4 : 1, 2 : 0
3 : 2, 3 : 2
3 : 2, 2 : 3

.

Newgame 5: current list of subgoals is: 5 : 0, 0 : 0
4 : 1, 2 : 0
3 : 2, 3 : 2
3 : 2, 2 : 3
2 : 3, 1 : 3
1 : 3, 1 : 3
2 : 3, 1 : 2

white begins: sequence of moves is:

7, 7 (8, 7*) 8, 6 (7, 6) 6, 8 (7, 8) 5, 9 (9, 5*) 4, 10 wins

Game analysis is:

move 4, 10 achieves subgoal 5 : 0, 0 : 0 at level 1: backtrack

move 5, 9 achieves subgoal 2 : 3, 1 : 3 at level 5

description 4 : 1, 1 : 3 added to list of subgoals at level 2

new list of subgoals is: 5 : 0, 0 : 0

4 : 1, 1 : 3

3 : 2, 3 : 2

3 : 2, 2 : 3

2 : 3, 1 : 3

1 : 3, 1 : 3

2 : 3, 1 : 2

.

Newgame 10: current list of subgoals is: 5 : 0, 0 : 0

4 : 1, 1 : 3

4 : 0, 3 : 1 3 : 2, 3 : 2

3 : 2, 2 : 0

2 : 3, 1 : 3

1 : 3, 1 : 3

2 : 3, 1 : 2

white begins: sequence of moves is:

7, 7 (8, 7*) 8, 8 (7, 8) 9, 9 (6, 6*) 10, 10 (11, 11*) 10, 8 (9, 6) 6, 9 (10, 9*) 9, 8 (12, 8*)

11, 9 (12, 10*) 8, 10 (11, 7*) 9, 10 (9, 11*) 7, 10 (11, 10*) 6, 10 wins

Game analysis is:

move 6, 10 achieves subgoal 5 : 0, 0 : 0 at level 1: backtrack

move 7, 10 achieves subgoal 4 : 1, 1 : 3 at level 2: backtrack

move 9, 10 achieves subgoal 2 : 3, 1 : 3 at level 5

description 3 : 1, 3 : 1 added to list of subgoals at level 3

new list of subgoals is: 5 : 0, 0 : 0

4 : 1, 1 : 3

3 : 1, 3 : 1

3 : 2, 2 : 0

2 : 3, 1 : 3

1 : 3, 1 : 3

2 : 3, 1 : 2

.

Comments

Game 2. Illustrates the continuation of the BA after the first new subgoal found. The condition $D_{\text{move}} \supset D_{\text{white}}$ (see Fig. 6) is satisfied for move 11, 5 but not for move 11, 9: the BA therefore stops at move 11, 9.

Game 5. This game is dramatically lost by the LP because it did not recognise a particular example of the winning pattern of Fig. 1. The minimal description of a move achieving this subgoal is 4 : 1, 0 : 0. A non-minimal description learnt in Game 1 is 4 : 1, 2 : 0, and this non-minimal description is on the lsg at the start of Game 5. The move 5, 9 of Game 5, which actually achieves the winning pattern A of Fig. 1, has the description 4 : 1, 1 : 3 which 'contains' the minimal description (4 : 1, 1 : 3 $\supset$ 4 : 1, 0 : 0) but does *not* 'contain' the description on the lsg at level 2 (4 : 1, 1 : 3 $\not\supset$ 4 : 1, 2 : 0). In fact (see analysis record) the move 5, 9 is only recognised as achieving a subgoal at level 5 (4 : 1, 1 : 3 $\supset$ 2 : 3, 1 : 3). The LP has a 'better' move 7, 8, which achieves a higher level attacking subgoal and this is selected for play. The LP learns its lesson however that the BA recognises

that the new subgoal $4:1, 1:3$ should be added at level 2 *not* level 5. In adding the new subgoal at level 2 it notes that the previous subgoal at this level 'contains' the new subgoal, and so is redundant and removed from the lsg. The lsg therefore now has a more minimal description of the winning pattern C of Fig. 1.

Game 10. A similar but more striking example of the convergence to minimal descriptions. White's move 9, 10 achieves a winning pattern of the kind of Fig. 1. The minimal description of this pattern is $3:1, 3:1$. The description on the lsg at this stage is the non-minimal description $3:2, 3:2$. It follows that, during play, the LP does not recognise move 9, 10 as achieving a subgoal at level 3. It is worth noting that there is another description $(4:0, 3:1)$ on the lsg at level 3. This is a (minimal) description of a winning pattern of the kind CC Fig. 1. Neither description at level 3 is contained within the other. When, as a result of the BA of this game, the description $3:1, 3:1$ is added to the lsg at level 3, the non-minimal description $3:2, 3:2$ is deleted. In addition the minimal description $4:0, 3:1$ is also recognised as being now redundant ($4:0, 3:1 \supset 3:1, 3:1$) and level 3 contracted to a single description which allows all winning patterns of the kind CC Fig. 1 to be detected.

7

AN APPROACH TO AUTOMATIC PROBLEM-SOLVING

JAMES DORAN

EXPERIMENTAL PROGRAMMING UNIT
UNIVERSITY OF EDINBURGH

Summary

A digital computer program, the Graph Traverser (Doran & Michie 1966), can seek a solution to any problem which may be interpreted as that of finding a path from one specified node of a graph to another. Emphasis is placed upon the *evaluation* of intermediate states of the problem (nodes of the graph) according to the extent to which they resemble the 'goal' state.

Sample results from first applications of the program, and possible future developments, are discussed. The program is related to other problem-solving programs.

INTRODUCTION: PROBLEMS AND PROBLEM-SOLVING PROGRAMS

How to travel from London to Birmingham may, in some circumstances, be a 'problem'. So may be a crossword puzzle, the planning of a school timetable, the solving of an algebraic equation, and the curing of a sick person. I shall not attempt to *define* a 'problem', except to suggest that the solver is typically faced with an essentially passive 'situation' or 'system' which must be 'manipulated' until it satisfies certain requirements.

This paper is primarily concerned with digital computer programs to solve problems. It describes one such program in detail. This is the Graph Traverser (Doran & Michie 1966), which is being developed at the University of Edinburgh Experimental Programming Unit.

Problem-solving programs

A problem-solving program must solve, or help solve, problems. It must

be applicable to a range of problems, narrow or broad, but need not necessarily be useful for all the problems within that range.

Even when a problem is precisely stated, and when a method of solution known to be infallible is available, it does not follow that this provides a useful method of attack in practice. A solution must be obtained within certain limits of time, and the machine will have a finite 'memory'. Methods have to be found which are useful within these restrictions. Typically, they will be 'probably useful' rather than 'infallible'. Such methods are commonly called 'heuristic'.

Before discussing in detail an example of a simple heuristic problem-solving program, the Graph Traverser, I shall give an indication of the most important work carried out in this field, stressing the more fundamental ideas. For a more extensive, and admirably clear, review of such work the reader is referred to Newell & Ernst (1965).

The GPS (General Problem Solver) program of Newell, Shaw and Simon (1960) is well known. It is designed to permit its application to a range of problems such as proving trigonometric identities, proving theorems in the propositional calculus and compiling computer programs. The GPS attempts to solve a problem by constructing a hierarchy of 'goals', where a goal involves one of the following:

- (1) transforming one object into another;
- (2) reducing the difference between one object and another;
- (3) applying a transformation (operator) to an object.

Given a goal, the program seeks either to attain it or to reduce it to a set of subsidiary goals (subgoals), each being of the type which the GPS can handle. Then each subgoal is similarly treated. Thus *subgoals* and *recursion* are major features of the program.

A descendant of the GPS is 'Gaku', a sophisticated problem-solving system with a number of new features (Hormann 1965). Gaku attempts to *plan*, that is, to use past experience to 'outline' a solution to a problem, and also to *induce* from the solutions of particular problems a general strategy which will suffice to solve any problem of a similar type. The system, which is still under development, has been little tested as yet. Ultimately it is intended that it should be capable of learning not only from direct experience, but also from a 'tutor' or other 'secondary' source of information.

Another program, SAINT (Symbolic Automatic INTEGRator (Slagle 1963)), which is influenced by the GPS, has a more restricted field of application, namely symbolic integration problems. For example, if SAINT is asked to integrate

$$\int \frac{x^4}{(1-x^2)^5} dx$$

it will, like a human solver, try various lines of attack, and various substitutions, before ultimately finding a sequence of transformations leading to an integral of standard type.

The organisation of SAINT has a number of similarities to that of the Graph Traverser program. It is a characteristic of SAINT, however, but not of the Graph Traverser, that it pays little attention to the problem of *which* line of attack to develop next, reserving its effort for *how* that attack is to be developed.

Another important and famous program is the checkers (draughts)-playing program of Samuel (1959). This is a 'game-playing' rather than 'problem-solving' program, and, unlike the programs so far described, is designed primarily to perform well, with no attempt to simulate human behaviour. Its performance has been, in fact, remarkably good. When deciding what move to play, it follows out a large number of alternatives, including, of course, the opponent's moves, and then comes to a decision based upon estimates of the values of the board positions found at the limit of its 'lookahead'. In order to attach values to each of these board positions, the program uses an 'evaluation function' based upon various features of the position. Furthermore, different versions of the program are capable of improving their play in two ways:

- (1) by keeping for reference a catalogue of the board positions so far encountered, together with certain information about each; and
- (2) by modifying the parameters of the evaluation function in the light of the discrepancies between the estimated value of the current board position, and the values of the positions discovered during 'lookahead'.

An important class of problem-solving program is characterised by the use of a subset of natural English for communication between the user and the program. The emphasis is on ease of communication, rather than problem-solving power, or even generality of application.

Raphael's (1964) SIR (Semantic Information Retriever) can 'understand' and manipulate simple statements such as 'X is part of Y' and 'Z is to the right of W'. Each of the variables may be replaced by any noun so that the program can 'understand':

'The boy is to the left of the table'

and can combine this with:

'The table is to the left of the chair'

to discover:

'The boy is to the left of the chair'.

Notice that all the program knows about the boy, the table and the chair is the given set of relationships, together with any others which it can *deduce*.

Bobrow (1964) describes 'a question-answering system for high school algebra word problems'. If the problem is posed:

'The price of a radio is \$69.70. If this price is 15% less than the marked price, find the marked price.'

The program will respond:

'The marked price is \$82.00.'

Here, as with SIR, the program can 'recognise' certain relationships (this time primarily algebraic) between entities, and work with them to solve the problem. Moderately large variations are permissible in the form of words used to pose the problem.

There are a number of different aspects to a problem-solving program. I shall distinguish four:

- (1) the external *application problem* to which the program has been applied;
- (2) the *internal problem* which the program generates from the application problem and which it must solve in order to produce a solution to the application problem (this internal problem will be an 'idealised' version of the application problem, and will typically vary little from one application to another. It is important to realise that more than one internal problem may be capable of derivation from a given application problem);
- (3) the *strategy* which the program uses to solve the internal problem;
- (4) the translation process from the application problem to the internal problem. (This involves an *application package* of information, large or small, which must be provided for each different application.)

The Graph Traverser program will now be described in these terms.

THE GRAPH TRAVERSER PROGRAM

Work on the Graph Traverser to date has concentrated primarily on one particular aspect of automatic problem-solving, the *evaluation function*, which has been examined in some detail in the context of simple puzzles. In a problem-solving context, an evaluation function may be defined as a means of measuring the 'promise' of a potential problem state, as part of the process of deciding which possible course of action to investigate next. The aim has been to develop methods by which the program may generate, or at least improve, its own evaluation function.

The program has, however, been written with sufficient generality to permit a much wider range of application and experimentation.

Description of the Graph Traverser program

The description of the program and its organisation will follow the 'internal problem', 'strategy', 'application package' framework given in the introduction.

In order to discuss the internal problem of the program it is convenient to use a little of the language of graph theory (see, for example, Berge 1962). A *graph* in the mathematical sense is a set of *nodes* any two of which may or may not be connected by an *arc* which may be directed or undirected. A graph is said to be *symmetric* if all its arcs are undirected.

A *path* from one node to another is a sequence of arcs, each either undirected or pointing 'forwards', which extends from the first to the second node. The *distance* from one node to another is the smallest number of

arcs which form a path joining them. A graph is *connected* if for any two nodes x, y there is a sequence of arcs of any type between them. A *tree* is a connected graph without any kind of 'loop', as in the examples of Fig. 2. The *descendants* of a node are all nodes which can be reached from it along a path.

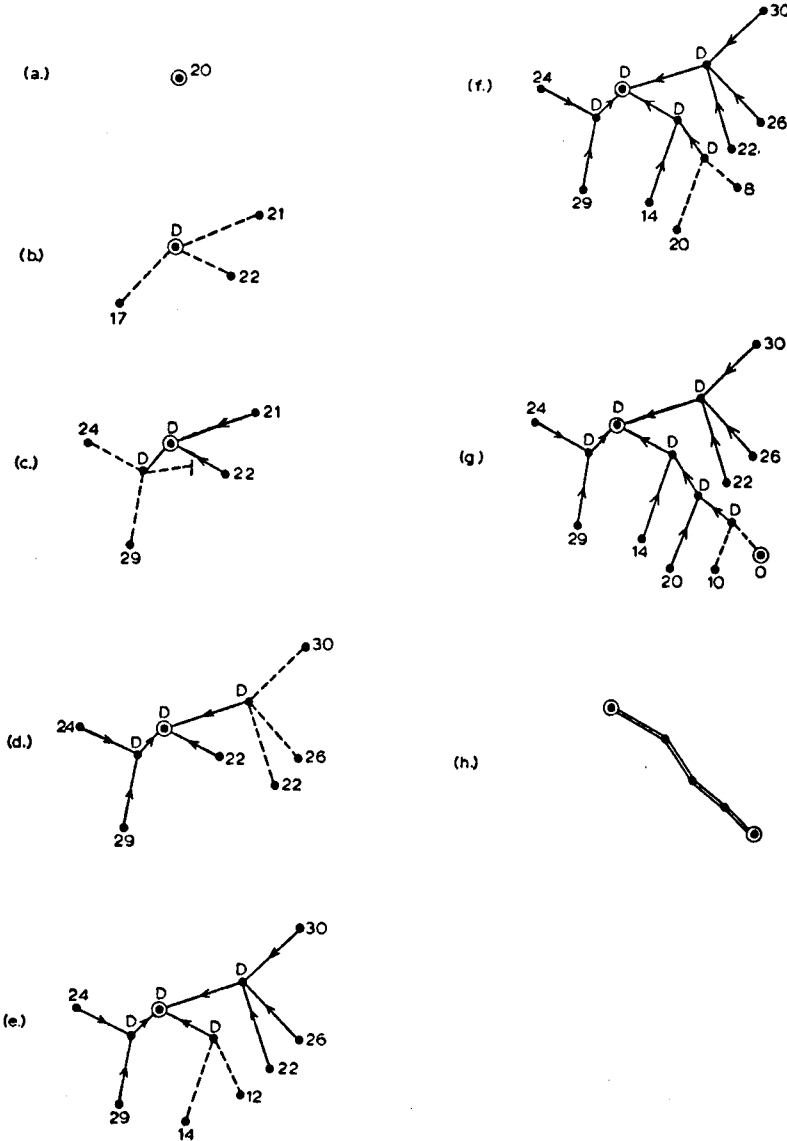


FIG. 1. The Graph Traverser in action. Successive diagrams show a search tree being enlarged by successive developments until the goal is located and a solution path determined. The start and goal nodes are ringed, the symbol 'D' indicates a developed node, and the figures are the values assigned to nodes by the program's (imperfect) evaluation function. (Reproduced from Doran & Michie 1966.)

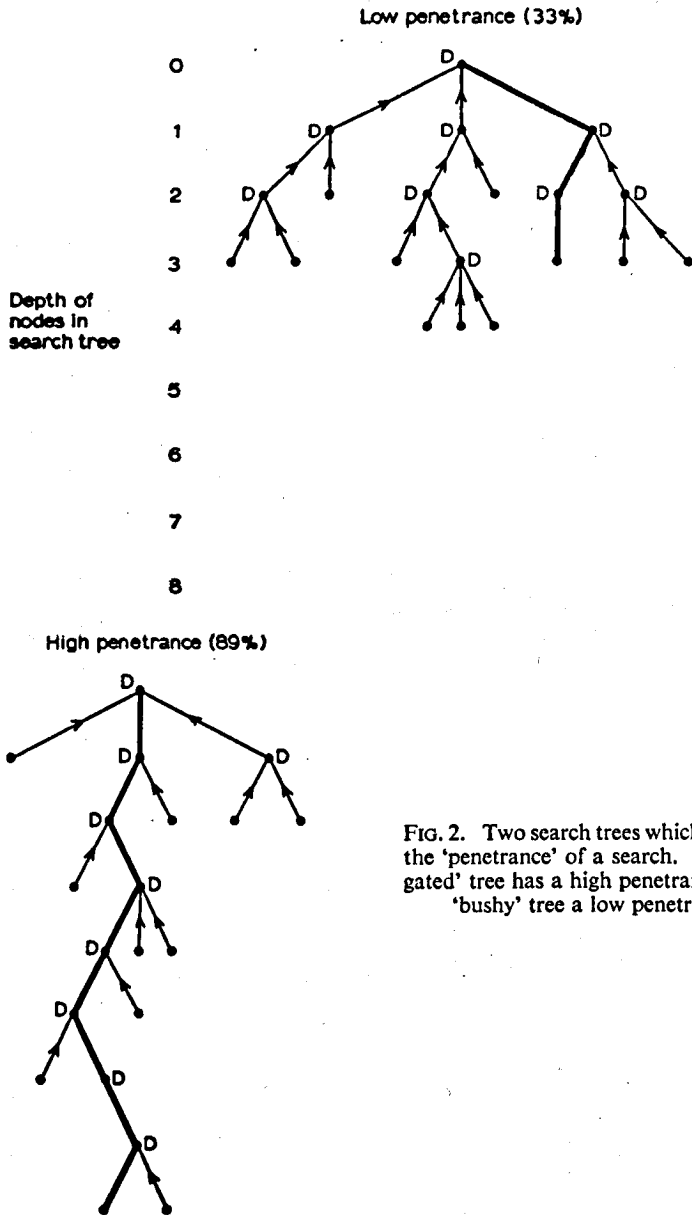


FIG. 2. Two search trees which illustrate the 'penetrance' of a search. An 'elongated' tree has a high penetrance, and a 'bushy' tree a low penetrance.

The internal problem attacked by the Graph Traverser may now be described. It is given a 'problem graph', with two of its nodes labelled *start* and *goal*, to find a path from the start to the goal. Minsky (1961) remarks, 'Almost any problem can be converted into a problem of finding a chain between two terminal expressions in some formal system.'

The graph is defined to the program as a set of distinct integer matrices,

each of which corresponds to a node, together with a procedure 'develop' which given a node lists all the immediate (i.e., one-arc) descendants of that node. The program seeks always to achieve maximum economy, that is, the minimum amount of searching, and is unconcerned as to the length of the path it finds from the start to the goal.

The program's strategy is founded upon the evaluation function. This is embodied in a procedure evaluate which when applied to a node attaches to it a 'value', hopefully correlated with the distance of the node from the goal.

The search proceeds iteratively. At the commencement of an iteration the program has stored the nodes it has so far discovered by repeated application of 'develop', together with the following information about each:

- (1) its value, as obtained by applying 'evaluate'; and
- (2) a pointer to the node from which it was developed.

The former is required for directing the search and the latter for constructing a path when the search terminates.

The iteration proceeds by finding the 'undeveloped' node with the smallest value (i.e., greatest 'promise') and using 'develop' to find all nodes adjacent to it. Every node *not previously located* is evaluated and stored. The program is thus constructing a 'search tree'. The iteration is now complete.

Initially the program holds one node, the start, and development continues until the goal is located. A procedure is then entered which constructs and prints out a path from the start to the goal. Should the number of nodes stored ('size' of the tree) reach a pre-set limit before the goal is located, the program will select the most promising undeveloped node, and print out a 'partial path' to that. A fresh search is then initiated using the selected node as the new start. The term 'partial search' will be used for the growth of one of a sequence of search trees. It follows from this that search could continue indefinitely. In practice, a 'resignation' criterion is included.

Fig. 1 shows the program at work on a problem graph. To understand what follows, it is essential that the reader understands this diagram. A fuller explanation has been published elsewhere (Doran & Michie 1966).

The crucial importance of the evaluation function in the search should be clear. If the function is constant, so that it conveys no information, then the program's strategy reduces to a systematic search of the nodes of the graph working from the start outwards.

The application package

The internal problem and strategy of the Graph Traverser have now been described, and there remains only to specify the application package.

To apply the program to any particular problem it is necessary:

- (1) to write input and output routines which convert the external problem-state into its internal integer matrix representation, and vice-versa;

MACHINE LEARNING AND HEURISTIC PROGRAMMING

- (2) to write a procedure 'develop' which, given a node, must list all immediate descendants of it so as to reflect the relationships between the external problem-states; and
- (3) to write a procedure 'evaluate' which must embody an evaluation function.

In addition various other quantities, such as the dimensions of the node-matrices, the maximum tree size permissible, and variable parameters of the evaluation function, will be set from the data tape.

The Graph Traverser: operational characteristics

The program is written in Elliott Algol. The language provides a high degree of flexibility, not, however, extending to list processing, a restriction which is more felt as the work advances.

The actual programming is, on the whole, naïve. No attempt has been made to produce a particularly efficient program, as it seemed more sensible to concentrate on what the program did, rather than how it did it. One exception to this rule is that, although a node has been stated to be an integer matrix, the program saves space by usually handling nodes in compressed form. Thus there are, in all, three levels of representation of a problem-state:

- (1) in whatever form it takes on the data tape;
- (2) as an integer matrix; and
- (3) in the compressed form as one or more integers.

When results obtained with the program are described, an indication will be given of the machine time required for typical operations. On the NCR Elliott 503 digital computer (using 8000-word core store only) search trees of between 500 and 1000 nodes are the largest which can be handled.

APPLICATION OF THE GRAPH TRAVERSER

An indication only of the program's performance in practice will be given here. For further details see Doran & Michie (1966).

Measures of performance

The three most interesting measures of the program's performance over a particular search of a graph are (1) the length of the path produced (P) (i.e., the number of arcs comprising the final path), (2) the total number of nodes developed (D), and (3) the total number of nodes added to the search tree (T). Since every node on the constructed path except the last must have been developed, it follows that $P \leq D$, and obviously $D \leq T$. Denote the minimal path length for a given start and goal by P^* . This will also be the minimum number of developments which must be made to find a path from the start to the goal. Then we define the *path efficiency* as P^*/P . Similarly, we define the *development efficiency* as P^*/D .

The calculation of both these efficiencies requires a knowledge of P^* . Typically, however, P^* will be unknown. The ratio P/D , which is the fraction of the total number of nodes developed which are incorporated into the actual path found, *can* always be calculated (even, by using the partial path, when the search has ended without the goal being located), and is of considerable importance. It will be called the 'penetrance' of a search, or partial search, and may be thought of as representing the degree to which the search tree is 'elongated' rather than 'bushy' (see Fig. 2).

The Eight-puzzle and the Fifteen-puzzle

The initial applications of the Graph Traverser have been to certain simply defined sliding block puzzles, which are particularly well adapted to the investigation of evaluation functions.

The Fifteen-puzzle is of this type and is often found as a toy. It consists of fifteen numbered square pieces in a 4×4 frame. One particular configuration of the pieces might be:

4	7	10	12
13	9	3	1
15	14	0	6
2	5	8	11

where the zero represents a space. The problem posed is to convert one given configuration into another by appropriate sliding movements of the pieces. A 'move' is defined as the movement of one piece into an adjacent square.

The Eight-puzzle is a reduced version of the Fifteen-puzzle, and has eight pieces in a 3×3 frame.

For the purposes of this work, arbitrary goal configurations have been defined for these two puzzles. These are:

1	2	3	4	1	2	3
5	6	7	8	8	0	4
9	10	11	12	7	6	5
13	14	15	0			

It should be clear that if nodes are identified with puzzle-configurations and arcs with moves, then these puzzles fit perfectly the schema of the Graph Traverser.

Note that for any particular goal configuration, only half the possible ways of setting up either puzzle lead to a soluble problem. All the configurations attempted by the program are chosen to be soluble. The combinatorial aspects of the Eight-puzzle are examined elsewhere in this volume by P. D. A. Schofield (p. 125).

In a study of human solving of the Eight-puzzle (see Hayes, Michie, Pole & Schofield 1965), it was found that groups of untrained subjects had averages of path efficiency of from 30 to 40 per cent. Practised subjects attain 80 per cent or even 90 per cent.

Tests of the Graph Traverser on the Eight-puzzle were run using an evaluation function involving a single variable parameter w . This was of the form $P + wS$ where P summed the distances of the pieces from their 'correct' location, and where S measured the extent to which the pieces were in their 'correct' sequence.

With the most favourable setting of w , the path efficiency was about 75 per cent, and the development efficiency was about 55 per cent. Furthermore, the penetrance was closely correlated with the development efficiency. This is important, for, as has been mentioned, the penetrance can be calculated where the efficiencies cannot, and is therefore potentially useful as a general measure of the efficiency of a search.

Table 1 refers to a test battery of ten randomly chosen Fifteen-puzzle configurations. Using the best evaluation function so far constructed, the Graph Traverser solves six out of these ten configurations within a maximum

TABLE 1

Results obtained by applying the Graph Traverser to ten randomly selected soluble Fifteen-puzzle configurations, using the best available evaluation function. The symbol '—' implies that the search was terminated, the size of the tree having reached 500 nodes. The penetrances of the unsuccessful searches are shown for purposes of comparison.

	Fifteen-puzzle configurations									
	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	F(8)	F(9)	F(10)
Path length	99	115	84	—	82	—	—	78	—	83
Nodes developed	165	182	99	—	126	—	—	114	—	111
Tree size	394	440	225	—	304	—	—	284	—	266
Penetrance, %	60.0	63.2	84.8	58.1	65.1	48.0	39.3	68.4	36.9	74.8

tree size of 500 nodes. This evaluation function involves several features of a Fifteen-puzzle configuration analogous to those Eight-puzzle features mentioned above.

The approximately constant ratio between the number of nodes developed and the tree size is a consequence of the homogeneity of the problem graph.

To solve a typical Eight-puzzle configuration on an Elliott 503 takes the program about fifteen seconds, and a Fifteen-puzzle configuration requires about four minutes.

Algebraic manipulation

Given a set $\{a, b, c, d, e, f, g, h, i, j\}$ upon which is defined a binary operation '*', we may write expressions of the form

$$(c * (((a * b) * d) * (e * f))).$$

If the operation '*' is commutative and associative, then we may apply these properties to transform one expression into another. For example, by

applying first commutativity and then associativity we may transform the expression

$$((a * b) * c)$$

first into

$$((b * a) * c)$$

and then on into

$$(b * (a * c))$$

The performance of the Graph Traverser when given a simple task of this type is shown in Table 2. Notice again the correlation between the development efficiency and the penetrance as the parameter varies. A very simple evaluation function was used which attached varying weights to the symbols of the expression, and gave as its value the sum of the moments of the weights about the left-hand end. In fact, the expressions can be transformed one into the other in just eight steps.

The most important difference between this simple algebraic manipulation problem and the sliding block puzzles is that the number of immediate descendants of a given node in the former case is much higher. This has the effect of reducing (severely, if the algebraic expression is large) the number of developments possible during the growth of a tree of a given size.

TABLE 2

Results obtained by applying the Graph Traverser to a simple algebraic manipulation problem. An evaluation function with one variable parameter was employed. The symbol '—' indicates failure to solve the problem within a search tree of 500 nodes.

(Reprinted from Doran & Michie 1966.)

Starting expression: $((h * (a * f)) * ((c * d) * c))$

'Goal' expression: $(((((a * c) * c) * d) * f) * h)$

Parameter setting for evaluation function

	5	7	8	9	10	11	12	13	14	15	20
Path count	—	—	13	9	9	10	10	10	—	—	—
Nodes developed	—	—	46	15	16	17	19	19	—	—	—
Tree size	—	—	268	112	117	124	132	132	—	—	—
Penetrance, %	24.4	11.5	28.3	60.0	56.3	58.8	52.6	52.6	18.2	13.6	10.2

FUTURE DEVELOPMENT OF THE GRAPH TRAVERSER

The remainder of this paper will be devoted to a discussion of some of the ideas and facilities which may become part of the Graph Traverser during further work. The material will range from work currently under way to speculation.

One important topic will *not* be considered except briefly at this point. The difficulty noted above, that in some circumstances a node will have inconveniently many immediate descendants, leads us to an important observation about the Graph Traverser. As so far described, it can recognise and manipulate the possible *states* of a problem, but not the transformations, or *operators*, which change those states. Thus it manipulates Eight-puzzle *configurations*, but not, in other than a trivial sense, the *moves* which connect

those configurations. Yet, commonly, an operator will be capable of definition quite independently of any particular problem state. Even in the Eight-puzzle we may define the operator 'north-move', say, which, where applicable, moves into the empty cell the piece immediately to its 'north'. Given any Eight-puzzle configuration it is clear whether 'north-move' is applicable, and, if so, what its action is.

Once manipulation of operators is possible, then the number of immediate descendants of a node may be reduced, effectively, by applying during the development of the node only a 'promising' subset of the operators actually applicable.

This topic is discussed in the article by Dr Donald Michie in this volume.

Range of application

The applications of the Graph Traverser to date have been within a restricted class of problem, for reasons already indicated. However, more attention is now being paid to the generality of the program, and, for example, the algebraic manipulation experiments represent merely a first attempt in this sphere (see also R. J. Popplestone's application of the program to a symbol-manipulation problem in this volume (p. 31)).

One of the weaker aspects of the Graph Traverser in its role as a general problem-solving program is the magnitude of its application package. To apply the program to a problem unlike any previously attacked may require a considerable programming effort. This situation cannot be improved in any simple way. The program must be equipped with a general problem specification such that the user may indicate his own particular problem with the minimum of effort. This is not easy. The work of Raphael (1964) and Bobrow (1964) already mentioned is relevant at this point.

Considerable extensions to the range of applicability of the program come from the realisation that its internal problem can be significantly altered with very little modification to the program itself.

Firstly, we may easily remove the requirement that the goal node be fully specified, and replace this by a 'goal recognition' rule. This corresponds, for example, to a sliding block puzzle where the 'goal' requires a subset of the blocks to be in particular locations irrespective of the positions of the others.

Secondly, we may specify that the path is *not* of primary importance, but that the main concern is to find a node which satisfies the 'goal recognition' rule. This situation seems appropriate to a range of applications including transportation problems, timetabling and allocation problems generally. The structure of the problem graph is now likely to be a matter of definition, rather than inherent in the problem itself.

Strategies and problem graphs

If the Graph Traverser's internal problem changes, then clearly its strategy

should change correspondingly. Why go to trouble to construct a solution path, when no path is required? The reader may have noticed that in each of the examples of the program at work, the problem graphs have been *symmetric*. When solving the Eight-puzzle, the program should perhaps have grown a search tree from the *goal* simultaneously with that grown from the start. This is an example of a more general point. The program assumes for the purpose of its search that the problem graph is quite general. However, the problem graph may well be symmetric, or a tree, or structured in some other way.

This raises two possibilities:

- (1) that the program should be *told* when the problem graph has some property which will make the search easier; and
- (2) more ambitiously, that the program should grow a small search tree, and then before continuing, inspect this tree for useful properties which it would then provisionally assume to hold for the entire problem graph.

Possibility (1) will be implemented in some form in the foreseeable future. (2) is much more complex, and will not be pursued further here, except to remark that in some circumstances it may be advantageous for the program to solve a problem under an assumption which is in fact not entirely sound, and then to 'correct' the solution.

'Pruning' schemes

There is a limit to the size which the search tree of the Graph Traverser can attain. Either this will be fixed by the storage capacity of the machine upon which the program is running, or else it will be determined by the time taken to work with a large tree. The reaction of the program when all the available storage space has been filled by the tree has already been described: a partial path is fixed upon, all the tree except a single node is erased, and a new partial search begins. This is unnecessarily catastrophic. There is no need to throw away so much hard-won information and it seems foolish to do so. The latest version of the Graph Traverser commits itself only to a single additional step along the path (or some specified number of steps), and only that part of the tree thereby rendered valueless is discarded. The distinction is illustrated in Fig. 3, where the significance of the diagrams is as follows:

- (a) The search tree at its growth limit (12 nodes).
- (b) The old version of the program ('static' tree) commits itself to a path to the undeveloped node with least value. This partial path is output. The new version of the program ('dynamic' tree) takes just one 'step', and this is output.
- (c) Deletion occurs. The dynamic tree version deletes only that part of the tree dependent on the portion of path selected.

(d) Two developments later. The situation illustrates how the old version may make a mistake.

Of course, the dynamic tree version of the program has to go through the 'pruning' process fairly frequently, and this is a disadvantage.

Table 3 presents the results of a simple experiment to compare the performance of the two versions of the program. Twenty-four Eight-puzzle configurations were used for this purpose. Twelve of these (test battery A) each required at least 30 moves for solution, this being the longest possible minimal solution path. The remainder (test battery C) were randomly chosen. The maximum size of the tree was fixed at 40 nodes, and the extent of the 'dynamic' pruning was determined by deciding that at each pruning operation a path segment three nodes long was to be output.

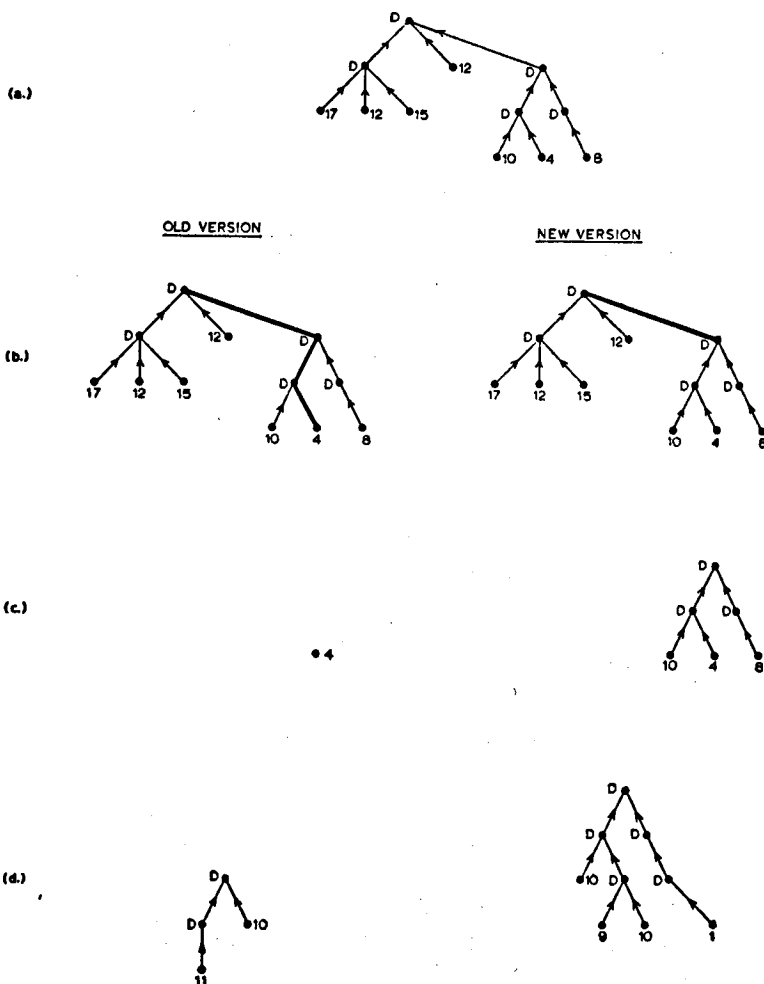


FIG. 3. Illustration of the operation of a 'dynamic' search tree as used by the latest version of the Graph Traverser. Earlier behaviour is shown for comparison.

TABLE 3

Performance of the old ('static' tree) version of the Graph Traverser compared with the new ('dynamic' tree) version over 24 Eight-puzzle configurations. The maximum tree-size permitted was 40 nodes, and a search was terminated (indicated by '—') when more than 200 developments had taken place without the goal having been located. Figures in brackets are minimal pathlengths. The evaluation function was not varied.

<i>Test battery 'A'</i>	'Static' tree		'Dynamic' tree	
	Nodes developed	Path length	Nodes developed	Path length
A1(30)	—	—	—	—
A2(30)	65	42	63	40
A3(30)	48	42	52	42
A4(30)	64	42	70	42
A5(30)	—	—	72	44
A6(30)	48	42	52	42
A7(30)	—	—	126	62
A8(30)	84	58	89	54
A9(30)	36	32	39	32
A10(30)	—	—	150	56
A11(30)	31	30	33	30
A12(30)	42	32	45	32
<i>Test battery 'C'</i>				
C1(20)	34	32	34	28
C2(24)	43	34	45	34
C3(22)	79	38	77	38
C4(22)	—	—	—	—
C5(24)	54	38	54	34
C6(24)	31	28	32	28
C7(22)	28	24	29	24
C8(14)	26	16	26	16
C9(22)	42	28	39	28
C10(24)	—	—	60	36
C11(8)	8	8	8	8
C12(18)	27	20	27	20

It will be seen that where a configuration was solved in both modes, performance was virtually identical. However, of the six configurations not solved by the 'static' version of the program four were solved by the 'dynamic' version, and none was solved by the former but not by the latter. Thus there is some evidence that the 'dynamic' tree is of real value just at the point where the old approach falters. As yet, of course, the evidence refers only to the Eight-puzzle.

It is easy to imagine, though less easy to implement, more complicated pruning schemes. For example, it would be possible to delete high-valued (i.e., unpromising) branches without committing the program to any additional path segment.

Program improvement of evaluation functions

Can the Graph Traverser improve its own evaluation function? If it can be made to do this, then the problems posed by the program's limited memory will have been attacked in a new way since the evaluation function itself will have become a repository of acquired knowledge.

Suppose that the program has been given a sequence of similar but not identical problems to solve, each problem involving several partial searches. It is required to find the 'best' evaluation function while solving the first few problems, and then to use this to solve the remainder. By 'best' evaluation function I mean here merely 'that setting of the parameters of the current function which optimises its performance over this class of problem'. Suppose that at the end of each partial search an adjustment of the function parameters is to be made. How can this be done so as to converge on the best function?

One attempt to answer this question runs as follows. If the program can calculate from a partial search a 'figure of merit' indicating how 'good' was the evaluation function which controlled that search, then it can seek to optimise the function by a standard hill-climbing operation. That is, it can carry out a sequence of partial searches with varying settings of the function parameters, and use its 'figure of merit' to converge on the optimal settings. The situation will typically be 'noisy', that is, the figure obtained will vary over different partial searches even for a fixed evaluation function.

The penetrance has already been mentioned as potentially useful as a general measure of the efficiency of a search—intuitively, it measures how 'confident' the search has been, independently of the tree size and the structure of the problem graph. Can the penetrance of a search be used as a 'figure of merit' in the sense introduced above?

We have seen that, when a search has been successfully completed, the penetrance has been strongly correlated with the development efficiency. To some extent this is to be expected—if the development efficiency is high, then the penetrance *must* be. However, the converse need not be true—it is possible for a search to have a high penetrance, but to be circuitous. For a small problem graph this hazard is slight, for the evaluation function can usually be made to take a minimum at the goal node giving little opportunity for deviation from the true path to go unrecognised. If the problem graph is large, however, so that several partial searches are needed, it seems quite possible that a partial search should have a high penetrance, but in fact be quite misdirected and therefore controlled by a poor evaluation function.

There is some experimental evidence on this point. The test battery of Fifteen-puzzle configurations already mentioned was attacked by the Graph Traverser with a number of different parameter settings for the evaluation function. Each attempt at a configuration involved at most two partial searches, each with a maximum tree size of 250 nodes. When mean penetrance over the first partial search was plotted against the proportion of configura-

tions solved within the second, a strong positive correlation was found, suggesting that here at least the penetrance was a guide to the effectiveness of the evaluation function.

Distance estimators

In the last section I discussed one possible answer to the question: 'How can the Graph Traverser adjust the parameters of its evaluation function at the end of each partial search so as to converge on the optimum settings?'

Another approach to this problem starts from the idea that the evaluation function, in the rather restricted sense in which it is being used here, is a particular case of something a little more general. As we have used it, an evaluation function effectively estimates the distance from a node to the goal node. Let us now define a *distance estimator* as a function which, when applied to any two nodes, specifically estimates the distance over the graph from the first to the second.

The purpose of this definition is to transfer attention from the '*goal seeking*' ideas associated with an evaluation function to '*graph describing*' ideas associated with the concept of a distance estimator. Notice that we may discuss good or bad distance estimators for a particular graph quite independently of any problem associated with that graph. Nevertheless, given a good distance estimator, *a fortiori* we have a good evaluation function for use in any particular search over the graph.

Now, if a distance estimator is required for a particular graph, but the information available is, in fact, merely a small fragment of the graph itself together with evidence that the graph is relatively 'homogeneous', then it is natural to find that estimator which best 'fits' the graph fragment available for inspection, and then to adopt this estimator for the graph as a whole.

But this is exactly the situation in which the Graph Traverser finds itself at the end of a partial search. The program has available its search tree, which is effectively a 'fragment' of the problem graph, and it is seeking a good distance estimator, as this automatically provides a good evaluation function. It is plausible therefore that initially the program should be provided with a rudimentary distance estimator which, at the end of each partial search, should be modified until it best 'fits' the search tree.

One specific method of performing such a 'fitting' process would begin by labelling each node of the tree with its 'depth' (as defined in Fig. 2). The depth of a node is the distance over the tree from the 'root' node to it. The program would then adjust the distance estimator until the correlation between these tree distances and the corresponding estimated distances given by the estimator was maximised. The tree distances that would be used in this calculation are not necessarily exactly the true distances over the problem graph. The method must assume, however, that they are nearly so.

Having obtained the distance estimator which best 'fits' the search tree, the program would then, as indicated above, use it as an evaluation function to guide the next partial search.

It is worth stressing the important practical difference between this approach to evaluation function optimisation, and that based on the penetrance. Using the penetrance, optimisation can take place only over a potentially rather long sequence of partial searches. It is hoped that the distance estimator approach, however, will permit considerable progress to be made *after just one partial search*, and that in general the optimisation process will be much speeded.

CONCLUSION

My aim has been to discuss problem-solving programs in a rather general way, and to describe in some detail the Graph Traverser program, its progress and its possible future. The Graph Traverser is not yet much more than an elementary exercise in machine intelligence. Even so, it will be instructive to end by seeking in the program three of the features of general problem-solving programs mentioned in the introduction.

Planning (see Minsky 1961) involves using additional information about a problem to obtain an 'outline' solution which will provide intermediate 'subgoals' for the search proper. This process does not appear explicitly in the Graph Traverser. Yet, in a rather vague sense, an evaluation function may be regarded as a generalisation of the notion of subgoals. Furthermore, the concept of operator-synthesis, the concatenation of several operators to form a macro-operator (Michie in this volume), is close to the type of planning which occurs in Gaku (Hormann 1965).

Induction, or the formation of a general rule from particular instances, will occur whenever the Graph Traverser seeks to draw general conclusions from the properties of some particular part of the problem graph. Relevant here are both the proposed improvement of an evaluation function by the study of a search tree, and also the suggestion that the program might induce structural properties of the problem graph from an inspection of a small portion of it.

Program-user interaction has as yet no place in the Graph Traverser. Yet it seems very plausible that problem-solving programs and human problem solvers are destined to work in co-operation rather than in isolation. The procedures 'develop' and 'evaluate' of the Graph Traverser could each involve interaction with an 'on-line' user. The value of such collaboration in actual problem-solving is difficult to judge, but the experiment would be interesting.

Indeed, the most useful role of the Graph Traverser is perhaps that of a 'test bed', not only for evaluation functions but for a wide range of ideas important to automatic problem-solving.

ACKNOWLEDGEMENTS

The research described in this paper is supported by the Science Research Council. I am greatly indebted to Dr Donald Michie, with whom I have carried out most of the experiments described here. He is the originator of

many of the better ideas which the paper contains. I am also grateful to my colleague Dr. R. M. Burstall for helpful criticisms and comments.

REFERENCES

- Berge, C. (1962). *The theory of graphs and its applications*. Alison Doig, Trs. London: Methuen.
- Bobrow, D. G. (1964). A question-answering system for high school algebra word problems. *AFIPS Conference proceedings—1964 Fall Joint Computer Conference*, Pt. 1, pp. 591-614. Baltimore: Spartan Books.
- Doran, J. E., & Michie, D. (1966). Experiments with the Graph Traverser program. *Proc. R. Soc. (A)*, 294, 235-259.
- Feigenbaum, E. A., & Feldman, J., Eds. (1963). *Computers and Thought*. New York: McGraw Hill.
- Hayes, J. E., Michie, D., Pole, K. E., & Schofield, P. D. A. (1965). A quantitative study of problem-solving using sliding block puzzles: the Eight-puzzle and a modified version of the Alexander Passalong Test. *Experimental Programming Reports*. No. 7. Edinburgh: Experimental Programming Unit, University of Edinburgh.
- Hormann, A. (1965). Gaku: an artificial student. *Behav. Sci.*, 10, 88-107.
- Minsky, M. (1961). Steps towards artificial intelligence. *Proc. Inst. Radio Engrs.*, 49, 8-30.
- Newell, A., & Ernst, G. (1965). The search for generality. *Information Processing 1965: Proceedings of IFIP Congress 1965*. Vol. 1, pp. 17-24. Kalenich, W. A., ed. London: Macmillan.
- Newell, A., Shaw, J. C., & Simon, H. A. (1960). A variety of intelligent learning in a general problem solver. *Self-organising systems*. Yovits, M. C., & Cameron, S., eds. pp. 153-189. London: Pergamon Press.
- Raphael, B. (1964). A computer program which 'understands'. *AFIPS Conference proceedings—1964 Fall Joint Computer Conference*. Pt. 1, pp. 577-589. Baltimore: Spartan Books.
- Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *I.B.M. J. Res. Devl.*, 3, 211-229. Reprinted in Feigenbaum and Feldman (1963).
- Slagle, J. R. (1963). A heuristic program that solves symbolic integration problems in freshman calculus. *Computers and Thought*. Feigenbaum, E. A., & Feldman, J., eds. pp. 191-203. New York: McGraw Hill.



8

COMPLETE SOLUTION OF THE 'EIGHT-PUZZLE'

P. D. A. SCHOFIELD
COMPUTER UNIT
UNIVERSITY OF EDINBURGH

'For the last few weeks, the "Fifteen-puzzle" has been prominently before the American Public, and may safely be said to have engaged the attention of nine out of ten persons of both sexes and of all ages and conditions of the community.'

American Journal of Mathematics (1879)

The 'Eight-puzzle' is a reduced form of the 'Fifteen-puzzle', the subject of the somewhat extravagant claim quoted above. Its use in the study of learning processes, both human and programmed, is described in two other papers in this volume, Michie (p. 135) and Doran (p. 105). This paper describes the calculation of optimum solutions to all the 20 160 possible versions of the puzzle.

The 'Eight-puzzle' consists of eight square pieces, numbered 0-7,† capable of sliding in a shallow square tray of size nine times that of the individual pieces: there is thus one empty square. We define a *standard position* as one in which the centre square is empty, and the problem discussed here is that of *sliding* the pieces (i.e., without lifting any piece over another) from one standard position to another in the minimum number of moves.

Without loss of generality, we can obviously select either one fixed starting configuration or one fixed target configuration. For an exhaustive computation, the former is convenient, but in the heuristic programs and experiments described in Michie's and Doran's papers the latter has usually been chosen.

† Piece 0 in this paper corresponds to piece 8 in Michie's and Doran's papers. 0 has been used in the program described below, for ease of representation in 3 binary digits within the computer.

NOTATION

If we regard the squares as being numbered

1	2	3
0		4
7	6	5

then the natural fixed configuration from which to start is that in which piece 0 occupies square 0, etc.

It is convenient to describe permutations of standard positions in terms of numbered squares, which are fixed, rather than numbered pieces, which move. Thus the transition from

5	3	4
0		2
1	7	6

to

3	4	5
0		2
7	1	6

would be expressed in terms of the disjoint cycles:

(132) (0) (4) (5) (67) or more briefly (132) (67).

(I.e., the piece initially in square 1 goes to square 3, the piece initially in square 3 goes to square 2, the piece initially in square 2 goes to square 1. The pieces in squares 6 and 7 are transposed.)

A *corner twist* is a permutation (starting from a standard position) causing the three pieces nearest to a corner to be permuted cyclically. A clockwise permutation of three pieces is described as twist A, B, C or D depending on the corner concerned:



i.e., $A = (012)$
 $B = (234)$
 $C = (456)$
 $D = (670)$

Notes. (i) A corner twist is a permutation of order 3: $A^3 = I$, the identity, and so $A^2 = A^{-1} = (210)$ is an anti-clockwise permutation about the top left-hand corner. There are thus eight possible corner twists: A, B, C, D, A^{-1} , B^{-1} , C^{-1} , D^{-1} .

(ii) A corner twist can be expressed as a pair of transpositions, e.g., $(012) = (01)(20)$, and is an even permutation.

(iii) A corner twist can be carried out in four moves (one piece moves to the centre, two moves round the outside, and the centre piece is moved out). However, as will be shown, a single transposition, being an odd permutation, cannot be carried out within the rules of the puzzle.

It follows from Johnson & Story (1879) that a puzzle can be solved (without lifting pieces over one another) if, and only if, the target position is an even permutation of the starting position. In the case of the 'Eight-puzzle', this may be seen more directly by reference to 'corner twists'. (The

arguments below will most easily be followed if a model of the Eight-puzzle is used.)

Theorem 1. In all solvable puzzles, the starting and finishing positions are even permutations of one another.

Proof. Consider any solution which involves no 'back-tracking' (immediate cancellation of a move or series of moves). The same effect can be produced by a succession of corner twists, since, if the fourth move does not complete a corner twist, two dummy moves can now be inserted (one to the centre to complete the twist, the second immediately cancelling the previous move). This process can be continued, along the chosen solution. Hence the solution consists of a product of even permutations.

To demonstrate the converse, consider:

Lemma A. A cyclic permutation of any three adjacent squares is either a corner twist or can be produced by a succession of 4 corner twists.

Proof. If not a corner twist, it must be a permutation along one edge, e.g., (123). This may be expressed:

$$(123) = (321) \quad (321) = (12) \quad (23) \quad (01) \quad (01) \quad (34) \quad (34) \quad (12) \quad (23)$$

and as disjoint transpositions can be commuted, this is:

$$(12) \quad (01) \quad (23) \quad (34) \quad (01) \quad (12) \quad (34) \quad (23) = AB^{-1} A^{-1}B$$

and similarly along the other edges.

Note. In fact, the 16-move solution thus obtained is an optimum.

Lemma B. Any transposition is equivalent to a succession of legal 3-cycles (as in Lemma A), followed or preceded by a transposition of adjacent pieces.

The form of this can be seen by the example:

$$(15) = (12) \quad (23) \quad (34) \quad (45) \quad (34) \quad (23) \quad (12)$$

which can be collected *in pairs* to give a succession of legal moves (as in Lemma A), followed or preceded by (12) as required.

Theorem 2. If the starting and finishing positions are even permutations of one another, then the puzzle is solvable.

Proof. Expressing the even permutation as transpositions, it is evidently sufficient to show that a pair of transpositions can be produced legally.

From Lemma B, the first can be expressed as legal moves followed by a transposition of adjacent pieces, and the second as legal moves preceded by transposition of adjacent pieces. If the outstanding transpositions are disjoint, insert dummy pairs of transpositions, e.g.

$(15) \quad (47) = \text{legal moves} \dots \overline{(12) \quad (23)} \quad \overline{(23) \quad (34)} \quad \overline{(34) \quad (45)} \dots \text{legal moves}$
giving a legal method of solution by combining in pairs and using Lemma A.

THE ALTERNATING GROUP

Hence the solvable puzzles correspond to the even permutations of eight elements, that is to the Alternating Group A_8 , of order $\frac{1}{2} \cdot 8! = 20\,160$.

THE 'FIVE-PUZZLE'

Before embarking on a computer program to obtain the minimum path solutions to the eight-puzzle, the present writer wished to obtain an upper bound for the number of moves required in the most difficult cases. This was obtained by first considering the five-puzzle (the eight-puzzle with the bottom row missing, thus losing some symmetry properties), e.g.,

1	2	3
0		4

The previous theorems apply, but the group concerned is the very much smaller Alternating Group A_5 , of order $\frac{1}{2} 5! = 60$.

All paths in this puzzle can be expressed in terms of the four corner twists A, B, A^{-1}, B^{-1} . In general, the number of moves involved is four times the number of twists, but this is reduced if, for example, twist B is followed by A . Given a starting position,

2	3	1
0		4

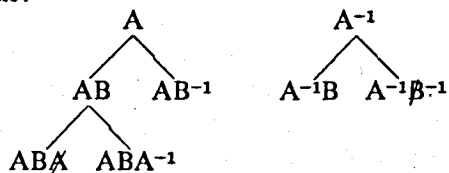
the pieces moved would be 4 1 3 4 and then 4 2 0 4. As the fourth move shifts piece 4 only for the fifth move, to replace it at once, these 8 moves can be reduced to 6—i.e., 4 1 3 2 0 4.

We call this one *elision* and write the path $B\cancel{A}$. The same applies when A^{-1} is followed by B^{-1} . The number of moves, m , is given in terms of the number of twists, t , and elisions, e , as: $m = 4t - 2e$. Hence the path:

$AB^{-1}A^{-1}B$ consists of 16 moves, while
 $ABAB$ consists of 14 moves.

Any path between two standard positions can be expressed uniquely in this way.

Of the four possible starting twists, only two, A and A^{-1} say, need be followed, those resulting from an initial B^{-1} or B being deduced by reflection in the vertical axis:



After following these binary trees through 6 twists, all elements of the group A_5 had been generated, the maximum number of moves required being 20. From this a very crude upper bound of 48 moves was deduced for the eight-puzzle.

Before leaving the five-puzzle, it is of interest to note that the group A_5 is isomorphic to the dodecahedral group, and that the axes of symmetry of dodecahedron correspond to the classes of permutation within the five-puzzle as follows:

SCHOFIELD

No. and class of permutations	Axes of dodecahedron
20 cyclic permutations of 3 pieces	10 pairs of opposite vertices (rotations of $2\pi/3$, $4\pi/3$ about each)
24 cyclic permutations of 5 pieces	6 pairs of opposite faces (rotations of $2\pi/5$, $4\pi/5$, $6\pi/5$, $8\pi/5$ about each)
15 pairs of transpositions	15 pairs of opposite edges

SYMMETRY PROPERTIES OF THE 'EIGHT-PUZZLE'

The greater symmetry of the eight-puzzle enables us to carry out an exhaustive trial of all possible paths which might lead to an optimum solution of some puzzle, by starting from one corner twist only (A, say). By rotating the resulting permutation through $\pi/2$, π , $3\pi/2$ and 2π , we reach corresponding positions for which the initial corner twist would have been B, C, D, A respectively, and a subsequent reflection in the vertical axis gives corresponding situations with initial corner twists A^{-1} , D^{-1} , C^{-1} , B^{-1} .

These eight permutations, which are not necessarily distinct, are regarded as forming an *equivalence class*, since a minimum-move solution to any member of the class leads immediately to a minimum-move solution to the remainder. (If X is any given permutation of a standard position, the other members of the equivalence class are given by $\phi_i^{-1} X \phi_i$ where the ϕ_i are the eight elements of the dihedral group D_4 , and the minimum path solutions are obtained by making the same transformations to the corner twists of the solution to the puzzle represented by X.)

The 20 160 possible versions of the eight-puzzle break down into 2572 such equivalence classes, with a consequent saving of computation and storage space.

A further reduction may be made in the number of essentially different puzzles which need be studied, by regarding inverse permutations as belonging to the same equivalence class as one another, thus giving a maximum of 16 members of a class. In this case, the number of equivalence classes reduces to 1439. Although this was done in an early version of the program, it was found less intermediate information needs to be held in the case of the 2572 class version.

ELISION

As with the five-puzzle, the number of moves taken may be less than four times the number of twists, on account of elision. Of the eight twists which might follow a given starting twist (A, say),

- (i) Five (B, C, B^{-1} , C^{-1} , D^{-1} in this case) will give no elision.
- (ii) One (D) gives one elision,
- (iii) Two (A, A^{-1}) will never be used in an efficient solution, since $AA = A^{-1}$ and $AA^{-1} = I$.

PROGRAM FOR THE ATLAS COMPUTER

A program was written for the Atlas computer to calculate the minimum move solutions to the 20 160 possible puzzles. Two dictionaries of permutations were held in the machine:

(i) *Main dictionary.* This had one entry for each possible permutation, and simply stated whether this particular situation had been solved so far, and if so, the length of the minimum path solution. (Only 5 bits needed for each entry.)

(ii) *Short dictionary.* This was built up as solutions were found and contained one representative of each equivalence class, as described in 'Symmetry properties of the "Eight-puzzle"' (p. 129). With each permutation stored, there was a list of corner twists which could be the last twist used in a minimum move solution. This list was increased whenever a given permutation was reached by a new route, using the same number of moves.

STORAGE OF PERMUTATIONS IN MAIN DICTIONARY

For economy in space and to avoid searching down the main dictionary, it is desirable to map the even permutations uniquely on to the integers 0 to 20 159.

If $C_i, i=0, 1, \dots, 7$ is the number on the piece in square i , then

$$S = \frac{1}{2} \sum_{i=0}^5 \left\{ (7-i)! \sum_{j=i+1}^7 H(C_i - C_j) \right\} \quad (1)$$

gives such a mapping, where $H(t) = \begin{cases} 1 & \text{if } t > 0 \\ 0 & \text{if } t < 0 \end{cases}$

(In the 'target' ordering given in Fig. 1, $C_i=i$, and the sequence number S is 0, whereas in the position

6	5	4
7		3
1	0	2

all values of $H(C_i - C_j)$
are 1 and so

$$S = \frac{1}{2} \sum_{i=0}^5 \left\{ (7-i)! (7-i) \right\} = \frac{1}{2} \sum_{i=0}^5 \left\{ (8-i)! - (7-i)! \right\} = \frac{1}{2} \{ 8! - 2! \} = 20\,159.$$

GENERAL STRATEGY

The process of exhausting all possibilities was carried out as follows:

1. Set identity permutation as solved in 0 moves (main dictionary).
Place it as first entry on short dictionary.
Set all corner twists as solved in 4 moves (main dictionary).
Place twist A as second entry on short dictionary.
Set $m=6$.

SCHOFIELD

2. Examine all permutations on short dictionary reached in $(m-4)$ moves and apply all twists which involve no elision. Examine all permutations reached in $(m-2)$ moves and apply twists which do achieve elision.
3. For each permutation produced in stage 2, look at the main dictionary entry (direct look-up in an array using sequence number computed as in equation (1)).

(i) *If unsolved to date*

(a) In main dictionary, record all members of equivalence class as solved in m moves.

(b) Put one representative of class on short dictionary, together with terminal twist(s) which could be used.

(ii) *Already solved in m moves*

Find the representative member in the short dictionary and, if necessary, add to list of possible terminal twists.

(iii) *Already solved in less than m moves*

No action.

4. If any new permutations have been found in m or $(m-2)$ moves, increase m by 2 and return to stage 2. Otherwise process is complete.

Using the above strategy, the numbers of permutations which can be achieved in m moves were calculated to be:

No. of moves	Total permutations	No of equivalence classes	No. of extended equivalence classes (counting inverses)
0	1	1	1
4	8	1	1
6	8	1	1
8	40	6	5
10	88	11	6
12	232	29	18
14	556	70	40
16	1 254	159	94
18	2 456	312	167
20	4 020	506	278
22	5 048	638	340
24	4 121	529	298
26	1 902	248	139
28	366	49	39
30	60	12	12
<i>Total</i>	<u>20 160</u>	<u>2 572</u>	<u>1 439</u>

PRINTING OF RESULTS

The printed results were designed for use in experiments described in Michie's (pp. 135-52) and Doran's (pp. 105-23) papers and therefore assume Fig. 1 to be the *target* position. Although one member of each equivalence class would be sufficient, the method of look-up would be tedious and so a complete table was printed. Against each possible puzzle is printed:

MACHINE LEARNING AND HEURISTIC PROGRAMMING

- (i) the starting twists from which it is possible to achieve a minimum path solution;
- (ii) the length of the minimum path;
- (iii) a specimen minimum path.

The notation for twists is as described on page 126 except that to economise in space across the page, A^{-1} is printed as W, B^{-1} as X, C^{-1} as Y and D^{-1} as Z. The complete tabulation covers 168 pages, of which a specimen half-page is given opposite, together with an explanation of the method of use.

USE OF TABLES

To identify a particular soluble puzzle, it is only necessary to consider the numbers in 6 of the 8 positions, as the sequence of the remaining two is determined by the condition that the permutation be even. The tables are based on the numbers initially in the squares whose target numbers are 0 1 2 3 4 5. Hence, the puzzle

0	2	5
1		3
7	4	6

would be looked up under the code numbers 102536.

In the tables (see Table 1), the first three of these numbers are given at the head of the half-page, the next three on the left of the corresponding solution. For the above case, we read that the minimum path solution is 20 moves long, that such paths can start with either twist W or Z, and that a specimen solution is WXYZADY.

REFERENCES

- Hayes, J. E., Michie, D., Pole, K. E., & Schofield, P. D. A. (1965). A quantitative study of problem-solving using sliding block puzzles: the 'Eight-puzzle' and a modified version of the Alexander Passalong Test. *Experimental Programming Reports*, No. 7 (revised ed.). Edinburgh: Experimental Programming Unit, University of Edinburgh.
- Johnson, W. W., & Storey, W. E. (1879). Notes on the '15' puzzle. *Amer. J. Math.*, 2, 397-404.

1	0	2
3 4 5) DC..... 16 D EF XY	3 4 6) DC..W..Z 18 D EF ZXX	3 4 7) DC..... 20 D EF WXY
3 5 4) D.B..... 16 D EF ZXX	3 5 6) D..... 14 D EF XY	3 5 7) D.....Y. 18 D EF WXY
3 6 4) D.B..... 18 D EF ZXC	3 6 5) D.B...YZ 20 D EF ZXC	3 6 7) .C..... 18 CD EF XY
3 7 4) .B..... 14 BXYX	3 7 5) .B...Y. 18 BXYZC	3 7 6) .B..... 18 BXYZY
4 3 5) D.B.W.Y. 18 D EF YXX	4 3 6) D.B.W.Y. 22 DWZAB EF	4 3 7) D.B.W.Y. 22 D EF ZYXX
4 5 3) DC..... 20 D EF ZXY	4 5 6)Z 16 Z EF XY	4 5 7) D..... 22 DWZ EF XY
4 6 3) .C..X... 20 CZ EF XY	4 6 5) D..... 20 DWZ EF XY	4 6 7)Z 20 Z EF CD EF XY
4 7 3) .B..... 22 BXYZXY	4 7 5) .B..... 18 BXYZXY	4 7 6) D.....Y. 24 DWZ EF XY
5 3 4) DC..... 20 D EF ZXY	5 3 6)N..Z 20 WXYZXY	5 3 7) .C..... 22 CD EF ZXY
5 4 3) E.B..... 20 D EF ZXY	5 4 6) .B..... 20 BD EF ZXY	5 4 7)Y. 22 YBXYZXY
5 6 3) D..... 18 D EF ZXY	5 6 4)Z 20 Z EF CD EF XY	5 6 7) D..... 20 D EF ZXY
5 7 3) .B..... 18 BXYZXY	5 7 4) DC...X. 24 D EF ZXY	5 7 6) .B...Y. 22 BXYZXY
6 3 4)W... 16 WXYZXY	6 3 5) D..... 18 D EF ZXY	6 3 7)W... 20 WXYZXY
6 4 3) D..... 22 D EF ZXY	6 4 5) .C.....Z 20 CXYZXY	6 4 7) D.B..... 24 D EF ZXY
6 5 3)X. 20 YXYZXY	6 5 4) B.B..... 24 DZBXYZXY	6 5 7) .C..... 22 CD EF ZXY
6 7 3) .B...Y. 22 BXYZXY	6 7 4) .B..... 22 BXYZXY	6 7 5) D..... 24 D EF CD EF XY
7 3 4) .C.....Z 22 CXYZXY	7 3 5)Z 22 Z EF CD EF XY	7 3 6)Z 24 ZXYZXY
7 4 3)X.. 20 XXYZXY	7 4 5) D.B..X.. 24 DXZAB	7 4 6)XY. 24 XXYZXY
7 5 3) .B..X.. 24 BXYZXY	7 5 4) .C...Y. 24 CXYZXY	7 5 6) D..... 24 D EF ZXY
7 6 3) .B..... 20 BXYZXY	7 6 4) .B..... 24 BXYZXY	7 6 5) ..B..... 24 BXYZXY

TABLE I

Specimen half-page from the computer tabulation of Eight-puzzle solutions.



9

STRATEGY-BUILDING WITH THE GRAPH TRAVERSER

D. MICHIE

EXPERIMENTAL PROGRAMMING UNIT
UNIVERSITY OF EDINBURGH

I shall discuss automatic methods of search for solutions in problems susceptible of a particular formal representation, namely that on which the Graph Traverser program (Doran & Michie 1966, and see Doran p. 105) has been based. In this representation which is essentially that of Newell, Shaw & Simon (1960) a problem consists of a set of *states*, one or more of which is labelled 'goal', together with a rule-book. The rule-book lays down for each state what *moves* may be made from it to reach other, neighbouring, states. Solution consists in a demonstration that a goal state can be reached from some given initial state via a sequence of intermediate states, sometimes with the additional requirement that an actual sequence, or *path*, be demonstrated as part of the solution; in this last case, as in the sliding block problem discussed later, the goal (or goals) is typically specified in full as part of the statement of the problem. In other cases, where the goal is only specified in terms of some defining property (such as cost in a transportation problem, or simplicity in a problem of algebraic manipulation) we are interested solely in discovering states possessing this property, and not at all interested in the particular paths leading to them. Indeed the *only* thing 'given' in the problem statement apart from the description of the set of states may be the defining property of the goal state(s): in that case, if the method of attack is to remain within the formal bounds considered here, either the problem-solving program or its author must propose an initial state from which to launch the search, and must also make up the rule-book. Burstall's program for designing electricity distribution networks (p. 65) provides an excellent illustration of this last eventuality. In Burstall's (see pp. 65-85) problem a solution is defined as a network, the calculated

cost of which does not exceed some stated quantity, and which satisfies certain security constraints. After an initial trial network has been proposed, the program then follows rules invented to allow moves of the following two elementary kinds within the limits set by the security constraints:

- (1) adding a line joining points i and j for any $i \neq j$;
- (2) deleting a line joining points i and j for any $i \neq j$;
together with two more obtained as compounds of these; namely
- (3) two operations of (1) above;
- (4) two or more operations of (2) above.

For any substantial number of points in the network, the number of possible moves which can be generated in these four categories becomes very large, necessitating drastic methods of selection. The way in which this selection is effected is discussed by Burstall (*loc. cit.*). The point I wish to make here is that the basic moves by which neighbouring states are inter-transformable *may* be unalterably fixed in the structure of the problem as given, or definitions of allowable moves *may* be imported into the problem in order to give it a structure susceptible to general search methods. A fuller classification of problem types is given by Doran & Michie (1966) in a paper to which the reader is referred for the background information which the present discussion takes for granted.

TWO APPROACHES

Work on automatic problem-solving has tended in the past to polarise around two basic approaches. One approach, based on *state-evaluation*, generates all the states of the problem which can be reached in a small number of moves from the current state, and then seeks by some process of evaluation to decide which state shall form the next point of departure. Most, but not all, of the work on game-playing has been of this kind (for review see Michie 1966). The other approach applies selection *not* to the individual states of the problem but rather to the set of operators used for effecting transformations of state. In the classical studies of Newell, Shaw & Simon (1960) selection is applied by going down a priority sequence of operators, applying to each in turn a number of *tests*, first of applicability to the current state and then of whether the operator conduces towards one or another of various desirable intermediate states, or *subgoals*. I shall refer to this approach as placing the emphasis on *operator-selection*.

THREE STAGES

Present and planned work with the Graph Traverser program can be divided into three stages.

In *stage 1*, which has monopolised our attention so far, we concentrate on the exploitation and improvement of evaluation functions, with operator-selection deliberately side-tracked. For this purpose the program's 'develop'

procedure (which operates upon the given state of the problem to generate descendant states) is set simply to generate immediate descendants by applying the elementary moves. In other words, in this mode the develop procedure contains *only* the rule-book.

In *stage 2* the develop procedure is extended so that it (a) generates operators with a more ambitious reach than the elementary moves by stringing together combinations of the latter, and (b) applies to a given state not all available operators but only the most promising selection from the full set. In this stage, selection is integrated with state-evaluation into a unified system, suitable for use as a tool for improving an already-existing problem-solving strategy.

In *stage 3* the program is endowed with the ability, in any given problem environment, to construct capabilities (a) and (b) for itself out of the fruits of its own experience of that environment. At this stage the program is no longer a tool in the hand of the designer but acquires something of the designer's own role.

Stage 1, which has already been implemented and extensively tested, is discussed elsewhere in this volume (Doran, p. 105). Stage 2 has been the subject of some preliminary experimentation which I shall here report, small in amount, but sufficient for an initial demonstration of feasibility. Stage 3 belongs to the future. I shall conclude my account by detailing a series of evolutionary steps leading from the most primitive form of stage 1 to a fully-fledged form of stage 3, each step needing for its accomplishment no more than the systematic application of existing methods.

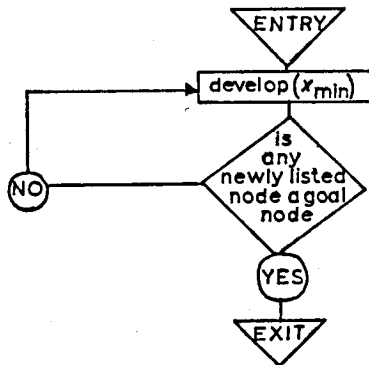
THE SYNTHESIS OF OPERATORS AND THE NEED FOR PRUNING

Elementary moves may be strung together to form compound moves. Suppose that we have three moves at our disposal called, say 'jump', 'hop' and 'skip', and that the sequence 'hop, then skip, then jump' is found often to be useful. Then it may be advantageous to add a new operator 'hop-skip-jump' to the armoury. 'Operator' is here used as a generic term to denote both moves and compound moves. I. J. Good once referred conversationally to the use of highly compounded chains of moves as 'the method of seven-league boots'. In the spirit of this phrase one should perhaps regard Burstall's (see pp. 65-85) program as proceeding in part in one-league boots (elementary moves) and in part in two-league boots.

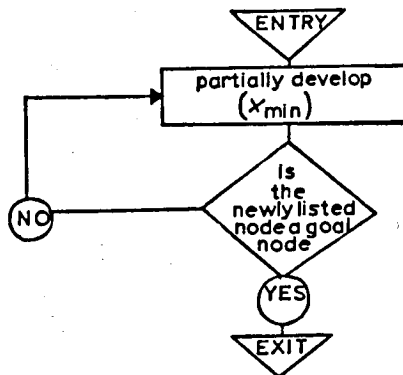
The main difficulty about the method of seven-league boots is that the added range brings with it a proliferation of choices. If for a given problem there are available an average of m moves from each problem-state, then the number of operators which can be formed by compounding sequences of n moves will be of the order of m^n . Drastic methods of pruning are therefore desirable, and it is this pruning process to which we refer when we talk about 'operator selection'.

SELECTION BY PARTIAL DEVELOPMENT

Consider the rather striking consequence which follows from an extremely natural modification of the current version of the Graph Traverser. In its present form the cycle of iteration is as follows:



where x_{min} denotes the minimum-valued undeveloped node. Note that since 'develop' involves applying in turn *all* the operators in the set, no selection is involved. Now consider what would happen if for 'develop' we substituted 'partially develop' denoting by this a procedure which produces only *one* descendant for evaluation (compare the 'short-listing' procedure proposed by Maynard Smith & Michie 1961). First envisage this being done by a random selection from among those members of its operator set which had not previously been applied to the node in question. We now have the following picture of the main loop:



i.e., list *one* descendant not previously listed and evaluate it.

The effect on the search strategy brought about by this innocent-looking modification is radical. It can most easily be illustrated by assuming a hypothetical case, typical of those with which real life abounds, in which the operator set is very large, so that to develop completely one node of the problem-graph would entail the drawing of literally thousands of lines radiating from it. Now suppose that at some stage of the search x_{min} has

the value 200, and that the develop procedure is called by the program. Old style, the application of operators to locate next-neighbours marches grimly on, as in Fig. 1, with no means of arresting the process short of locating *all* the descendants of x_{min} even though it is clear from the very first *partial* development that x_1 is a more promising node than x_{min} for further development. The new-style search shifts the label ' x_{min} ' as soon as x_1 has been evaluated and the relation $x_1 < x_{min}$ discovered, as in Fig. 2.

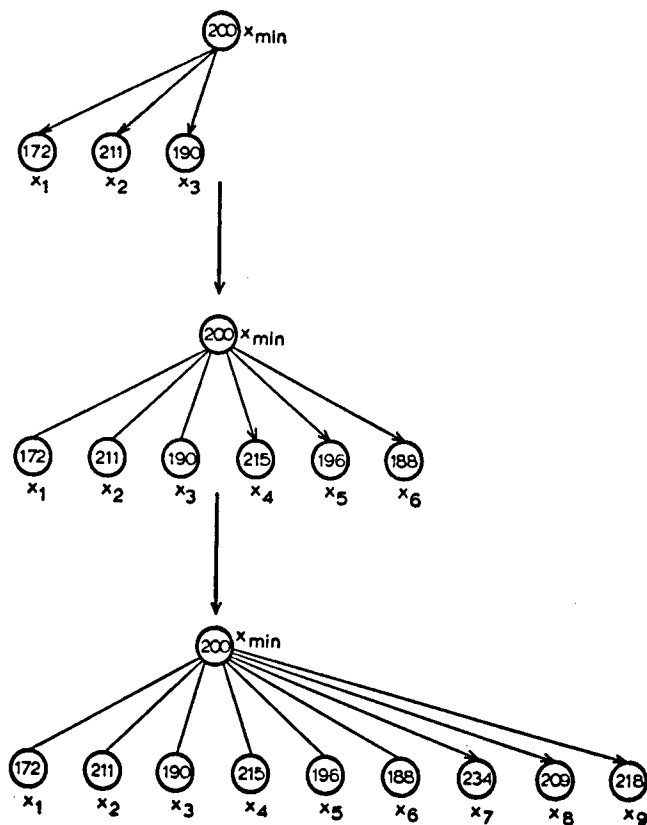


FIG. 1.

A considerable pruning of the search tree can be achieved by this fundamentally simple extension of the Graph Traverser's philosophy of opportunism, as has been well shown by Popplestone (pp. 31-46) in his application of essentially this form of the program to a theorem-proving problem.

CONDITIONAL CHOICE STRATEGIES

In the extreme case of operator selection only *one* operator is chosen for application at each given state of the problem, and this choice is made according to abstracted strategic features of the given state. Thus in algebraic

simplification one might follow a rule that *whenever* an expression of the form $\frac{a}{b} + \frac{c}{d}$ appeared, the compound move 'cross-multiplication' should be applied.

Indeed a conditional-choice strategy is almost invariably what the human designer or mathematician or programmer produces when he is asked to devise a strategy for a specific problem. The search then proceeds, not as a spreading tree in the manner of Fig. 1 of Doran's paper (p. 109) (trees grow *downwards* in this book), but more like the non-branching stem of a bamboo as shown in Fig. 3. Each node in the diagram can be expanded, to show what goes on at each stage, into the form shown in Fig. 4.

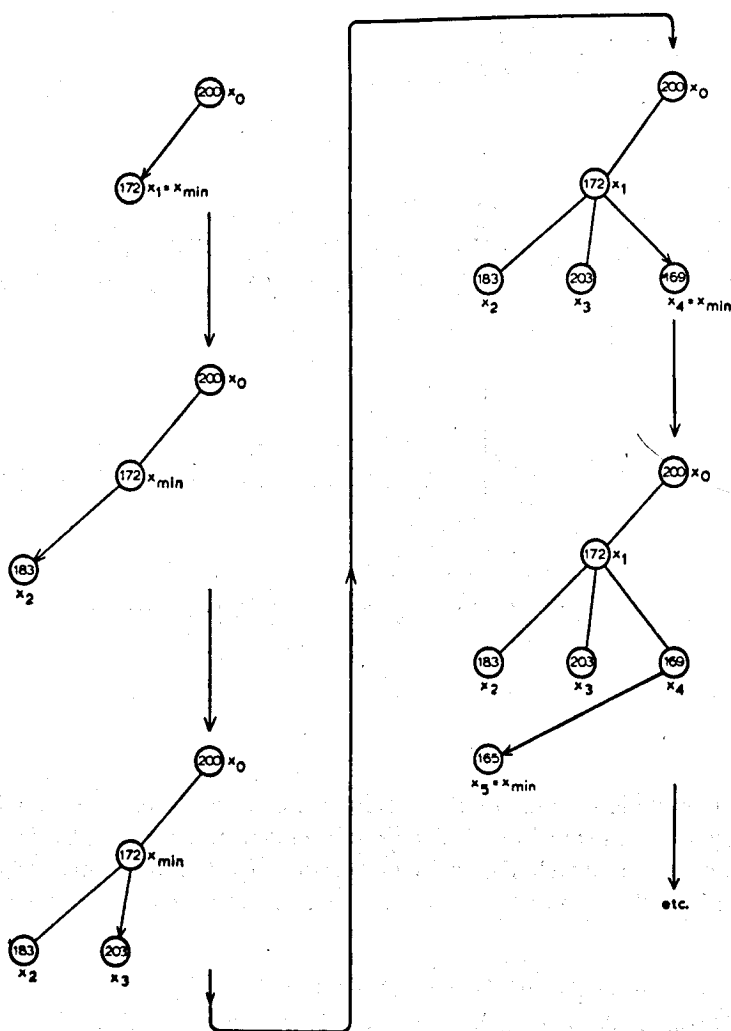


FIG. 2.

The nature of such algorithms can be illustrated rather vividly by considering the task of devising a conditional-choice strategy for a simple puzzle, the Eight-puzzle, of which we have made much use while developing the Graph Traverser program.

The Eight-puzzle is a poor relation of the well-known Fifteen-puzzle (Johnson & Storey 1879, Tait 1880). It requires the rearrangement of eight numbered blocks by a succession of sliding movements within a 3×3 square frame. We take, by convention, the following configuration to be the goal,

1	2	3
8	0	4
7	6	5

where the zero stands for the empty space. P. D. A. Schofield (pp. 125-33 of this volume) has shown that the $8! - 1$ possible centre-empty starting configurations are distributed over a range of difficulty from minimal paths of 4 moves up to minimal paths of 30 moves. Intelligent human subjects with no previous

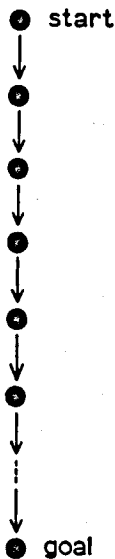


FIG. 3. A conditional-choice strategy depicted as a non-branching tree.

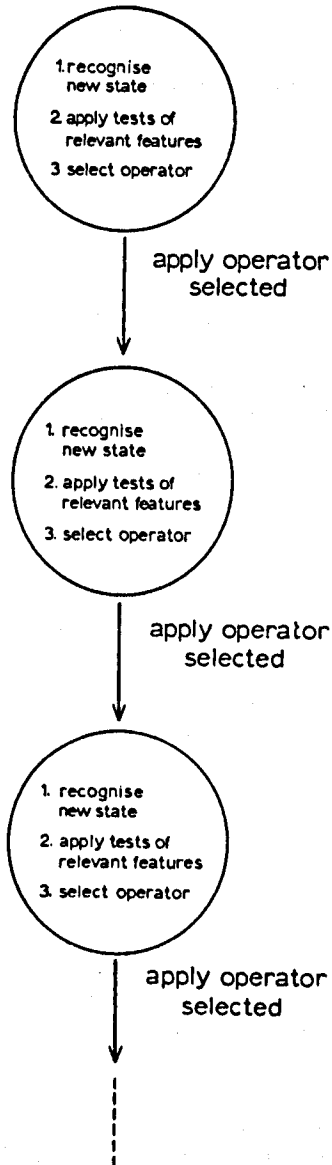


FIG. 4. Expansion of Fig. 3 to show what happens at each node.

experience of the puzzle achieve path efficiencies in the range 30-40 per cent for the more difficult configurations, where path efficiency

$$= \frac{\text{no. of moves in minimal path}}{\text{no. of moves in actual path}}$$

With intensive practice the average rises to about 50 per cent. We have found one individual, Mr Roger Chambers, who can consistently score 90 per cent.

A conditional-choice strategy giving about 75 per cent path efficiency can be fairly easily designed by setting up as *first* objective that the longest existing numerical sequence of blocks (like the string 456 in the following example)

6	8	2
5	0	7
4	1	3

should be extended until a sequence of six or more has been achieved, and as *second* objective that the blocks should be got to their homes by a sustained process of circulating them around the outer 'ring road'.

The flow chart shown in Fig. 5 depicts the main loop of this rather simple, but moderately effective, 'simple chaining' strategy. I am now going to indicate how such a strategy can be neatly embedded in the 'develop' procedure of the Graph Traverser, in such a way that the 'evaluate' procedure can be used to hammer the weak points into shape and so improve the strategy. My object will be to show, with the aid of this toy problem, the immediate potential applicability of the Graph Traverser as a design tool.

DESIGNING A STRATEGY FROM THE TOP DOWN

Let us view the simple chaining strategy described above as a means of developing a node of the problem graph, in the sense of locating its immediate descendant(s) for evaluation. We write 'descendant(s)' with the terminal 's' bracketed because the whole point of a conditional-choice strategy in its complete form is that only *one* operator is selected at each stage; any evaluation of the resulting single descendant node would thus be a phantom calculation without effect on the search. This is evident from the flow chart shown in Fig. 6, in which the main Graph Traverser program itself plays a dummy role, merely standing as a gateway between the exit and entry of successive applications of a conditional-choice strategy. I shall now say why I want to place conditional-choice strategies within the Graph Traverser framework.

The weakness of a conditional-choice strategy, as with man-made algorithms generally, is that while *some* of the mappings from state-features to operator-choices may have been made by the algorithm's author with great confidence, there are likely to be others where he has specified an operator more because *something* must be specified than because he knows that the choice is the best possible in the presence of the given features.

An example of the former category in the case of the Eight-puzzle might be the following:

2	3	4
1	0	6
8	7	5

where the conjunction of the reversal of the two pieces '5' and '6' with

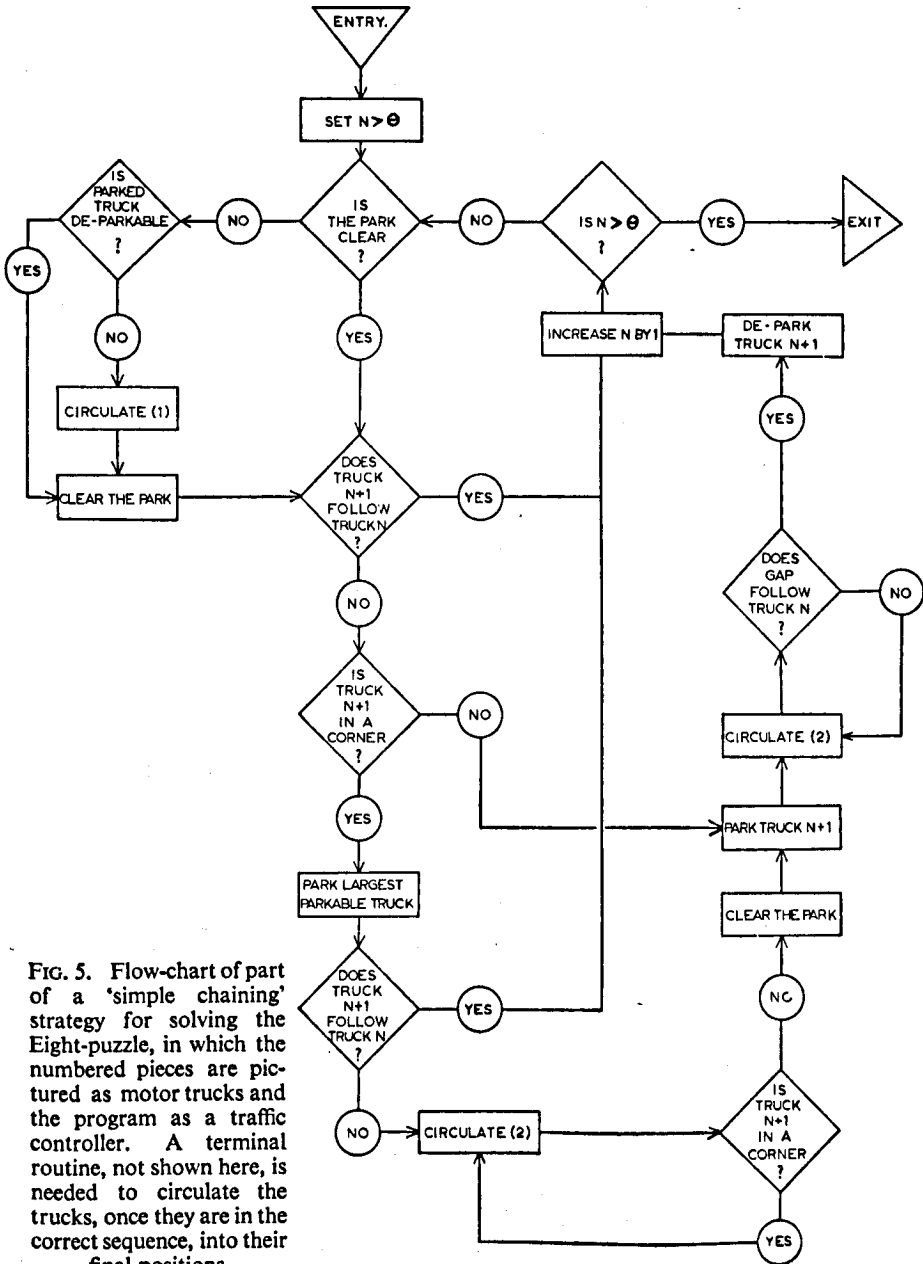


FIG. 5. Flow-chart of part of a 'simple chaining' strategy for solving the Eight-puzzle, in which the numbered pieces are pictured as motor trucks and the program as a traffic controller. A terminal routine, not shown here, is needed to circulate the trucks, once they are in the correct sequence, into their final positions.

MACHINE LEARNING AND HEURISTIC PROGRAMMING

respect to their proper order and the cyclic displacement by one space of all other pieces is sufficient to specify the operator 'B.A.D.' (Schofield's notation, meaning 'park from square no. 4—we assume the squares to be numbered cyclically—circulate clockwise 6 spaces, and depark'.) In other cases, such as the following

7	4	5
2	0	3
6	1	8

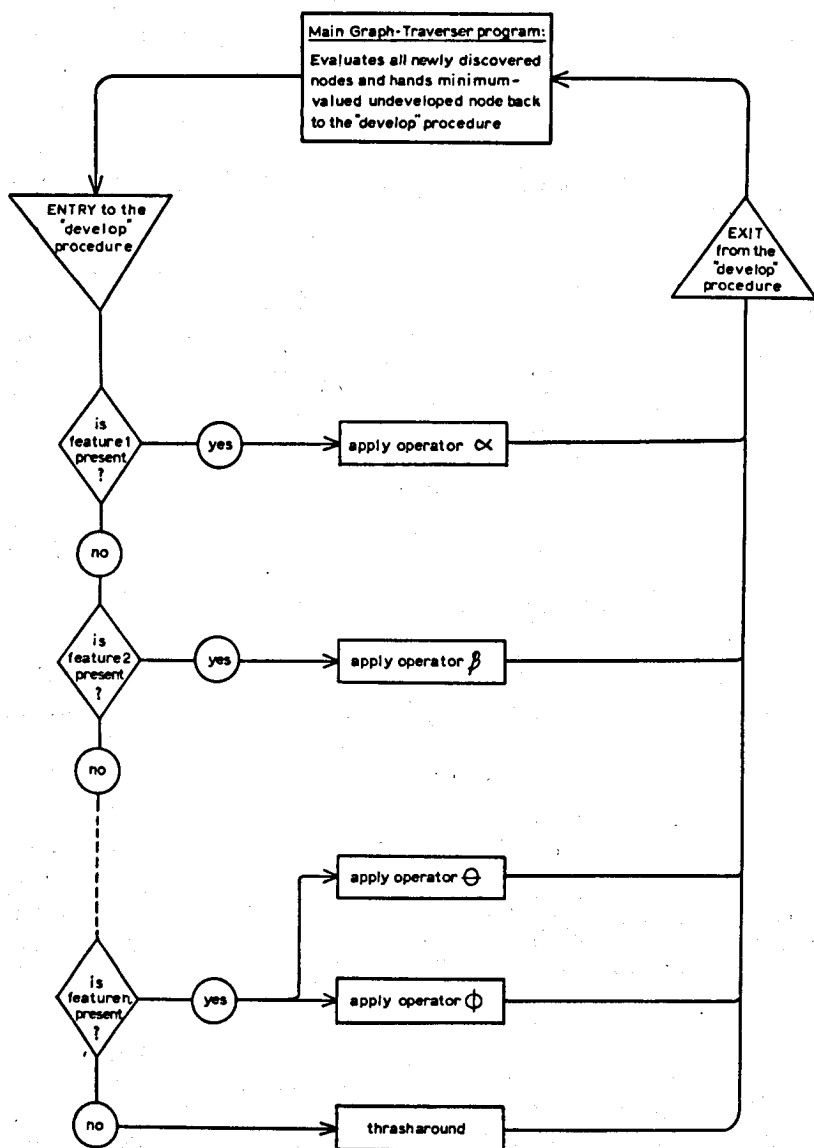


FIG. 6. Flow chart to show how a conditional-choice strategy can be embedded in the 'develop' procedure of the Graph Traverser.

it is not obvious whether it is better to aim at adding to the sequence 8, 1 (in which case 'W.X.Y.' is the operator of choice) or the sequence 4, 5 (in which case 'B' would be preferred). A conditional-choice strategy must necessarily specify one or other alternative, and we call such a forced choice a 'dogma'. It is here that the Graph Traverser can step in. What is required is to relax the operation of the 'develop' procedure *at this point* so that *both* the alternative descendant nodes are handed out to the main program. Evaluation now takes up its role of arbitration, and the two parallel lines of search, and any further parallels which may arise in this way, are pushed ahead according to their respective capacities to generate promising-looking further descendants.

THE 'THRASHAROUND' PROCEDURE

Situations *may* arise in which the author of a special-purpose algorithm has incorporated a *completely* arbitrary choice, through failure to recognise any features at all of the problem state which he knows how to translate into a rational prescription. In such a case the conditional-choice type of strategy is forced to exercise an extreme of irrational dogmatism. If, however, we are running the strategy as part of the develop procedure of the Graph Traverser, the dogma can be relaxed in a way which I shall describe. The idea is to provide a procedure, which I shall term 'thrasharound', to be called at such a juncture. 'Thrasharound' merely applies *all* compound moves up to a given level of compounding. In the case of the Eight-puzzle a reasonable set of 48 operators might be as shown in Table 1. In Fig. 7 the flow chart of Fig. 5 is shown after modifying it to accommodate the treatment of dogmas.

TABLE 1

A set of 48 operators for the Eight-puzzle (see text)

<i>Category of compound move</i>	<i>Number in the given category</i>		<i>Effect of example move</i>
Park, then circulate 2, then depark	4 clockwise	4 anti-clockwise	
Park, then circulate 4, then depark	4 clockwise	4 anti-clockwise	
Park, then circulate 6, then depark	4 clockwise	4 anti-clockwise	
Park, then circulate 2, then depark, then (at a remove of 2) park, then circulate 2, then depark	4 clockwise	4 anti-clockwise	
Park, then circulate 2, then depark, then park (from starting square), then circulate 2, then depark	4 clockwise →anti-clockwise	4 anti-clockwise →clockwise	
Park, then circulate 2, then depark, then park (from opposite square), then circulate 2, then depark	4 clockwise →anti-clockwise	4 anti-clockwise →clockwise	
TOTAL	24	24	(inverse of previous case)

MACHINE LEARNING AND HEURISTIC PROGRAMMING

Note that 'thrasharound' acts as a safety barrier to prevent the program exiting from the develop procedure without having generated any descendant nodes.

The effect of relaxing the dogmas of the conditional choice strategy and incorporating 'thrasharound' is shown in Table 2, which gives results on a randomly constructed set of 50 puzzles. The performance of a trained human

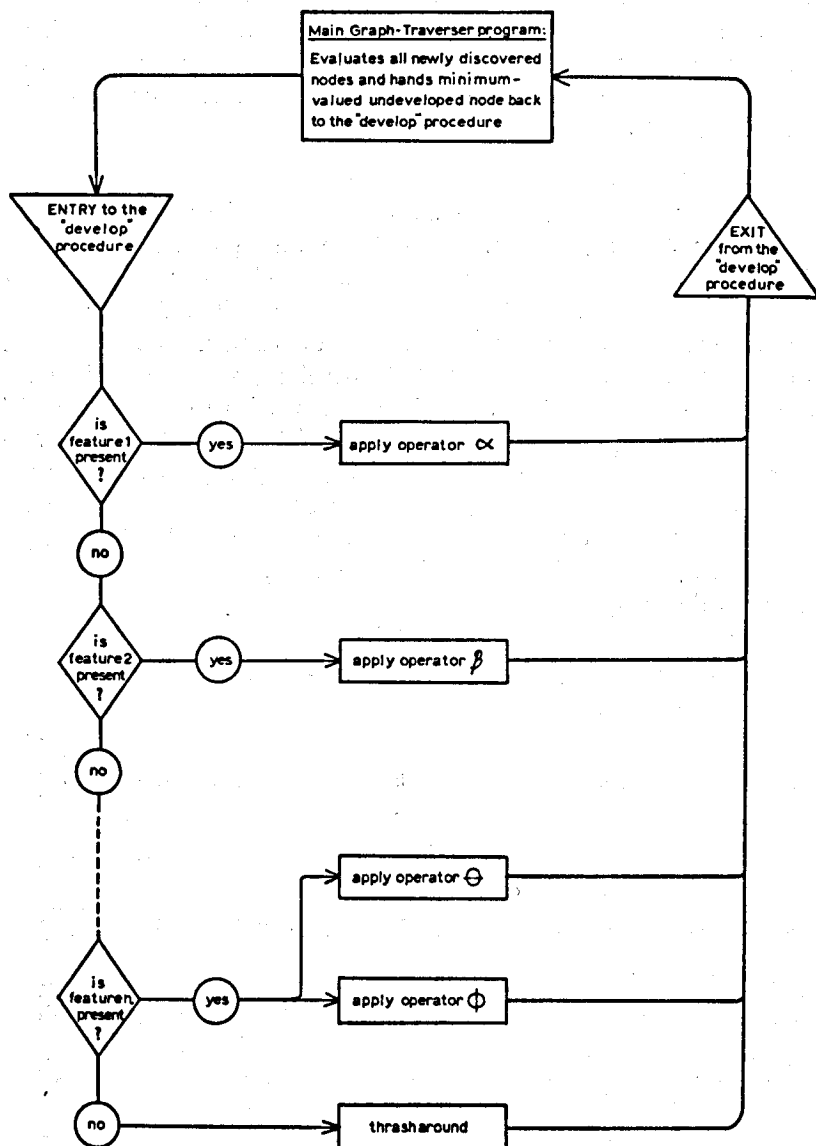


FIG. 7. A form of the develop procedure embodying a conditional-choice strategy, showing a relaxation of selection at a 'dogma' (test for feature n) and a call of 'thrasharound' when all tests fail.

subject is included for comparison. The second line of the table shows a 16 per cent saving in the number of moves, but this is clearly bought in terms of rather a large additional search effort, measuring this as the number of additional nodes which the program has to inspect. So we must prune the operator set, preferably in such a way as to cut away only dead wood, leaving untouched the potentially useful operators. In the present case this was done by prohibiting certain categories, specifically any manoeuvre which (i) breaks the main chain(s) or (ii) creates a 'scrambled edge' feature. The latter refers to the occurrence along an edge of the puzzle of precisely the three pieces which should occupy that edge, but in some wrong order, as in the following

2	7	4
1	0	5
6	8	3

The results of running the program with this option are shown in the third line of Table 2. The reduction of search effort is here accompanied by no significant change in the quality of the solutions obtained. The reason is that one of the pruning rules (elimination of scrambled sides) proved to be a positive benefit in barring certain lines of search, and roughly compensated for the losses incurred by the antisequence-breaking rule.

To summarise this small-scale exploration of feasibility, the use of the program to improve a conditional-choice strategy is clearly possible. Whether in a given practical application it would be worth-while will of course depend on the cost of the search process (measured here by the number of inspections) relative to the cash value of improvements in the solutions obtained.

TABLE 2

<i>Solver</i>	<i>mean minimal path</i>	<i>mean no. of moves</i>	<i>mean no. of nodes inspected during search</i>
Conditional choice strategy	21.5	34.6	4.7
Dogmas relaxed, with thrasharound	21.5	29.0	38.7
As above, operator set pruned	21.5	29.2	17.7
Trained human	21.5	30.8	?

Results with a randomly chosen set of 50 Eight-puzzles. The human solver was the best of 20 University students and had had three weeks' intensive practice (his mean no. of moves before practice was about $2 \times$ minimal path). The evaluation function was $\sum h_i p_i^2 + 100R$ (see Doran & Michie 1966).

DESIGNING A STRATEGY FROM THE GROUND UP

I shall now turn to the question: can the Graph Traverser be used to build a strategy 'from the bottom up', i.e., in a case where no 'conditional-choice' ideas, not even dogmas, are forthcoming from human sources?

Here we assume that the develop procedure has been given a rule-book containing (a) the elementary moves and (b) the means of enlarging this repertoire by forming compound moves to a certain depth. Unless this depth is very small, the operator set generated in this way will tend to be extremely large, leading to gross ramification of the search tree. Everything then turns on the provision of automatic pruning methods. Can the program be so constructed that it will gradually and systematically learn to prune its forest of operators? If so, could the process run to its logical conclusion, eventually leaving only one operator for each problem state? The program's action would in that case have achieved the design, without human aid, of a complete conditional-choice strategy, starting from no strategy at all. Let us consider the lines along which automatic selection methods might operate. In doing so we shall revisit some of the ground covered in the previous sections, but we shall be moving in the opposite direction: that is, instead of starting from a complete conditional-choice strategy and relaxing in the direction of Graph Traversal we shall be starting from pure Graph Traversal and moving in the direction of a conditional-choice strategy.

SELECTION ACCORDING TO APPLICABILITY OF THE OPERATOR

Selection can be applied even if the operator set consists only of the elementary moves. In a rather trivial sense, selection of a certain kind *must* be applied, in order to restrict moves to those which are legal. In terms of the Graph Traverser, the 'develop' procedure must incorporate tests of applicability. Taking the Eight-puzzle or any similar sliding block puzzle as a simple illustration, we might define the following elementary moves:

- (1) fill the empty square from above;
- (2) fill the empty square from below;
- (3) fill the empty square from the left;
- (4) fill the empty square from the right;

but of course tests must be incorporated† which eliminate in advance a given move in the case where the empty square lies on the top, bottom, left or right edge respectively. Granted, then, that operator-selection is inevitably present in embryo, can we develop this embryo into something which goes beyond selection by applicability and selects in addition according to criteria of utility?

SELECTION BY PATH-POPULARITY

If the program had available to it some way of judging the relative frequency with which the different operators in its set gave 'good results' then the process of selection by partial development could be further strengthened by imposition of an *ordering* on the operators, so that the more

† Not out of logical necessity: illegal states of the board *could* be nipped off at a later stage by ensuring that 'evaluate' assigned high values to them, but this would be artificial and time-wasting.

promising ones are tried first. But how can the program discover whether operators are giving 'good results' when it is engaged on a long and so far unrewarded search in a large problem space?

We have already met with a difficulty of this kind in regard to attempts at optimising the parameters of the evaluation function. We then disposed of it by finding a measurable property of the stored search tree, the 'penetrance' (Doran & Michie 1966) which could be used to rate the quality of the search behaviour, even before the search had led to any solutions. The remedy prescribed in the present case represents an extension of essentially similar reasoning, namely:

- (1) a partial search with a high penetrance is to be judged a better result, other things being equal, than a partial search with low penetrance;
- (2) an operator which directs the search along what subsequently becomes a partial path is, other things being equal, more likely to be generally useful than one which shunts the search into a side-branch of the search tree, *especially* if the penetrance is high.

Taking the above two points together, we may conveniently estimate the *path-popularity* of an operator as

$$\frac{\text{No. of occurrences of operator in partial path}}{\text{No. of applications of operator to operands in partial path}}$$

where 'operand' denotes 'developed node' and operators are mapped on to arcs of the graph. If we use the penetrance as a weighting factor the above expression can be used to construct a cumulative figure of merit, averaged over a number of partial searches.

SELECTION BY CONDITIONAL PATH-POPULARITY

The next step is to try to relate an operator's probable usefulness to features of the given problem state. Given that we have a list of possibly relevant features (the most obvious candidates for such a list are perhaps the terms of the function used by the evaluate procedure) we can proceed as before, but this time computing the path-popularity *separately for each feature or group of features*. Thus the conditional path popularity of an operator *given the presence of feature x* can be estimated by the expressions given in the last section except that 'operand' is replaced by 'operand containing feature x'.

THE EMERGENCE OF A FULLY-DESIGNED STRATEGY

We have reached the point where we can envisage automatic use of the features of the operand state to impose an ordering on the members of the operator set. This ordering is performed according to the estimated conditional path-popularity of each operator. Suppose that for a given constellation of features the operator first in order had accumulated a path-popularity

score over-towering those of all its competitors (an 'outlier' in fact in the statistical sense (Dixon 1953)). The effect would be for this operator to be *almost invariably* used whenever a problem state answering to the given description cropped up, its competitors hardly ever being called upon. A natural simplification, with a concomitant saving of computing time, would be to jettison, for problem states of the given feature-description, all operators but the favourite. Now imagine this process spreading, until every state-description becomes paired with one favoured operator instead of with a set. At this point the Graph Traverser program would have secreted, as it were, a fully-formed conditional-choice strategy or algorithm, of the kind which human problem-solvers devise.

EVOLUTIONARY PLAN

To sum up the ideas involved in this scheme for growing the Graph Traverser 'from the ground up', I have set out in Table 3 the three stages listed at the outset of this paper, each further sub-divided. In addition I have added a 'stage zero' at the start in order to exhibit the bare framework which underlies the superstructure here outlined. The table should be read in conjunction with the notes which follow.

NOTES

- Stage 0 This is a method of exhaustive search, related to a well-known algorithm (Berge 1962), guaranteed to find the goal and to construct a minimal path, but at high cost.
- Stage 1(a) This is the form of the program described by Doran & Michie (1966).
- Stage 1(b) 'Penetrance' is here regarded as a function of n independent variables, these being parameters of the evaluation function. Hill-climbing methods to maximise the penetrance are thus directly applicable.
- Stage 2(a) In this mode the program is encumbered with a surfeit of operators, since 'all' descendants could quite easily run into hundreds or thousands.
- Stage 2(b) Less than half of the operator set is now, on average, applied to each operand state, and very much less than half if the evaluation function is good.
- Stages 2(c) The program's user supplies the priority list (2(c): Compare and 2(d) Popplestone, pp. 31-46) or the rule for computing a priority list as a function of the current problem-state (2(d)).
- Stage 3(a) 'Skill' in the problem environment, accumulated by optimising the parameters of evaluation, now begins to be siphoned over into the develop procedure.
- Stage 3(b) The priority list is derived and up-dated by the program rather than being 'given'. The logical conclusion of the siphoning

process is that state-evaluation becomes redundant, leaving the operator-selection apparatus in possession of the 'distilled essence' of the program's accumulated experience of the problem.

ACKNOWLEDGEMENTS

Most of the ideas in this paper arose in the course of discussions with my colleague J. E. Doran, who has also contributed to this book. I also wish to thank Miss Janet Nairn for able assistance with program development.

The work described was supported by the Science Research Council.

TABLE 3
Evolutionary plan for the Graph Traverser

DEVELOP PROCEDURE				
Stage	Rule-book	All descendants or one only	Order of selection from operator set	EVALUATE PROCEDURE
0	Elementary moves	All	Immaterial	Random function
1(a)	Elementary moves	All	Immaterial	Directional function, i.e., designed to correlate with distance from goal state Directional, hill-climbing (evolving by optimisation of the function's parameters)
1(b)	Elementary moves	All	Immaterial	
2(a)	Compound moves	All	Immaterial	Directional, hill-climbing
2(b)	Compound moves	One	Random	Directional, hill-climbing
2(c)	Compound moves	One	Fixed priority list	Directional, hill-climbing
2(d)	Compound moves	One	Conditional priority list	Directional, hill-climbing
3(a)	Compound moves	One	Path-popularity	Directional, hill-climbing
3(b)	Compound moves	One	Conditional path-popularity	Directional, hill-climbing, but ultimately redundant

REFERENCES

- Berge, C. (1962). *The theory of graphs*, pp. 67-68. Doig, A. Trs. London: Methuen.
- Dixon, W. J. (1953). Processing data for outliers. *Biometrics*, 9, 74-89.
- Doran, J., & Michie, D. (1966). Experiments with the Graph Traverser program. *Proc. R. Soc., (A)*, 294, 235-259.

MACHINE LEARNING AND HEURISTIC PROGRAMMING

- Johnson, W. W., & Storey, W. E. (1879). Notes on the '15' puzzle. *Amer. J. Math.*, **2**, 397-404.
- Maynard Smith, J., & Michie, D. (1961). Machines that play games. *New Scient.*, **12**, 367-369.
- Michie, D. (1966). Game-playing and game-learning automata. *Advances in programming and non-numerical applications to computation*, ch. 8, Fox, L., Ed. London: Pergamon.
- Newell, A., Shaw, J. C., & Simon, H. A. (1960). A variety of intelligent learning in a general problem solver. *Self-organising systems*, pp. 153-189. Yovits, M. C., & Cameron, S., Eds. Oxford: Pergamon.
- Tait, P. G. (1880). Note on the theory of the '15' puzzle. *Proc. R. Soc. Edinb.*, **10**, 664-665.

COGNITIVE PROCESSES: METHODS AND MODELS



NETWORKS AS MODELS OF WORD STORAGE

G. R. KISS

BIRKBECK COLLEGE
UNIVERSITY OF LONDON

This paper is an attempt at the formulation of the processes which could account for the behaviour of human beings when they use meaningful words to communicate with each other. The two main aspects of this problem are the *selection* and *recognition* of words. This paper will suggest a possible model of these processes under certain restricted conditions. Ultimately the word storage problem will have to be put into its proper perspective in the setting of continuous language behaviour: the generation and reception of connected human discourse. It is needless to emphasise that we are far from achieving this aim. Correspondingly, the research reported here deals with the simplest kinds of situations in which word meanings play a significant role.

These simple situations have been much studied in a field of psychology which is now generally called verbal learning. In particular, the basic, free word association experiment could be regarded as perhaps the simplest paradigm of a situation in which most of the important semantic and pragmatic aspects of word handling are involved. One of the later sections will give a description of the word association process in terms of the model suggested.

The main motivation behind the research reported here is psychological. Although the model to be described may lead to mechanisms that may prove useful in constructing intelligent machines, this is accidental. The main aim has been to deepen our understanding of how human beings understand words. In a sense, this is a more stringent requirement than the construction of a machine which could understand words. For it is not sufficient to build a mechanism which can fulfil a certain function, but it is necessary to enquire whether that mechanism is similar to the one employed by humans.

To that extent, where artificial intelligence ends, the simulation of behaviour begins. The achievement of a working model is only the starting point.

VERBAL LEARNING AND VERBAL BEHAVIOUR

It would be out of place here to attempt a general survey of this field on which there is a vast and scattered literature. We shall confine ourselves to mentioning a sample of interesting findings in order to illustrate the nature of the phenomena to which the model is assumed to be relevant.

Mediated transfer of learning is the name given to a class of phenomena in which the effects of existing associative connections on the transfer of associative learning manifest themselves. Jenkins (1963) reviews a large number of experiments of this kind. To give a simple example, suppose a nonsense syllable, DAX, is learned as a paired associate to the word WAR. We know from word association frequency data collected from large samples of subjects that a frequent association to WAR is PEACE, and that in turn PEACE elicits JUSTICE with a high frequency. It is also known that WAR does not elicit JUSTICE. If now DAX is learned as a paired associate with JUSTICE, we find that there is a considerable saving compared with a control group of subjects who did not learn the DAX-WAR pair. Apparently the existing associative chain, WAR-PEACE-JUSTICE, transfers the effects of learning through the mediating word, PEACE. A criticism which can be directed towards most of these experiments is that in selecting specific associative chains for experimentation they fail to recognise the fact that these chains are embedded in a *network* of associative connections, so that many alternative paths may exist, possibly working against the desired experimental effect.

A number of studies have been done by Razran (1939) and others on semantic generalisation. Semantic generalisation refers to the phenomenon that a conditioned response (e.g., salivation) will generalise from one word to other words which have a similar meaning.

Interesting clustering phenomena have been discovered by Bousfield, Cohen & Whitmarsh (1958) in the free recall of lists composed of taxonomically related words. The words were presented in a randomised order, but during free recall it was found that the words belonging to the same taxonomic group (e.g., names of professions) tended to occur together. Similar clustering effects were found also with lists composed of associatively related words (Jenkins, Mink & Russell 1958). A related clustering effect was found by Bousfield & Sedgewick (1944) in the production of restricted associations, like naming as many quadruped animals as possible. They found that associations tend to occur in 'bursts' separated by longer time intervals, and that the clusters produced in this way consist of semantically related items. The curve showing the cumulative number of responses against time is well approximated by an exponential function.

The influence of associative connections between words of a list presented for free recall on the number of items recalled has been studied by Deese

(1959). It was found that the greater the associative cohesion of a list, the better the recall performance. The intrusion of words not on the original list was also related to cohesion.

WORD ASSOCIATIONS

The basic word association experiment consists of the presentation of a stimulus word to which the subject has to respond with the first word that comes to his mind. There are many variations on this basic theme. The responses can be restricted to certain classes of words. Several responses in a series may be requested instead of a single word. Several words may be presented instead of a single stimulus. The method of presentation can be auditory or visual, the response oral or written, and so on.

When the experiment is repeated with a sample of subjects a striking degree of agreement is observed as to the most frequent response words. The frequency distribution is usually of a very steep, highly skewed form. An example, taken from Russell & Jenkins (1954), is shown in Table 1. Tables like this, which are now widely collected from extensive samples of subjects, are obtained by presenting subjects with lists of stimulus words and asking them to respond to each word with the first word that comes to their mind.

The form of this frequency distribution has been shown by Skinner (1937) to conform to Zipf's law which states that if the frequencies of words in a sample are plotted on logarithmic scales against the ranks of the words, a straight line results. Howes (1957) has also shown that the sums of the associative frequencies taken over a hundred different stimulus words correlate highly with the frequencies of those words in general usage. These results seem to indicate that the word association experiment is a valid approach to a study of general language behaviour, at least in these overall statistical aspects.

TABLE 1

STIMULUS: Butterfly

RESPONSES:

144 moth	23 cocoon
117 insect	22 summer
104 wing(s)	21 colour(s)
84 bird	20 monarch, bee(s)
78 fly	14 stomach
62 yellow	9 worm, beauty
42 net	7 flying
34 pretty	6 animal, beautiful
33 flower(s)	5 blue
27 bug	4 flight, caterpillar, Madam, opera
3 ant, spring, butter, catch, dance, flutter	
2 light, warp, jet plane, sky, soft, mouse, colourful, song, dragonfly, Lepidoptera, dog, orange, spots, warm, lagoon, sunshine	
1 roach, tree, feather, delicate, thing, grace, car, canary, butterfly's, fragile, flap, swim, curtain, maggot, girl, air, morn, hobby, smooth, felt, hornet, dainty, robin, grass, mosquito, buttercup, silk, Orthoptera, wheat, game, collection, field, louse, tiger, California, blind, cow, chase, watch, Cecropia, outside, woods, dust, Spanish fly, freedom, nothing, nature, flea, sun, zoology, garden, Japan, look, dandelion, crazy	

COGNITIVE PROCESSES: METHODS AND MODELS

Another stable relationship which has been observed by several investigators is expressed by Marbe's law. It states that the time taken by subjects to produce an association is a logarithmic function of the associative frequency of the response word.

Semantic satiation is another phenomenon which is of considerable interest for any theory of connotative meaning. When a word is repeated or inspected for a prolonged period of time, the subject usually reports that the word loses or changes its meaning. Associative satiation refers to the fact that the satiated word will elicit a larger number of irrelevant and rare associations.

ELEMENTS OF AN INFORMATION PROCESSING THEORY FOR WORD STORAGE

The aim of this theory is to specify a set of hypothetical processes and structures which are capable of accounting for the kinds of behaviour described in the previous sections. The method used for the formulation of this theory is the construction of an information processing model which can be programmed for a digital computer. Then its behaviour will be studied and compared with human behaviour, leading to a successive refinement and elaboration of the model.

A novel feature of this model is that it is designed with a view to on-line computer-controlled experimentation in which the model will interact directly with human beings. This will serve two purposes. First, the machine can be used as an automated data collector. Second, psychological experiments can be set up in which the subject is put into a dynamic situation where the changes are made dependent on the subject's previous behaviour. These two points will be further discussed later.

THE POSTULATES

1. Human beings have two qualitatively different resources for information processing: (a) serial (a central processor), (b) parallel (a network of interconnected elements).
2. The serial and parallel processors interact. The serial processor can create, eliminate and activate the elements of the network. It can 'read' the current status of these elements, and it can create or eliminate the links between them. The central processor can gain access to information stored or associated with these elements.
3. The structure and functions of the serial processor closely resemble the central processors of digital computers. It has a limited-capacity fast access memory. It is sequential in operation and can do only one thing at a time. It is capable of executing a set of elementary operations and programs composed out of such operations. This postulate reflects the properties of higher levels of human consciousness and attention.
4. The parallel processor consists of a set of elements which are interconnected by a network of transmission links. The elements are internal representations of words. They consist of a set of storage cells which contain

all the information pertaining to an element. This information includes the following items: (a) excitation level (long-term and short term), (b) a set of pointers to a relational store (see below), (c) a set of pointers to other elements of the network.

The transmission links are physical linkages having a certain capacity for transmitting excitation to other elements. This transmittance value is variable.

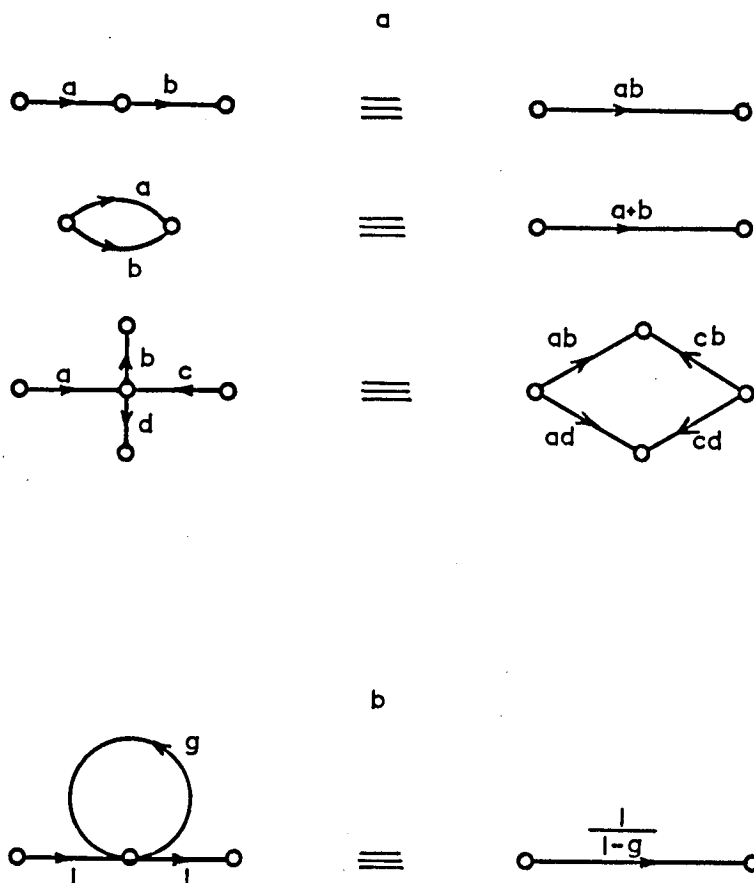


FIG. 1.

5. The spread of excitation along the transmission links is governed by the rules of signal flow graphs. Following the conventions first described by Mason (1953), it is assumed that the elements of the network are 'ideal' repeater stations which sum the incoming excitations and transmit the resulting signal along the outgoing links. In general a signal flow graph is a directed graph which is a graphical representation of a set of linear algebraic equations for the variables represented by the nodes of the graph. From this concept a basic set of transformation rules can be derived. These are shown in Fig. 1(a).

These elementary transformations, together with the elimination of a self-loop, shown in Fig. 1(b), can be used for the reduction of a flow graph to a form in which only the variables of interest are retained.

This postulate reflects our current ignorance of the nature of the interaction between the elements of the parallel processor. Since this theory is not primarily concerned with neurophysiological mechanisms, it does not seem necessary to assume that the elements of the parallel processor have the properties of neurons. However, for the convenience of digital simulation, the linear elements assumed by this postulate may have to be replaced by neuron-like elements.

6. There is a long-term relational store of images and concepts. The contents of this store represent the knowledge of the organism about the environment. It is called relational, because in contrast to the parallel processor it contains information on the various logical and other relationships between items. The parallel processor is an 'indexing structure', or thesaurus, for this relational store, and it is concerned with only one kind of relationship, that of the *relevance* of one element to another. The nature of the relational store is not discussed in detail in this paper, but we can point to Raphael's (1964) information retrieval system to indicate the kind of organisation we have in mind.

7. Word recognition. The parallel processor resolves the connotative meaning of a word into an excitation pattern produced over the network by the activation of the corresponding element. The recognition of connotative meaning implies the recognition of this excitation pattern. This pattern is a representation of aspects of the word's meaning. To the extent that clusters related to concepts and qualities exist in the network, excitation levels in those clusters reflect the relevance of these concepts to the activated word.

The network of elements thus acts as a transformation, mapping words into a semantic space. The central processor can obtain the coordinates of a point in this space by reading off the excitation levels of a set of selected elements in the vicinity of the activated element. The selection of the coordinate system can therefore be made dependent on the 'point of view' from which the word meaning is regarded. Since the structure of the network is likely to be hierarchical (corresponding to conceptual organisation), the description of the pattern which can be obtained in this way can represent meaning to a varying degree of detail, and can bring into focus some facets of meaning at the expense of others.

It now becomes possible to compare meanings and define degrees of similarities in meaning, or synonymy, by comparing patterns of excitation.

8. Word selection. This involves a search process executed by the central processor over the elements of the network. The specification of the desired word, either by its properties or by the activation of stimulus words, will produce a region of high excitation in the network. The search is confined to the region where the excitation level is above some threshold value. High levels of excitation indicate high relevance. In general the search

proceeds from high excitation levels to lower ones. This implies that in choosing among the alternatives at any given node, the path leading to the highest excitation is taken first.

Various search strategies are conceivable, and it is assumed that this is a characteristic of the individual. For example, in the word association experiment a 'width first' strategy would at first generate the elements connected to the stimulus node, then the elements connected to these 'first generation' elements, and so on. On the other hand, a 'depth first' strategy would follow any given chain of links to its limit, then the next such path would be selected, and so on. Clearly, these different strategies can be expected to have different behavioural consequences. A depth first strategy may be expected to produce more distant associates if the time allowed for giving an associate is limited.

The elements generated during this search are tested for acceptability. The nature of these tests will vary with the nature of the task. For example, in the word association situation it is known that coordinate words (of similar meaning) and contrasting words (opposites) are the most frequent responses. One test therefore consists of determining the similarity between the word generated and the stimulus word. Similars would be defined by high relevance and high similarity, while opposites would have high relevance and low similarity.

Other acceptability tests will be concerned with the determination of the relational operators connecting the stimulus element to the element under testing. These tests are based on the relational store. The nature of the relational operator which is acceptable to the subject will again depend on the interpretation of the task by the subject (Moran, Mefferd & Kimble 1964).

IMPLEMENTATION OF THE MODEL AS A COMPUTER PROGRAM

A computer program, WORSE (WORD SElection), is at present under construction, which will incorporate the Postulates outlined in the previous section. The main aim in writing this program is not the reproduction of word association norms, or associative responses for individual subjects. It is intended rather as a work-bench for testing various hypotheses in relation to the word selection problem, and also to experiment with direct man-machine interaction in using words.

The program is written in Information Processing Language 5, which has been implemented for the KDF-9 computer.

In WORSE each word is represented by a list structure. It contains all information that is pertinent to that word, such as its current excitation level, the names of the other words to which it is linked, the transmittance values of these links, the external 'printname' of the word, and so on.

WORSE can acquire word association network data in two ways. It can either read from paper tape a suitably prepared set of data representing a

network derived from a sample of subjects, or it can build up such a network through on-line interaction with an individual subject.

The flow graph reduction part of the program is at the moment implemented in the form of a separate ALGOL program along the lines indicated by Robichaud, Boisvert & Robert (1962). This program iterates over all nodes of a network, first eliminating self-loops and then removing all outgoing branches, thereby reducing the graph to sources and sinks.

In terms of a word association network, a set of stimulus words is designated, and the program derives the transmittance values leading from any stimulus word to all other stimulus words and to a set of response words.

When WORSE is run with network data obtained from a sample of subjects by means of the word association experiment, these transmittance values are input to the program. Since IPL-5 is rather inefficient for numerical calculations, it will probably not be possible to include in WORSE a complete flow graph solving routine, unless it is programmed in machine code as a special 'J-process'. However, some approximation to a complete reduction may be possible. Instead of taking into account all possible paths in the graph, the reduction process could be limited to paths up to a certain maximum length.

The program contains a process which represents the 'activation' of a word by the central processor. When this happens, the excitation level of the word is increased by one, and all other words within a specified topological distance from it are also increased by a value proportional to the transmittance of the path leading to them from the activated word.

The pattern recognition part of the program is not yet implemented, nor is the search process for a word satisfying some specific conditions.

SOME SIMPLE ON-LINE EXPERIMENTS

Considerable interest is attached to the use of WORSE as a man-machine experimental system. In the usual word association or verbal learning experiments the situation is always fixed in advance, and no account can be taken of a subject's individual characteristics or actual behaviour during an experimental session. Messick & Radoport (1964) have recently summarised the potentialities of computer-controlled experimentation in psychology and pointed out the great flexibility which can be obtained in experimental set-up, the economies of data processing during and immediately after the experiment, the improved standardisation of experimental conditions and the unlimited complexity in making the experimental situation a dynamic one.

In order to explore the feasibility of using IPL-5 on the KDF-9 computer for this kind of work, two 'J-processes' have been added to the system to enable the reading and printing of character strings through the on-line Flexowriter. Internally the character string is represented as an IPL list containing internal symbols which are the numerical representations of the characters.

The man-machine system part of WORSE contains an experimenter executive routine which iteratively presents a stimulus to the subject, identifies the response (in terms of the words which have already occurred during the experiment), processes the response, and then determines and presents the next stimulus. The routine is written in a general form so that different strategies of determining the next stimulus can easily be set up.

Strategy I, which is currently running on the computer, investigates the relationship between the excitation level of a word and the probability that it will be chosen as a response. The technique used has been taken from Baldridge & Hustmyer (1965). The subject is initially presented with a list of 15 words for free association. Then the 15 responses are given back to the subject as stimuli, and the same process is repeated over several cycles.

It may be thought at first that this procedure will generate 15 stimulus-response chains which will have little to do with each other. This turns out not to be the case. Even when the original 15 words are unrelated, the associative responses will soon converge on to a fairly small set of words and form a connected graph. The rate at which new words are introduced is at first high and then decreases and approaches an asymptotic value. Strategy I monitors the ratio of the number of new words to the total number of responses, and when a limiting value is reached it switches the subject to a new set of words.

In the experiments run so far the first set of 15 words are closely inter-related owing to the fact that they are all high-frequency associates to BUTTERFLY. The members of this list are: BEES, BUG, YELLOW, FLY, MOTH, NET, COLOUR, BUTTERFLY, COCOON, PRETTY, WING, INSECT, FLOWER, BIRD and BLUE. The members of the other list of 15 words are relatively unrelated words (the same list as the one in the experiments by Baldridge): ANIMAL, PACKAGE, TOWN, BOOK, TASK, SOUND, DREAM, PAPER, HOUSE, BOTTLE, PLANT, WORD, COLOUR, PICTURE and LETTER. The word COLOUR is repeated in the second list.

The behaviour of one of the subjects is shown in Fig. 2. The underlined words are members of the two starting lists of stimuli. The numbers show their order of introduction. The arrows represent a stimulus-response connection.

Although no quantitative evaluation of this data has been attempted so far, the qualitative picture is quite clear. There is a strong tendency to repeat words which occurred previously in the experiment. The path taken by the stimulus-response chains seems to depend on the bias created over the network by the previous words. Consider for example the chain PRETTY-SOUR-OINTMENT-FLY. The occurrence of OINTMENT is probably due to the previous occurrences of words like BEES, BUG, WASP, HORNET and FLY. If the experiment is carried on long enough, the subject finds it difficult to think of new words and gets trapped into endlessly repeating cycles.

The introduction of new words against time is shown in Fig. 3 for the

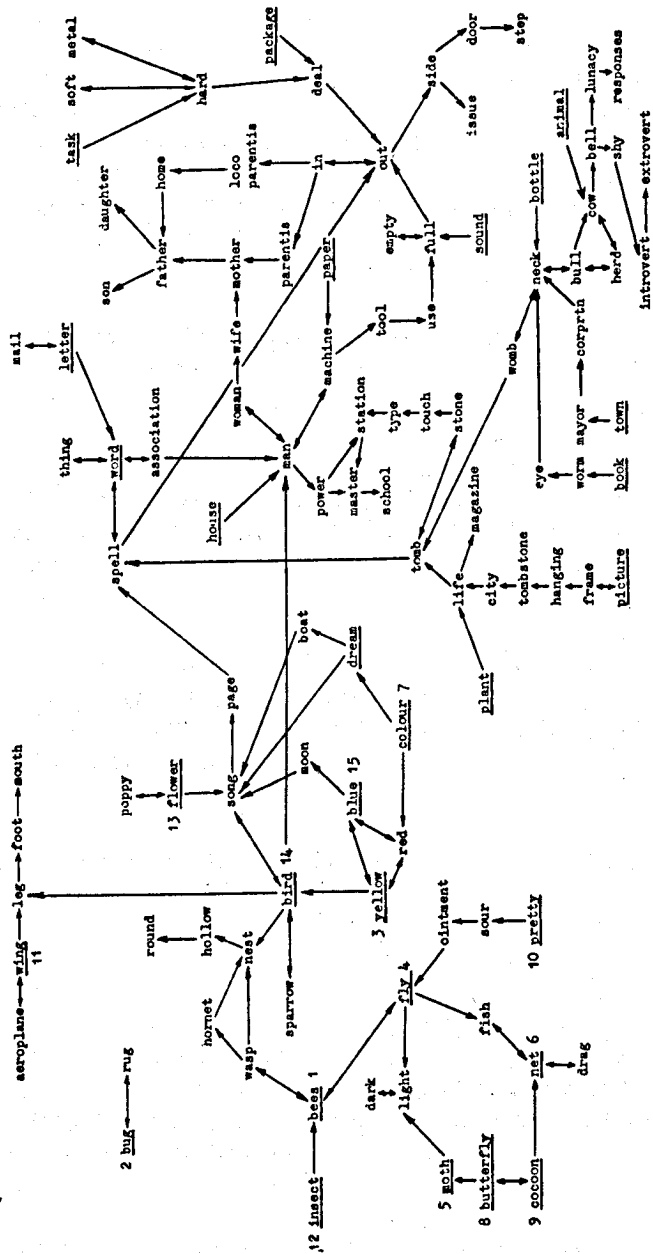


FIG. 2.

same subject. The limiting ratio of the number of different words to the total number of words was set in this experiment to 0.55. For the first group of words (interrelated items) the subject took 87 trials to reach this limit, while 151 trials were required in the case of the unrelated group. The break point A on this curve indicates the switch to the second (unrelated) group of words. At point B, during the second half of the experiment, the subject stopped and asked 'Am I supposed to be creative? I seem to repeat the same words too much.' No information was forthcoming from the experimenter, who referred the subject to the original instructions ('always give the first word which occurs to you . . .'). Nevertheless, there is clearly a shift in the subject's task-set here.

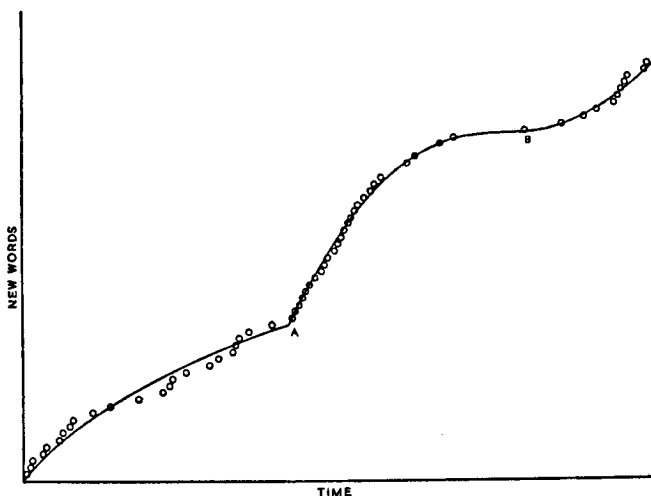


FIG. 3.

All five subjects used so far reported that they enjoyed the experience of working with the machine. Although this experiment is a fairly simple one, it shows the feasibility of using the machine in this way. For more complex strategies it will probably be necessary to rewrite the more time-consuming parts of the program in machine code. With Strategy I, the program takes 3-5 seconds to produce the next stimulus after the subject's response. Even without resorting to machine code, it would be possible to improve on this by more sophisticated search procedures in the word-identification routine.

SUMMARY

The elements of an information processing theory of word storage in humans were outlined. Eight postulates were formulated: (1) There are serial and parallel information processing resources in humans. (2) These interact with each other. (3) The serial processor is a limited-capacity sequential mechanism. (4) The parallel processor consists of a network of interconnected elements. (5) The links connecting the elements transmit

excitation according to a flow graph model. (6) There is a relational store of images and concepts indexed by the parallel processor. (7) Word recognition implies the recognition of the pattern of excitation produced by the activation of an element. (8) Word finding involves a search process over the elements of the network.

The progress with the implementation of this theory in the form of a computer program was reported and some simple on-line computer-controlled experiments were described.

ACKNOWLEDGEMENTS

In addition to the sources mentioned in the paper, I should mention that I have been much influenced by current developments in information retrieval, especially Giuliano (1963), and by Bousfield's (generally) and Deese's (1962) theorising in connection with word associations. Although working in a different context, Broadbent (1965) and Morton (1964) are thinking about word storage along similar lines.

I had many stimulating discussions with Phil Levy at Birmingham and with Rod Burstall of the Experimental Programming Unit at Edinburgh.

The work reported in this paper has been supported by a grant from D.S.I.R.

REFERENCES

- Baldrige, B. J., & Hustmyer, F. H. (1965). Iterative Association. *Percept. Mot. Skills.*, 20, 617-627.
- Bousfield, W. A., Cohen, B. H., & Whitmarsh, G. A. (1958). Associative clustering in the recall of words of different taxonomic frequencies of occurrence. *Psychol. Rep.*, 4, 39-44.
- Bousfield, W. A., & Sedgewick, C. H. (1944). An analysis of sequences of restricted associative responses. *J. gen. Psychol.*, 30, 149-165.
- Broadbent, D. (1965). Perceptual defence and the engineering psychologist. *Bull. Brit. Psych. Soc.*, 18, 1-15.
- Deese, J. (1959). The influence of inter-item associative strength upon immediate free recall. *Psychol. Rep.*, 5, 305-312.
- Deese, J. (1962). On the structure of associative meaning. *Psychol. Rev.*, 69, 161-175.
- Giuliano, V. E. (1963). Analog networks for word association. *Instn. elect. Engrs. Trans. Military Electronics*, MIL-7, 221-234.
- Howes, D. (1957). On the relation between the probability of a word as an association and in general linguistic usage. *J. abnorm. soc. Psychol.*, 54, 75-85.
- Jenkins, J. J. (1963). Mediated associations: paradigms and situations. *Verbal Behaviour and Verbal Learning*. New York: McGraw Hill.
- Jenkins, J. J., Mink, W. D., & Russell, W. A. (1958). Associative clustering as a function of verbal association strength. *Psychol. Rep.*, 4, 127-136.
- Mason, S. J. (1953). Feedback theory—some properties of signal flow graphs. *Proc. Inst. Radio Engrs.*, 41, 1144-1156.

- Messick, D. M., & Rapoport, A. (1964). Computer-controlled experiments in psychology. *Behav. Sci.*, **9**, 378-382.
- Moran, L. J., Mefferd, R. B., & Kimble, J. P. (1964). Idiodynamic sets in word association. *Psychological Monographs*, **78**, No. 2.
- Morton, J. (1964). A preliminary model for language behaviour. *Internat. Audiology*, **3**, 1-10.
- Raphael, B. (1964). SIR, A computer program for semantic information retrieval. *Project MAC, Report*, MAC-TR-2. Boston: M.I.T.
- Razran, G. H. S. (1939). A quantitative study of meaning by a conditioned salivary technique (semantic conditioning). *Science*, **90**, 89-90.
- Robichaud, L. P. A., Boisvert, M., & Robert, J. (1962). *Signal Flow Graphs and Applications*. London: Prentice-Hall.
- Russell, W. A., & Jenkins, J. J. (1954). The complete Minnesota norms for responses to 100 words from the Kent-Rosanoff Word Association Test. *Tech. Rep.* No. 11, N8-ONR-66216, Office of Naval Res. and Univ. of Minnesota.
- Skinner, B. F. (1937). The distribution of associated words. *Psychol. Rec.*, **1**, 71-76.

WILL SEEING MACHINES HAVE ILLUSIONS?

R. L. GREGORY

PSYCHOLOGICAL LABORATORY
UNIVERSITY OF CAMBRIDGE

The ability of the higher animals to accept and interpret information from distant objects confers enormous advantages for creatures (or machines) which respond only to immediate stimulation and have no opportunity to anticipate the future. Distance receptors, especially the eyes, serve as early warning systems by giving information of distance events, making it possible to gauge the probable future. The classical biological notion of stimulus-response applies to creatures limited to touch information. The development of distance-receptors evidently allowed brains to develop to give strategic behaviour. It is unfortunate that the early, now classical, studies of reflexes involving touch and the internal regulation of the body have been so largely taken over to describe brain function, for these concepts are inadequate for describing the central nervous system. They tell nothing about how brains handle information from the eyes, to allow animals and man to see. They tell us nothing about decision-making: how present experience is related to the stored past to predict the immediate future. Reflexes imply an essentially passive system—a system which would be quiescent in the absence of sensory inputs—but this is far from the case with the higher living systems.

Biological information processing is almost incredibly efficient. The general slowness of animal movements belies the rate of perceptual decision-taking, which is even more remarkable when we consider that sensory information is transmitted with pulse rates always under a kilocycle and generally but a few hundred cycles maximum. An important reason for the speed and efficiency of the visual system is its very large number of parallel channels—at least 10^6 for the human eye. Sequential processing is inevitably slow (which handicaps the ear) but the eye, though limited by the limited band-width of its channels, is faster than any man-made computer, though these have electronic components several orders more rapid in operation.

The brain interprets two-dimensional images in the eyes as representing objects distant in space. The more one considers what is involved, the more astonished one becomes at the ability of nervous systems to recognise objects and judge distance and speed in the ever-changing external world. Simultaneous parallel transmission of information is however but the beginning of the story: just as important are various 'strategy' short cuts in processing. These short cuts are not always successful, and can produce serious errors. Some visual errors are of little interest, but others are worth studying in detail for they are clues to the way the visual system functions. Some errors are due to *limitations of the components* of the nervous system—for example fatigue or adaptation of cells with prolonged use. Others are due to the *logical difficulties of the problems* to be solved in seeing the world. These logical difficulties are of general interest, for we should expect to encounter them in *any* system, including man-made seeing machines, set to solve the problem.

THE PROBLEMS FOR SEEING MACHINES

Biological seeing machines handle spatial patterns which must be classified, stored and compared. Some kind of filing system with ready access is required, and in human perception the perceptual 'filing cards' contain a wealth of information going far beyond purely sensory characteristics. To say, 'I see a table', is to go beyond, 'I see a brown rectangular shape, with square vertical protrusions extended below it . . .'. To see a table is to know that it has an underneath; that the 'square protrusions'—the legs—support it, and are necessary for its stability. It is to know that it will not bark, or explode, that it may be used for resting things on, that it is easily scratched, heavy to lift and that one may hurt one's knee on hidden edges. When we see a table, we 'see' a host of non-visual things—past and future, as well as present. We 'see' that it was man-made, and that it will not 'softly and suddenly vanish away . . .'.

The point is that to *see* a table is far more than to have a *picture* of a table. A picture is but a part copy of the original, and is nothing until interpreted.

A seeing machine is far more than a photographic or television camera. It translates optical information into appropriate action (be this limited to linguistic description) and this involves assessing the significance of the pictures in the eye.

Looking at a *picture* of a table is a most curious business. The visual input may even be identical to the original—truly excellent stereoscopic coloured cinema—and yet the situation is not equivalent to that of seeing a real table. Though the visual inputs are the same, the information on the internal 'file cards' for 'object' and for 'picture of object' are quite different. We cannot *use* this picture table. We cannot put things on it, or hurt our knee by banging it. Even a perfect picture is perceptually very different from the object itself, because a picture has quite different, if any, causal properties. And yet the retinal image is only a picture.

GREGORY

To identify an object a perceptual system equipped with filing cards has merely to recognise the *minimum* defining characteristics. It may use probabilities; it may 'consider' first the most likely alternatives, so that there is no need to hunt through the whole, or a large part, of the total file. But this approach to saving computation time must lead sometimes to misrecognition of improbable objects. It may also give spurious addition of characteristics which 'ought' to be present. It is said that people shown aircraft for the first time 'see' feathers on their wings. Feathers 'ought' to be present, and they are added from the data on the file card labelled 'wing', if all previous wings had feathers.

In studying the rare case of a man who recovered sight (by corneal grafting) after being blind since infancy, I noticed many times that he would correctly identify objects previously known to him by touch from quite small distinguishing features. He did not seem to see things completely, but generally made do with very little information provided he was familiar with the object from previous touch experience. When it came to drawing, he was incapable for a year or more of representing features not previously familiar to him. For example, he could not draw the front of a bus—and this he had never touched as too dangerous—though he could draw the rest quite well (Fig. 1), Gregory and Wallace (1963).

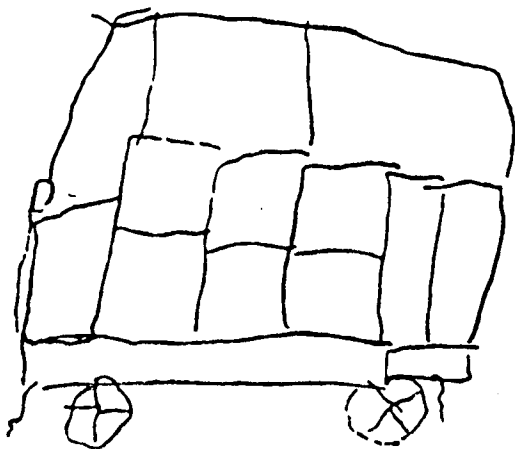


FIG. 1. Drawing of a bus (48 days after the first operation).

The ability of the human perceptual system to extrapolate way beyond the immediate sensory evidence is clearly demonstrated by perception of *imperfect* pictures. Consider cartoons: a few lines . . . we 'see' a complete face, expression and all (Fig. 2). But the blind man who recovered his sight made nothing of cartoons. He lacked the appropriate internal file cards.

Recently, electrophysiological studies of visual processes in lower animals have shown that but few characteristics are coded from the retina and presented to the central nervous system. In the case of the frog, Lettvin,

Maturana, McCulloch & Pitts (1959) found that movement, angles, corners and intensity of light are almost the only features of the world which are signalled to the frog's brain. It is a simple language, but for the frog generally adequate. It has been suggested by W. H. Rushton (1953) that the fovea of the human eye presents information of all pattern, brightness, colour and movement almost completely to our brain, while the peripheral regions of the retina are more selective, as in primitive eyes. As Gordon Walls (1963) points out, the more primitive the brain the more complex the eye. Primitive

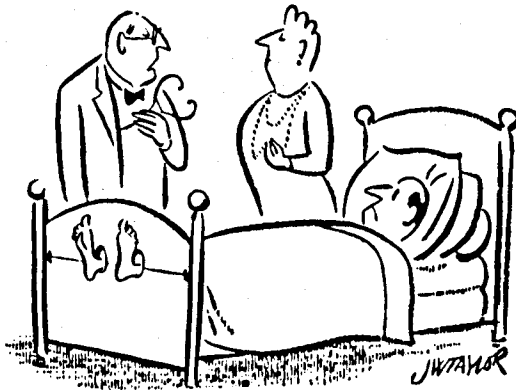


FIG. 2.
'... and keep him in bed for another day or two.'
Drawing reproduced by permission of *Punch*.

eyes are complex in order to transmit only what is necessary for the limited brain computer. Primitive brains have few, and built in, file cards while the higher mammals have simple eyes, but vastly complex brains which develop cross-indexed file cards during the long period of helpless infancy and childhood. It now seems clear that for 'perceptual learning' to occur, there must be repeated active movement with associated visual signals, which come to represent external objects through lengthy experiments with real objects. Vision does not stand by itself; it needs the more direct information from active touch for the signals from the eyes to have meaning, to give perception of the external world of objects. Retinal images are not stimuli producing reflex action—they are more like references to the perceptual file cards, built up by active experiment and conflict with objects.

SEEING OBJECTS IN SPACE

Stereoscopic vision is rare in Nature, and is never of use at distances beyond a hundred feet or so. It is used for precise location of near objects by arboreal and some predator animals. Of far more general importance is the remarkable ability of all large brains to interpret the two-dimensional images of its individual eyes in terms of external three-dimensional space. The problem set is acute, because two-dimensional projections of three-dimensional space are infinitely ambiguous. Retinal images are subject to

an indefinite number of possible interpretations. Consider the simple case of an elliptical-shaped image. It could represent a tilted circle, an ellipse of a certain eccentricity located normal to the line of sight, or any of an infinite set of ellipses of given eccentricity and corresponding angle of tilt. Further, it may be a large distant object, or smaller but correspondingly nearer, to give the same image at the eye. An infinite set of objects lying at various angles and distances give the same image. The brain has to 'decide' which possibility to adopt for perception. When it decides wrongly, we suffer an illusion.

There is generally a wealth of available information—though none entirely reliable—for settling the size and distance of external objects, with sufficient precision for normal use. As is well known, the visual system makes use of a host of 'depth cues', such as gradual loss of detailed texture with increasing distance, haziness due to the atmosphere and nearer objects partly hiding those more distant. These cues were discussed in the nineteenth century by the great von Helmholtz (1925), who fully realised their importance, and they have been the subject of many investigations since, especially by J. J. Gibson (1950). Whatever the richness of depth cues, however, the visual input is always ambiguous. Though the brain makes the best bet on the evidence—it may always be wrong.

The kind of mistakes which occur when the bet is on the favourite though the favourite is not placed, is shown most dramatically by the demonstrations of Adelbert Ames (1946). The most impressive demonstration is given simply with a room which is non-rectangular, but so shaped that it gives the *same retinal image* as a rectangular room to an eye placed in a certain position. Now clearly this room, though queer shaped, *must* appear the same as a normal rectangular room, for it gives the same image to the eye. But consider what happens when objects are placed inside the Ames room. The further wall recedes at one side, so that an object or person standing in one corner is actually at a different distance than is a second object placed at the other far corner. These objects (or people) appear, however, to be at the same distance—and they are seen the *wrong size*. This is clear evidence that we assume rooms to be rectangular (because they usually are) and we interpret the size of objects according to their distance as given by this assumption. When the assumption is wrong we see wrongly. What Ames did was to rig the odds, and then we make the wrong decision on size and distance. A child may appear larger than a man. We may know this is absurd and yet continue to see a bizarre world. The retinal image is all right, but the odds have produced the wrong internal file cards and then the human seeing machine is upset, and gives a wrong answer.

It is interesting that the Ames room is seen correctly by peoples, such as the Zulus, brought up in a 'circular culture' of beehive huts where there are few reliable perspective features, such as rectangular corners and parallel lines, in their visual environment. To the Zulus, the odds are not rigged by the Ames room—to them this is not misleading perspective. They are not

subject to this illusion, but accept the room as the shape it is, and see the objects in it correctly in distance and size. This is a matter of very real importance. It shows that when we are transferred to an alien or bizarre environment, where our filing cards are inappropriate, we interpret the images in the eyes according to principles found reliable in the previous, familiar world—but now they may systematically mislead and then perception goes wrong. Space travellers beware!

The human perceptual system largely compensates for the geometrical shrinking of retinal images with object distance, to give approximate size-distance invariance, known as size constancy. It has recently been suggested, by the author, that size constancy should be attributed to two quite different active scaling processes. One scaling process functions simply according to seen distance. The other is set, or triggered, directly by depth cues—especially perspective features. It seems that we may attribute the well-known 'visual illusion' distortions to the direct setting of this constancy scaling mechanism by perspective features in the illusion figures. In every case, perspective features indicating *distance* produce illusory *expansion*. Now this perceptual expansion with indicated distance would be appropriate if the perspective angles were in fact due to different distances—as in retinal images of the truly 3-D world—but the illusion figures *lie on flat planes*, and so the size scaling induced by their perspective is inappropriate. What is appropriate for the retinal images of the 3-D world is inappropriate for the identical retinal images of 2-D projections of the world. Constancy scaling is inappropriate for pictures, for they are in fact flat, though they have depth cues (Gregory 1963, 1966).

The relation between touch and vision is important, but it has received little investigation until recently so that rather little is known. On general grounds, one can say that vision must ultimately depend on active exploration of the world by immediate contact, since retinal images must somehow come to represent the real world though they are but indirectly related and indeed infinitely ambiguous. This is borne out by the work of R. Held & A. Hein (1963). Current evidence suggests that in man vision is greatly dependent on infant experience, though logically it is possible that retinal patterns may have immediate significance through genetic coding derived from ancestral correlations between vision and touch. It is however an open question how much a man-made seeing-machine should be allowed to learn for itself. If it is to see unusual, or unique, objects it must be capable of some individual learning. We find that animals which are highly adaptable to a variety of environments require a long period, apparently for perceptual learning, before they can respond, except in a stereotyped and reflex manner, to external objects. We should expect the same for man-made seeing machines.

Consider the simple stimulus-response system; or a machine in which the output is directly determined by the state of its inputs. Such systems have a basic limitation—they cannot function with zero or with strictly inadequate

input. To take an example: a car driver can very often continue steering without disaster for several seconds if he is blinded by oncoming lights, or if he sneezes. He steers according to his prevailing 'hypothesis' of the position of the road ahead. He will also base his decisions on what other drivers are likely to do in the immediate future—not merely on what they are doing, as signalled by his eyes. This ability of animals to keep going in the temporary absence of sensory information is vital to their survival, and would seem an essential characteristic of an adequate man-made seeing machine. It involves the notion that seeing is not bound tightly to the current state of affairs as signalled to brain or computer. We may think of perception of the world as ever-changing running hypotheses, suggested and tested by sensory data but never determined wholly by current information from the senses. When sensory information is particularly inadequate, the 'perceptual hypothesis' will diverge from the truth, but generally it will be better than nothing in familiar environments. In strange environments we must expect the normally useful stock of hypotheses to be inappropriate and misleading. Indeed, a refresher course of perceptual learning will be required to cope with it. We know, from studies with distorting spectacles, that the adult human perceptual system is capable of radical readjustment to transformed conditions. This is generally useful, but it follows that perceptual systems are not good at absolute judgements, for we become adapted to changing conditions and lose our ability to tell that they have changed.

The nervous system and especially the sensory systems, apart from those of pain, are generally a.c. coupled. This has obvious advantages: it makes circuits more reliable in the face of internal noise and drift; it emphasises signals indicating change, which are of biological importance, but it can give rise to certain errors. For example the well-known 'waterfall effect'. Prolonged viewing of systematic movement leads to reduction in apparent velocity, and to a curious after-effect when the movement ceases, when illusory movement in the opposite direction is seen. Now it happens that this illusory movement is logically paradoxical—we see movement and yet we see no change in position. This is quite a common feature of sensory illusions, and may be attributed to a state of affairs where one out of two or more parallel channels is disturbed. The false signal may be accepted together with the other, correct, signals—then we experience a logical paradox—a physical impossibility. In the case of the paradoxical 'waterfall' illusion, it seems that movement can be signalled both (i) by change of position in a given time, and also (ii) by direct velocity signals from the retina. (This is analogous to velocity indicated by instruments such as a speedometer which make no reference to a clock.) If either system is upset we may experience a paradox—velocity without changes of position in time. We should expect any machine having parallel channels to produce paradoxes of this kind, when some channels are upset, or receive misleading information. This is fairly common when different senses disagree, but generally one

sense dominates over others so that errors are simply discarded. In man, vision dominates over auditory direction-finding, so that sound generally appears to come from where the source is seen. Thus when wrong, vision can produce *auditory* illusions.

As we saw from the Ames (1946) demonstration of the queer shaped room, the wrong perceptual hypothesis can cause basic sensory information to be wrongly interpreted and then we are misled by the most direct experience. The human seeing machine works very largely by means of a running hypothesis—which to us is reality though always it may be wrong. We are safeguarded from forced inactivity when sensory information is removed, by the prevailing hypothesis flywheeling along in the absence of correction by new data. It will gradually—though in unfortunate cases all too rapidly—get out of date as reality changes, but until then it is a great deal better than nothing. Seeing machines are not stimulus-response devices, but continuously relate incoming data to internal file cards which are kept up to date and serve to give meaning to the fleeting pictures in the eyes. They give due warning of the future, making possible elaborate strategies going far beyond mere reflex behaviour, which is but a hand-to-mouth existence. Thanks to the ancient survival value of space perception—which gives warning of the imminence of distant objects by signalling space and speed, together with identity of Friend and Foe—we now have in our brains seeing machines giving more detailed knowledge than we need for mere survival. Perceptual knowledge goes far beyond the purely sensory characteristics of objects, and is incorporated and becomes one with immediate experience. We see what we know—or perceptually 'think' we know.

Suppose that we wish to make a seeing machine to report an alien world. We must allow it to learn—to relate its past to its present—if it is to do more than transmit pictures which we must interpret with our brains, to do the seeing. If it is to do the seeing it must interpret its own images according to stored data; but these, from an alien world, we may not know. Communication demands a sea of shared assumptions and common data. An alien seeing machine is likely to be difficult to understand—for its filing cards will not be ours, and yet they are essential for it to interpret the pictures in its eyes—to see. It may learn how to survive, but the more surprising and interesting its information, the more difficult it will be for us to learn from its experience. It is tempting to build a seeing machine, and send it to distant worlds, but unfortunately its filing cards will either be inappropriate or give answers we will not understand—as the blind cannot ultimately understand those with sight though they use the 'same' words. The same is true of sending a human seeing machine—a man—to an alien environment. As he learns to see it, so will communication become more difficult and ultimately be impossible. If he, or it, does not learn to see the alien world (by failing to write new file cards to develop perceptual hypotheses) then the sensory information will be inappropriate or have no meaning. Communication will seem easy but the messages will be false. The alien seeing machine will be

riddled by illusions, through misinterpretation of the ambiguous images in its metal and glass eyes.

REFERENCES

- Ames, A. (1946). *Some demonstrations concerned with the origin and nature of our sensations*. Laboratory Manual. Dartmouth Eye Institute.
- Gibson, J. J. (1950). *The perception of the visual world*. Boston: Houghton Mifflin.
- Held, R., & Hein, A. (1963). Movement produced stimulation in the development of visually guided behavior. *J. comp. Psychol.*, 56, 872-876.
- Helmholtz, H. von. (1925). *Physiological Optics*. J. P. Southall, Trs. New York: Optical Soc. Am.
- Lettvin, J. Y., Maturana, H. R., McCulloch, W. A., & Pitts, W. H. (1959). 'What the frog's eye tells the frog's brain.' *Proc. Inst. Radio Engrs*, 47, 1940-1951.
- Rushton, W. H. (1953). *Aspects of retinal physiology*. Rio de Janeiro: Instituto de Biofisica.
- Walls, G. L. (1963). *The vertebrate eye*. London: Hafner.

PATTERN RECOGNITION



PERCEPTION, PICTURE PROCESSING AND COMPUTERS

DR M. B. CLOWES*

M.R.C. PSYCHO-LINGUISTICS RESEARCH UNIT
UNIVERSITY OF OXFORD

INTRODUCTION

The ability to interpret and respond to the significant content of pictures is shared by a large proportion of the animal kingdom and a small but growing number of machines. The machines (usually digital computers) will classify simple shapes and printed letters and digits represented (by means of a suitable television scanner) as a matrix of 1s and 0s. Animals, and specifically man, will do this and more; that is, not only will a human subject correctly recognise letters written badly but also comment, 'That's a funny way to write a two', or 'It's a good portrait, but the eyes are a little too far apart'. No machine can as yet approach this sort of performance. We may well ask, therefore, what if anything we can learn from a study of 'machine perception' which could conceivably help us to understand human perception. The answer lies in the fact that any realistic account of the mechanism underlying perceptual or any other human skills will be complex. The virtue of a computer lies in its ability to capture in a definite form processes of indefinite complexity and subtlety, and moreover, to permit an *evaluation* of the efficacy of the proposed description by trying it out on the actual task.

This paper will be concerned with a description of a new method of interpreting pictures, now under development. The method arises directly out of earlier attempts (Clowes & Parks 1960, Clowes 1962) to develop letter recognition machines for commercial use. However, it now appears that the approach is capable of a wider interpretation in terms both of psychological observations and recent physiological evidence.

* Present address: Computing Research Section, C.S.I.R.O., Canberra, Australia.

HIERARCHIES OF DESCRIPTION

Psychologists concerned with analysing complex behaviour often have to select an appropriate level of description. For example, driving a motor car can be characterised in terms such as 'overtaking', 'giving way', 'filtering', or at a more detailed level in terms of 'changing from 2nd to 3rd gear', 'turning the wheel anti-clockwise', 'depressing the clutch pedal' or yet more detailed still in terms of the positions of the limbs as a function of time. Choosing the correct level of description might be vital to the assessment of the relative skills of two drivers, or the design of two different types of car. On the other hand perhaps all these levels of description should be employed simultaneously—if so, how are they related? This question forms the basis of a book *Plans and the structure of behaviour* (Miller, Galanter & Pribram 1960) in which it is shown that the problem arises not only for manual skills, but also in many cognitive processes and especially in language. It is in this latter case that perhaps the clearest expression of a hierarchy of description is evident—phonemes, syllables, words, phrases, clauses, sentences . . .—and it is also the area in which the most progress has been made in answering these questions.

LINGUISTIC GRAMMARS

The problem is to formulate a reasonably precise set of rules by means of which we can take an utterance, e.g., 'the boy kicked the red ball' and, by

S_1	=	NP	+	VP	}	----- (1)
S_2	=	NP	+	VP_1		
VP	=	V	+	NP		----- (2)
VP_1	=	V	+	ANP		----- (3)
NP	=	T+N	}			----- (4)
ANP	=	$T+N_1$				
N_1	=	A+N				----- (5)
V	=	kicked/rolled/pushed/	}			-- (6)
T	=	the/a/every/				
N	=	man/boy/ball/				
A	=	red/young/striped/				

FIG. 1. A set of ordered substitution rules specifying the grammar of a simple sentence.

applying the rules in the prescribed manner, can obtain a description of this utterance at a number of different levels.

Let us confine our attention to three levels: words, phrases, sentences. Fig. 1 shows a set of rules sub-divided into numbered groups (1, 2, 3, . . . , 6).

We can regard each rule as defining a permissible substitution. Thus group 6 rules state that we may replace any of the words *red, young, striped* (i.e., any adjective) by the symbol A, and similarly for verbs, nouns and articles. So that the sentence

the boy kicked the red ball

becomes by application of group 6

(6) T N V T A N

Group 5 states that a pair of symbols A, N appearing *contiguously and in that order* can be replaced by N_1 so that the string becomes

(5) T N V T N_1

and so by application of successive rules, in descending order of their group number, we obtain

(4) NP V ANP

(3) NP VP_1

(2) NP VP_1

(1) S_2

If we now combine these different strings of symbols into one diagram (Fig. 2) we have a description of the utterance at a number of levels in a fashion which indicates the relation between the levels.

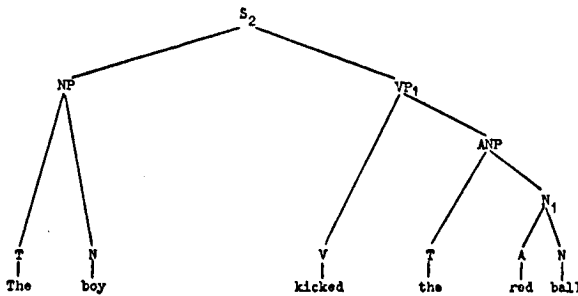


FIG. 2. The structural analysis of a simple sentence as obtained by applying the rules of FIG. 1.

The rules may be used to generate or analyse sentences. A typical generative sequence starting from the schema S_2 would be $S_2 \rightarrow NP VP_1 \rightarrow T N VP_1 \rightarrow the N VP_1 \rightarrow the boy VP_1 \rightarrow the boy V ANP \rightarrow \dots \rightarrow the boy kicked the red ball$. The generative path can be diagrammed in tree form as in Fig. 2.

This diagram—a so-called *tree*—indicates not only that the utterance is a sentence, but also what type of sentence it is (S_2 rather than S_1), as well as indicating what features (i.e., phrases like ANP, etc.) it had which caused

the label S_2 to be applicable. Let me hasten to add at this point that this is not presented as a serious piece of linguistics (there are grounds for believing it to be incorrect in several ways) but merely as an introduction to the notion of a set of rules (the grammar) and a procedure for applying them (a parsing algorithm) by which we can rigorously derive a *structural description* of a complex object like a sentence. A mechanism or mechanisms of this type undoubtedly underlie our ability to understand sentences.

PICTURE PROCESSING

Comparable mechanisms for processing pictures have not appeared to any significant extent either in Artificial Intelligence or in Psychology. Instead the emphasis has been on classifying pictures, i.e., upon the equivalent of deciding the type (S_1 or S_2) of a sentence. The possibility that a structural description of a pictorial object is necessary has only recently emerged in Artificial Intelligence (Kirsch 1964). It appears to be crucial to the automation of many picture processing tasks (e.g., interpreting bubble-chamber photographs, recognising fingerprints). That perceptual behaviour involves description as well as classification could be supported by innumerable examples beyond the two already quoted in the introduction to this paper. However, it seems to have received scant attention in recent psychological literature. This may well be because overtly descriptive (rather than classificatory) behaviour has a large verbal element. There is a strong temptation to avoid this 'complication' by designing experiments which merely require simple YES/NO responses and which have, therefore, no significant verbal component.

It follows from the foregoing remarks that our own objective is the development of a formal (i.e., rigorous) theory of picture processing which aims to provide a structural description of pictorial objects as well as a classification of them (wherever appropriate). The particular system I shall describe is by no means the only possible approach, both Ledley (1964) and Narasimhan (1964) have proposed alternatives. I shall illustrate the technique in terms of a limited class of pictures—the handwritten numerals. There are several reasons for choosing this class. (1) They are (hopefully) relatively simple objects; (2) at least two levels of description naturally occur, e.g., the symbol 7 is readily described as 'a horizontal stroke above a diagonal stroke . . .'; (3) psycholinguistic skills, e.g., reading (with which the Unit† is primarily concerned), involve pictorial objects of this type.

A PICTURE GRAMMAR FOR HANDWRITTEN NUMERALS: NOTATION

The substitution rules in the linguistic grammar (Fig. 1) contain on their right-hand side (typically) two elements, e.g., T, N or NP, VP, linked by a relational operator '+' meaning 'followed by'. This operator expresses the notion of contiguity as it arises in a *string* of symbols; contiguity in a

† MRC Psycholinguistics Research Unit, University of Oxford.

geometrical sense is the more general relationship 'next to'. Thus a simple pictorial property like 'Edge' (i.e., boundary between black and white regions of the picture) might be defined as 'black area next to white area'. Specifically we might define a North Edge as 'white area *above* black area', which can be expressed in a rigorous (computable) form with the aid of a suitable notation. For example, in a pictorial notation the definition of North Edge for a picture composed only of black or white elements might take the form illustrated in Fig. 3. This asserts that the point arrowed

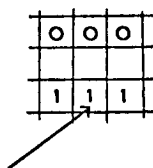


FIG. 3.

will be said to have the property N EDGE if it and the points to the left and the right of it are black (i.e., *True*) and the three points denoted '0' are white (i.e., *False*). If we introduce a Cartesian co-ordinate system we can write this definition as a Boolean expression:

$$\begin{aligned} \text{N EDGE } (x, y) = & \text{PICTURE } (x-1, y) \ \& \ \text{PICTURE } (x, y) \\ & \& \ \text{PICTURE } (x+1, y) \ \& \ \sim \text{PICTURE } (x-1, y+2) \\ & \& \ \sim \text{PICTURE } (x, y+2) \ \& \ \sim \text{PICTURE } (x+1, y+2) \end{aligned} \quad (1)$$

where (x, y) is the 'arrowed' point, a black point has the value *True* and a white point the value *False*.

The assignment of a label (e.g., N EDGE) to a picture point involves a decision based upon the state of a number of neighbouring points. Fig. 3 is a picture of the decision criterion. The values of only 6 specified points are considered. All other points are ignored, for the purpose of assigning this label to the arrowed point.

IMPLEMENTATION

This notation is used in the picture analysis program with some small differences for reasons of economy and ease of programming. Fig. 4 shows a system of such definitions as they are input to the computer. The first two lines specify various program options relating to print out, etc., and also define the size (48×48) of the input picture. The third line is the definition for N EDGE given above (1), but specified in Reverse Polish (i.e., parenthesis free) form.[†] Repetitions of the label PICTURE have been suppressed and the label N EDGE appears at the end of the definition rather than preceding it. The x, y appearing on both sides of the expressions (1) above have also been omitted, it being understood that this rule is applied to the picture (Fig. 5(a)) for all values of (x, y) so as to derive another picture called (Z) N EDGE (Fig. 5(b)). The axes have been rotated so that x is measured down the

[†] The Reverse Polish form of the Boolean expression $(A \ \& \ B) \vee (\sim C \ \& \ D)$ is $A, B, \ \&, C, \ \sim D, \ \&, \vee$.

FIG. 4. A partial 'picture grammar' for handwritten numerals. Each group of rules is preceded by the control instruction + CODE>NNL + 02BY02.

```

+LINEYX+48BY48
+CODE>NNL+02BY02
ZPICTURE.(00,00) NOT      .(00,01) NOT AND      .(00,02) NOT AND      .(02,00) AND      .(02,01) AND      .(02,02) AND
Z(02,01)Z EDGX048,48
ZPICTURE.(00,00) NOT      .(00,01) NOT AND      .(00,02) NOT AND      .(02,00) AND      .(02,01) AND      .(02,02) AND
Z(02,01)Z BLOX000,72
ZPICTURE.(00,00).(01,00) AND      .(02,00) AND      .(00,02) NOT AND      .(02,02) NOT AND      .(01,02) NOT AND
O(01,00)ZE EDGX024,00
ZPICTURE.(00,00).(01,00) AND      .(02,00) AND      .(00,02) NOT AND      .(02,02) NOT AND      .(01,02) NOT AND
O(01,00)Z BLOX000,72
ZPICTURE.(00,00).(00,01) AND      .(00,02) AND      .(02,00) NOT AND      .(02,02) NOT AND      .(02,02) NOT AND
Z(00,01)Z EDGX024,48
ZPICTURE.(00,00).(00,01) AND      .(00,02) AND      .(02,00) NOT AND      .(02,02) NOT AND      .(02,02) NOT AND
Z(00,01)Z BLOX000,72
ZPICTURE.(00,00) NOT      .(01,00) NOT AND      .(02,00) NOT AND      .(02,00) NOT AND      .(01,02) AND      .(02,02) AND
Z(01,02)Z EDGX048,00
ZPICTURE.(00,00) NOT      .(01,00) NOT AND      .(02,00) NOT AND      .(02,00) NOT AND      .(01,02) AND      .(02,02) AND
Z(01,02)Z BLOX000,72
ZPICTURE.(00,02).(01,01) AND      .(02,00) AND      .(02,03) NOT AND      .(03,02) NOT AND      .(04,01) NOT AND
Z(01,01)ZSE EDGX024,24
ZPICTURE.(00,02).(01,01) AND      .(02,00) AND      .(02,03) NOT AND      .(03,02) NOT AND      .(04,01) NOT AND
Z(01,01)Z BLOX000,72
ZPICTURE.(00,02).(01,03) AND      .(02,04) AND      .(02,01) NOT AND      .(03,02) NOT AND      .(04,03) NOT AND
Z(01,03)ZSW EDGX024,72
ZPICTURE.(00,02).(01,03) AND      .(02,04) AND      .(02,01) NOT AND      .(03,02) NOT AND      .(04,03) NOT AND
Z(01,03)Z BLOX000,72
ZPICTURE.(00,03) NOT      .(01,02) NOT AND      .(02,01) NOT AND      .(04,02) AND      .(03,03) AND      .(02,04) AND
Z(03,03)ZSW EDGX048,24
ZPICTURE.(00,03) NOT      .(01,02) NOT AND      .(02,01) NOT AND      .(04,02) AND      .(03,03) AND      .(02,04) AND
Z(03,03)Z BLOX000,72
ZPICTURE.(00,01) NOT      .(01,02) NOT AND      .(02,03) NOT AND      .(02,00) AND      .(03,01) AND      .(04,02) AND
Z(03,01)ZNE EDGX048,72
ZPICTURE.(00,01) NOT      .(01,02) NOT AND      .(02,03) NOT AND      .(02,00) AND      .(03,01) AND      .(04,02) AND
Z(03,01)Z BLOX000,72

```

Fig. 4. (continued)

```

*CODENL*02BY02
ZW EDGE*(02,01)ZE EDGE*(02,01),(02,02) OR AND Z BLOB*(01,00) NOT AND .(00,01
NOT AND .(00,02) NOT AND 2(02,01)ZN ENDX048,60 AND .(01,03) NOT AND
ZN EDGE*(01,02)ZS EDGE*(01,02),(02,02) OR AND Z BLOB*(01,00) NOT AND .(02,00) NOT
NOT AND .(03,01) NOT AND 2(01,02)ZN ENDX048,12 AND .(01,02) NOT AND
ZN EDGE*(01,00)ZS EDGE*(01,00),(02,00) OR AND Z BLOB*(00,01) NOT AND .(02,02
NOT AND .(03,01) NOT AND 2(01,00)ZE ENDX048,00 AND .(02,01) NOT ANQ
ZW EDGE*(00,01)ZE EDGE*(00,01),(00,02) OR AND Z BLOB*(01,00) NOT AND .(02,02
NOT AND .(01,03) NOT AND 2(00,01)ZS ENDX048,48 AND .(01,01) NOT AND
ZNW EDGE*(00,02)ZSE EDGE*(00,02),(01,03) OR AND Z BLOB*(00,00) NOT AND .(02,02
NOT AND .(03,01) NOT AND 2(00,02)ZSW ENDX048,72 AND .(02,01) NOT AND
ZNE EDGE*(02,03)ZSW EDGE*(02,03),(03,02) OR AND Z BLOB*(03,00) NOT AND .(01,02
NOT AND .(00,03) NOT AND 2(02,03)ZNW ENDX048,36 AND .(01,01) NOT AND
ZNW EDGE*(02,00)ZSE EDGE*(02,00),(03,01) OR AND Z BLOB*(00,00) NOT AND .(02,02
NOT AND .(03,03) NOT AND 2(02,00)ZNE ENDX048,84 AND .(01,01) NOT AND
ZNE EDGE*(00,01)ZSW EDGE*(00,01),(01,00) OR AND Z BLOB*(03,00) NOT AND .(01,02
NOT AND .(01,02) NOT AND 2(00,01)ZSE ENDX048,24 AND
ZN EDGE*(00,00)ZS EDGE*(00,00),(01,00) OR AND 200 LINEX060,00
ZNW EDGE*(00,00)ZSE EDGE*(00,00),(01,01) OR AND 245 LINEX060,36
ZN EDGE*(00,00)ZSE EDGE*(00,00),(00,01) OR AND 290 LINEX060,24
ZNE EDGE*(00,01)ZSW EDGE*(00,01),(01,00) OR AND 2135 LINEX060,12
ZE EDGE*(00,00),(01,00) AND .(02,00) AND 2(01,00)ZE EDGEX036,00
ZS EDGE*(00,00),(00,01) AND .(00,02) AND 2(00,01)ZS EDGEX036,24
ZW EDGE*(00,00),(01,00) AND .(02,00) AND 2(01,00)ZW EDGEX036,48
ZSE EDGE*(00,00),(00,01) AND .(00,02) AND 2(00,01)ZNE EDGEX036,72
ZSW EDGE*(00,02),(01,01) AND .(02,00) AND 2(01,01)ZSE EDGEX036,12
ZSNW EDGE*(00,00),(01,01) AND .(02,02) AND 2(01,01)ZSW EDGEX036,36
ZNE EDGE*(00,02),(01,01) AND .(02,00) AND 2(01,01)ZNE EDGEX036,60
ZW EDGE*(00,00),(01,01) AND .(02,02) AND 2(01,01)ZNE EDGEX036,84

```

Fig. 4. (continued)

```

*CODENNL*02BY02
Z90 LINE.(00,00).(01,00) OR ZNW END.(00,00)ZN END.(00,00) OR ZNE END.(00,00) OR AND ZM LIMX072,00
Z00 LINE.(00,00).(00,01) OR ZNE END.(00,01)ZN END.(00,01) OR ZSE END.(00,01) OR AND ZM LIMX072,40
X072,12
Z90 LINE.(00,00).(01,00) OR ZSE END.(01,00)ZN END.(01,00) OR ZSW END.(01,00) OR AND ZM LIMX072,80
X072,24
Z00 LINE.(00,00).(00,01) OR ZSW END.(00,00)ZN END.(00,00) OR ZNW END.(00,00) OR AND ZM LIMX072,24
Z00 LINE.(00,01)Z45 LINE.(00,00).(01,00) OR AND Z135LINE.(00,02).(01,02) OR AND Z100,01)ZN CURVX072,40
Z90 LINE.(01,01)Z135LINE.(00,00).(00,01) OR AND Z45 LINE.(02,00).(02,01) OR AND Z101,01)ZE CURVX072,60
Z00 LINE.(01,00)Z45 LINE.(00,00).(01,00) OR AND Z45 LINE.(00,02).(01,02) OR AND Z101,01)ZE CURVX072,72
Z90 LINE.(00,00)Z45 LINE.(00,00).(01,00) OR AND Z135LINE.(02,00).(02,01) OR AND Z101,00)ZW CURVX072,84
ZN EDGE.(00,00)Z5 EDGE.(00,00).(01,00) OR AND Z00 LINEX060,00
ZNW EDGE.(00,00)Z5 EDGE.(00,00).(01,01) OR AND Z45 LINEX060,72
ZM EDGE.(00,00)ZE EDGE.(00,00).(00,01) OR AND Z90 LINEX060,48
ZNE EDGE.(00,01)ZSW EDGE.(00,01).(01,00) OR AND Z135LINEX060,24
Z00 LINE.(00,00).(00,01).(00,02) AND Z100,01)Z00 LINEX060,00
Z45 LINE.(02,00).(01,01).(00,02) AND Z101,01)Z45 LINEX060,72
Z90 LINE.(00,00).(01,00).(02,00) AND Z101,00)Z90 LINEX060,48
Z135LINE.(00,00).(01,01).(02,02) AND Z101,01)Z135LINEX060,24

*CODENNL*02BY02
*{(00,00)ZERO X180,00
*{(00,00)ZONE X180,09
Z45 LINE.(02,01)Z5 LINE.(00,00).(01,00) OR AND ZN CURV.(00,01)ZE' CURV.(01,02) OR AND ZE LIM8.(03,01
*{(03,02) OR AND Z102,01)ZTW0 X180,10
*{(00,00)ZTHREE X180,27
*{(00,00)ZFOUR X180,36
*{(00,00)ZFIVE X180,45
*{(00,00)ZSIX X180,54
*{(00,00)ZSEVEN X180,63
*{(00,00)ZEIGHT X180,72
*{(00,00)ZNINE X180,81

```

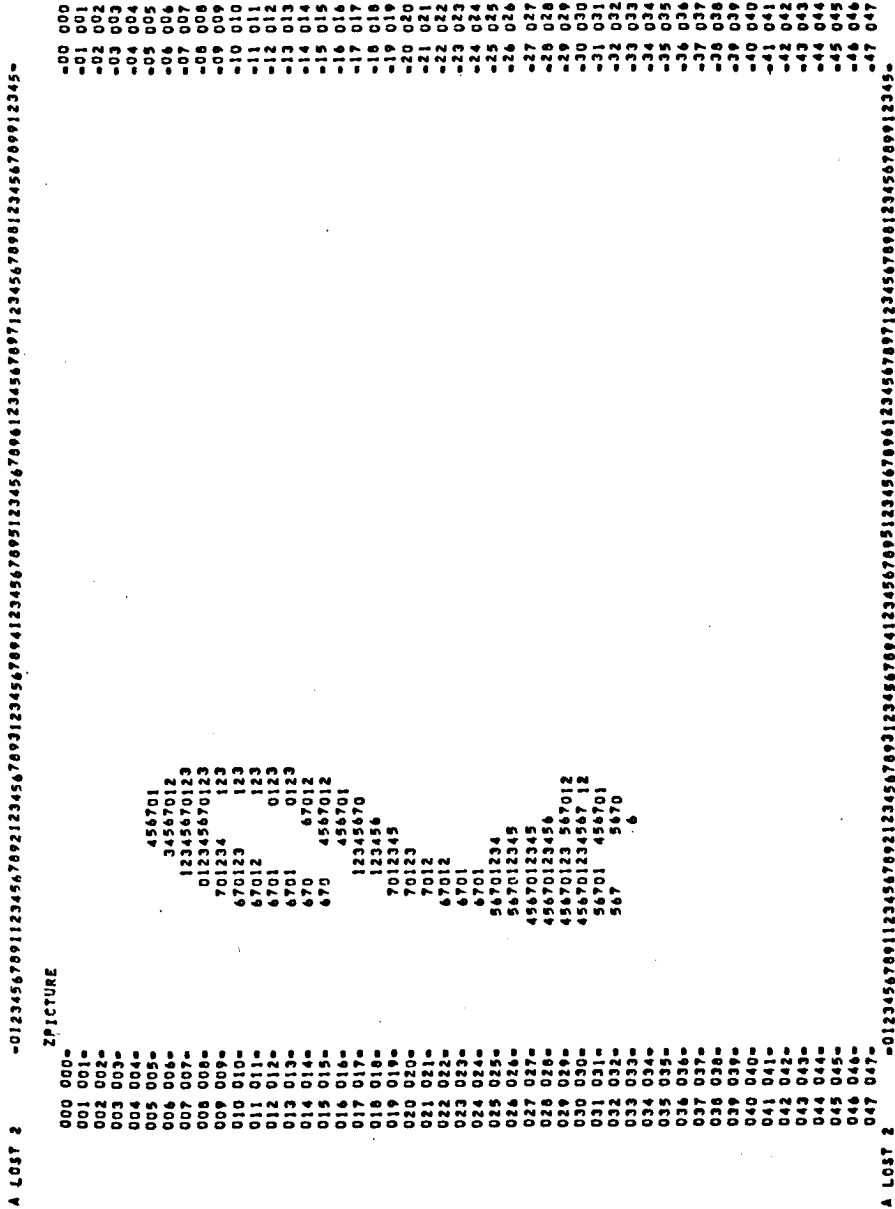


Fig. 5(a). The picture to be analysed. The origin of coordinates is the cell immediately beneath the Z of Z PICTURE. The picture is defined to be only 48 x 48 elements hence in the direction (down the page) we can only print lines 0 to 47. Each 'black' element is represented internally as a binary digit. For display purposes, however, the 'black' elements are numbered (on a scale of 8) in the horizontal direction.

FIG. 5(b). (continued)

[illegible]

A LOST 2																	-0123456789112345678921234567893123456789412345678951234567896123456789712345678981234567899123456-																
000 036-		ZE	EDGE	ZSE	EDGE	ZS	EDGE	ZSW	EDGE	ZW	EDGE	ZNW	EDGE	ZN	EDGE	ZNE	EDGE																
001 037-				6																													
002 038-				6	0									0																			
003 039-					0 2					3				1																			
004 040-					12									01																			
005 041-										3																							
006 042-																																	
007 043-																																	
008 044-							4																										
009 045-																																	
010 046-																																	
011 047-																																	
000 048-		ZE	END	ZW	END	ZSE	END	ZNW	END	ZS	END	ZN	END	ZSW	END	ZNE	END																
001 049-																																	
002 050-																																	
003 051-																																	
004 052-																																	
005 053-																																	
006 054-																																	
007 055-																																	
008 056-																																	
009 057-																																	
010 058-																																	
011 059-																																	
000 060-		Z00	LINE	Z135	LINE	Z90	LINE	Z45	LINE																								
001 061-			5		2																												
002 062-																																	
003 063-			5				6		70																								
004 064-							3 6		12																								
005 065-									01																								
006 066-							34																										
007 067-																																	
008 068-																																	
009 069-																																	
010 070-																																	
011 071-																																	
A LOST 2																	-0123456789112345678921234567893123456789412345678951234567896123456789712345678981234567899123456-																
-36 000																																	
-37 001																																	
-38 002																																	
-39 003																																	
-40 004																																	
-41 005																																	
-42 006																																	
-43 007																																	
-44 008																																	
-45 009																																	
-46 010																																	
-47 011																																	
-48 000																																	
-49 001																																	
-50 002																																	
-51 003																																	
-52 004																																	
-53 005																																	
-54 006																																	
-55 007																																	
-56 008																																	
-57 009																																	
-58 010																																	
-59 011																																	
-60 000																																	
-61 001																																	
-62 002																																	
-63 003																																	
-64 004																																	
-65 005																																	
-66 006																																	
-67 007																																	
-68 008																																	
-69 009																																	
-70 010																																	
-71 011																																	

FIG. 5(c). The outcome of the second group of rules applied to the contents of Fig. 5(b).

[illegible]

FIG. 5(d). The outcome of the third group of rules applied to the contents of Fig. 5(c).

[illegible]

FIG. 5(e). 'Recognition' as obtained by applying the fourth group of rules to the contents of Fig. 5(d). Note that the array size is now 3×3 elements. In principle more than one figure could have been presented in the initial Z PICTURE. Their relative position and identity would have been reflected in this display.

page and y across it, following the numbering appearing in the picture frame. Following the label (Z) N EDGE a further pair of coordinates (048, 48) specify the location in computer memory at which the resultant picture (Z) N EDGE is to be stored. The first group of definitions in the grammar expresses eight EDGE definitions corresponding to eight points of the compass, and a further definition (Z) BLOB which merely collects together the results of all the EDGE definitions in a single picture called (Z) BLOB. The next set of rules in the grammar contains concatenations of EDGES to define LINES having four orientations 00, 45, 90, 135 and line ENDS with eight orientations. These in turn are concatenated to form CURVES (combinations of appropriately oriented lines), LIMBS (combinations of lines and ends) and LINES (combinations of lines). Each set of rules in the picture grammar is separated by a line beginning +CODE. . . .

The action of the program is to apply each set of rules to the outcome of the previous set starting with a picture stored in computer memory (cf. the linguistic grammar). The performance of the program is monitored by having it print out those portions of its memory which currently contain a picture after each set of rules has been applied. The area of memory is—in effect—a matrix of 192×96 elements. At any one stage only a portion of this is in use, the portions being allocated by the coordinates associated with picture name given in the grammar, i.e., Z (N) EDGE (048, 48) is stored starting from location 48 (down), 48 (across). Thus the output for the first set of rules is as shown in Fig. 5(b).

The size of the initial picture (Z) PICTURE is 48×48 cells, the EDGE versions are, however, reduced in size to occupy only 24×24 cells. The necessity for this reduction in size is intimately connected with the form of picture grammar we have chosen to implement. A feature of the linguistic grammar was that substitutions were always defined for pairs of *adjacent symbols*, nowhere do we find a rule which calls for the replacement of symbols widely separated in the string being analysed. In spite of this immediate constituent or phrase structure restriction it is still possible to reflect with a single label (S) the structure of an extended utterance, because as we proceed with the analysis the string automatically shrinks by the very nature of the substitution operation. In the geometrical analysis we perform on a picture, the same immediate constituent limitation is imposed. If, however, we are able to label an extended property of a picture, e.g., a long line, with 'next to' concatenations, we need to draw together these local properties—hence this reduction in the size of the EDGE versions of the picture.

We can summarise the outcome of this 'parsing' process in a different form by collecting together the contents of several pictures, e.g., ENDS and LINES (Fig. 5(c)), and combining them into a single picture with a suitable pictorial convention to indicate the type of picture from which the entry originates (Fig. 6(b)). The results of the next level of description (Fig. 5(d)) containing CURVES, LIMBS and LINES can similarly be displayed pictorially (Fig. 6(c)).

They show that the program has adequately labelled the main features of this '2' and for the purposes of recognition we can define each numeral as being some combination of features. For a '2' this definition will involve a curve, a diagonal and a horizontal; and the last group of rules (Fig. 4) contains such a definition. Comparable definitions for the other numerals can be written.

Definitions of this type do not, strictly speaking, constitute a part of the grammar, and it is with the adequacy of the grammar that we are, at this point, concerned. Fig. 6(c) shows that certain regions of the '2', e.g., the intersection of the diagonal and the horizontal, are unlabelled. This arises from the adoption of 'lines' as syntactic categories: we are now studying a grammar having the same form but concerned exclusively with EDGES.

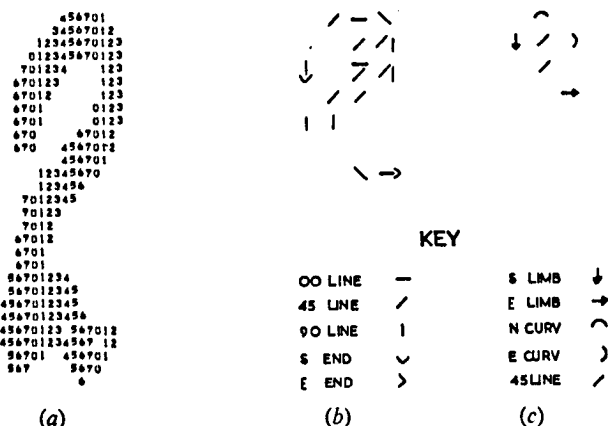


FIG. 6. A pictorial 'summary' of the results of the analysis. (a) The input picture Z PICTURE. (b) The contents of Fig. 5(c) super imposed and coded according to the convention given in the KEY. (c) The contents of Fig. 5(d) similarly encoded.

RELATION TO PHYSIOLOGY

It is possible to compare the organisation of this system with the organisation of the visual system, as revealed by microelectrode studies in the cat (Hubel & Wiesel 1962, 1965) and the frog (Lettvin, Maturana, McCulloch & Pitts 1959). Briefly the following points emerge:

(1) Cells in the visual cortex only respond to local properties of the visual scene, e.g., edges, line segments. This mirrors the immediate constituent constraint imposed for economic reasons in the picture grammar.

(2) The visual scene is portrayed many times over in the cortex in terms of the different picture properties, each portrayal being 'laid-out' two-dimensionally and 'in register' with all other portrayals. This corresponds to the sets of arrays which are also preserved 'in register', in the computer.

(3) In addition to cells portraying the local properties of corresponding regions of the retinal image, there are also cells (so-called 'complex' cells—

Hubel & Wiesel 1962) which indicate whether a particular local property occurs anywhere over a small area of the retinal image. Such cells are precisely what one would expect to find mediating the 'reduction' of scale (and resolution) injected into the picture grammar.

The detailed nature of the correspondence means that we can usefully consider the possible significance of discrepancies. For example, there are grounds for believing that the picture grammar is premature in beginning with EDGE definitions. In the visual system a considerable amount of processing—so called 'lateral inhibition'—precedes EDGE operations and is performed in the retina. Conversely we may ask of the physiologists what might correspond to certain features of our picture grammar which are designed to provide size invariance. This type of detailed cross-fertilisation is in many ways quite novel, at least with this subject matter.

The notation can also be used to express physiological results, e.g., the properties of a receptive field unit.

RELATION TO PSYCHOLOGY

Ultimately, the psychological relevance of this form of picture analysis will be in its ability to derive structural descriptions of pictorial objects which reflect our intuitions of their shape. To a limited extent the labelling already demonstrated achieves this: it will be more convincing, however, if we can, for example, show that the same grammar assigns adequate descriptions to objects which are generically different, e.g., fingerprints, chromosomes. It can already be shown, however, that by virtue of the way in which it computes picture labels, the system has some interesting weaknesses. Given a numerical value for the notion of local property, e.g., that no definition can refer to array elements separated by more than 10 units, the system is unable to judge the positions of an object relative to some frame, to better than 10 per cent (i.e., 1/10). This limitation appears in many forms of sensory judgment (Miller 1956). Again the system is unable to 'pick out' (i.e., uniquely label) one member of a large matrix of similar objects, e.g., a matrix of A's. This is a familiar experience. We overcome the limitation by 'tracing' or counting along a line with finger or pencil point. Significantly this use of a *pointer* changes the structure of the picture in a way which would also greatly help the computer system.

CONCLUSION

We can now return to a general remark made in the introduction. The essential characteristic of the system we have described is the volume of operations it involves. To calculate by hand the structural analysis illustrated earlier would occupy perhaps 2-3 hours for each numeral and would inevitably contain many mistakes. The validation of a picture grammar in terms of a whole alphabet and subsequently for other classes of picture would, therefore, be unthinkable except by computer. In that we already

CLOWES

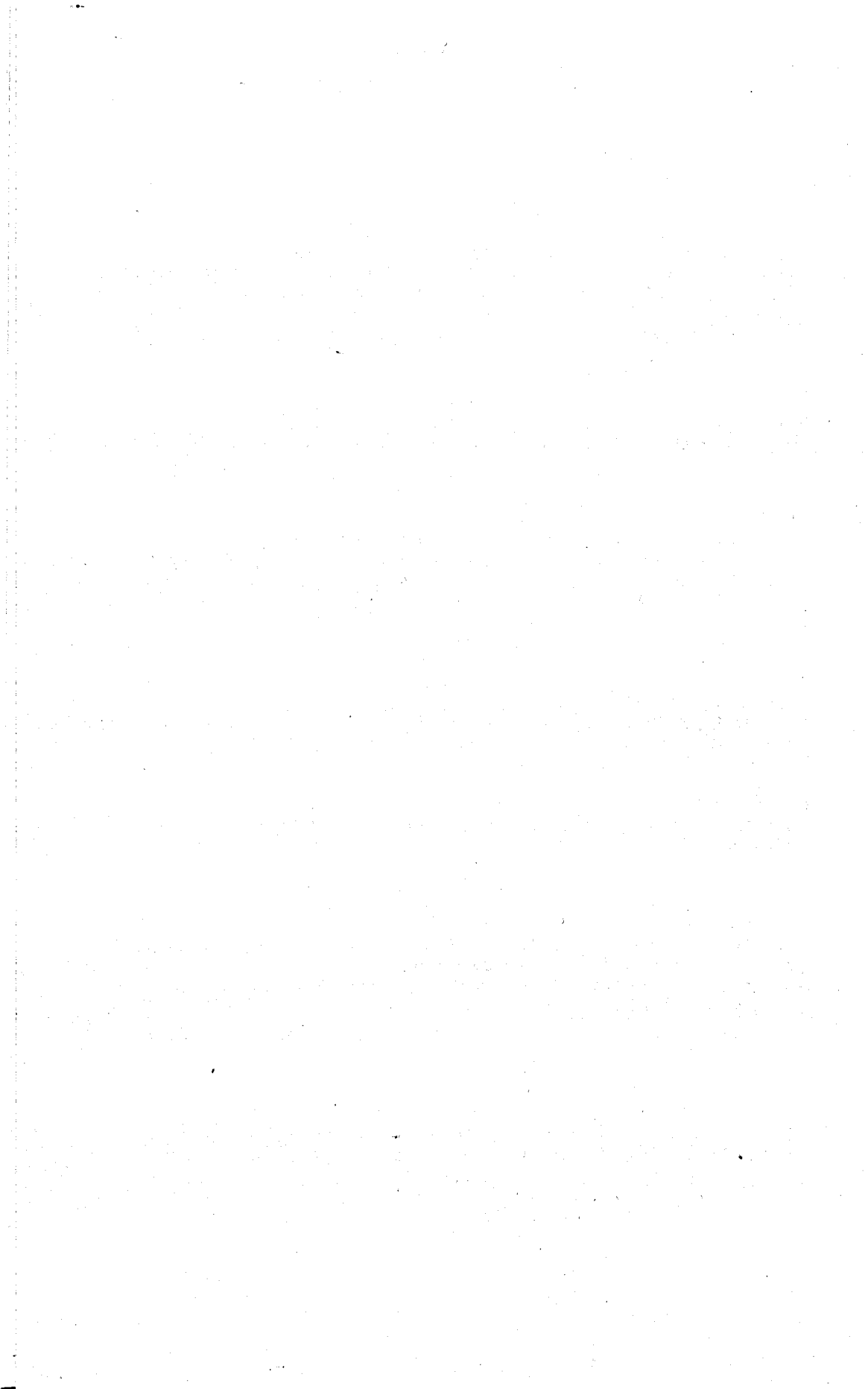
sense directions in which our approach requires further complications due to over-simplifications like assuming pictures to be only black or white, the claim that psychological theories are necessarily complex, and *require a computer to test them*, will need no further elaboration. In one way it is unfortunate that Artificial Intelligence has been so called, since it suggests that it is concerned with problems quite different from those traditionally studied by the biologist. In fact, however, both disciplines are concerned with the study of very complex systems and collaboration is becoming increasingly fruitful.

ACKNOWLEDGEMENTS

I would like to acknowledge with gratitude the use of the computing facilities at the Culham Fusion Research Laboratory, U.K.A.E.A., the assistance of Douglas Brand and Barry Astbury, and valuable discussions with John Marshall, Professor N. S. Sutherland and Professor R. C. Oldfield.

REFERENCES

- Clowes, M. B. (1962). The use of multiple auto-correlation in character recognition. *Optical Character Recognition*. Fischer, Pollack, Radack & Stevens, Eds. Pp. 305-318. Baltimore: Spartan Books, Inc.
- Clowes, M. B., & Parks, J. R. (1961). A new technique in character recognition. *Comput. J.*, **4**, 121-126.
- Hubel, D. H., & Wiesel, T. N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *J. Physiol.*, **160**, 106-154.
- Hubel, D. H., & Wiesel, T. N. (1965). Receptive fields and functional architecture in two non-striate visual areas (18 & 19) of the cat. *J. Neurophysiol.*, **28**, 229-289.
- Kirsch, R. A. (1964). Computer interpretation of English language test and picture patterns. *Instn. elect. Engrs. Trans. on Electronic Computers*, **EC-13**, 363-376.
- Ledley, R. S. (1964). High speed automatic analysis of biomedical pictures. *Science*, **146**, 216-223.
- Lettvin, J. Y., Maturana, H. R., McCulloch, W. S., & Pitts, W. H. (1959). What the frog's eye tells the frog's brain. *Proc. Inst. Radio Engrs.*, **47**, 1940-1951.
- Miller, G. A. (1956). The magical number seven. *Psych. Rev.*, **63**, 81.
- Miller, G. A., Galanter, E., & Pribram, K. H. (1960). *Plans and the Structure of Behavior*. New York: Henry Holt and Company, Inc.
- Narasimhan, R. (1964). Labelling schemata and syntactic descriptions of pictures. *Inf. Control*, **7**, 151-179.



AUTOMATIC SPEECH RECOGNITION: A PROBLEM FOR MACHINE INTELLIGENCE

DAVID R. HILL

STANDARD TELECOMMUNICATION LABORATORIES LTD
HARLOW

SUMMARY

Speech recognition by machine has not yet been achieved because no suitable specification of the recognition process has been formulated for the machine. The author outlines the disturbances and constraints found in speech, and goes on to a description of the structure implied by the constraints. This description is a prerequisite of speech recognition for two reasons, first to describe general speech structure in terms which allow knowledge of it to be built into the machine as an aid to the recognition process, secondly to allow a good enough description of the input signal for it to lead to a minimum set of recognition possibilities which includes likely alternatives. The outline is drawn of a hypothetical machine to recognise speech, comprising a basic recogniser working on short segments of acoustic waveform only, on to which may be added further structures to use knowledge of speaker characteristics, speech statistics, syntax rules, and semantics, in order to improve the recognition performance. Some detailed examples of possible structures are given. Finally there is a brief description of work in progress at Standard Telecommunication Laboratories towards implementing a basic recogniser of the type suggested.

INTRODUCTION

In a recent survey paper Lindgren (1965) says, '... the immediate aim of building automata that can recognise speech seems somewhat in abeyance'. Miller, on the same subject, says that the engineers concerned have a right to be discouraged. Why have two or more decades' of intensive research been rewarded with such apparent lack of success? It is not due to lack of

means of analysis for the acoustic signal. What is difficult is telling the machine what to do with the results of the analysis, for most machines working today work because someone has not only been able to tell them what to do but has been able to do so within the limitations of the machine. Suppose one tried to implement a recogniser by telling the machine to store every new pattern it encountered together with a label telling it what word or words the pattern represented, with the intention of recognising an arbitrarily large vocabulary for an arbitrarily large proportion of the total population of speakers. It would never work for a number of good reasons. It would have an endless need to store new data, and would therefore be so enormous that, even if it could be built and even if a sufficient number of the parts functioned correctly at the same time, it would be quite uneconomic, and would constantly be unable to make decisions through lack of data. The problem is to find a description of the recognition process which is sufficiently economical for it to be built into a machine to render the machine capable of performing a useful job at a competitive price.

Much early work proceeded on the assumption that there were, in the acoustic signal representing speech, invariant data groupings representing the phonemes used by linguists to categorise speech sounds in each particular language. It was hoped that these could be extracted to allow straightforward classification into the same categories, and hence into words. One important result of the work on the problem has been to show that this is not true, that in many cases there simply is not sufficient information in the acoustic signal representing a word to determine completely the word it is intended to represent (for example, Miller 1962). This is another reason why our hypothetical monster recogniser would not work, unless it worked on comparatively long portions of the input signal at something like the sentence level which would aggravate the storage problem. Miller (1964a) has conservatively estimated that 10^{20} sentences could be constructed, and these would take 1000 times the age of the earth even to utter! He (Miller 1965), Fry (1956) and Denes (1959), as well as others, have taken great pains to emphasise that the recognition of speech by human beings depends not only on the acoustic signal reaching the ear but on the whole structure of language. That man is the only biological system constructed to use language the way he uses it is suggested by the failure of attempts so far to teach our most intelligent non-humans to use language. It is very likely that our intelligence arises from our ability to use language rather than the other way round (Miller 1964b). One is tempted to suggest that the title of this paper could read, 'Intelligence—a problem for machine language' and still remain of great interest to the assembled company, for language is the 'stuff' of thought, and without thought we should be incapable of reaching those higher levels of abstraction, analogy, and generalisation which are the root of our biological eminence.

This then is the problem—what constitutes a sufficiently economical specification of the recognition process to enable a machine to be built to

implement it? The real reason for this paper is the assumption that economy requires an ability to generalise from incomplete data, an ability to adapt to new environments, an ability to abstract from large amounts of data, an ability to retrieve information with minimum cost, an ability to use to advantage the constraints of the environment, and an ability to benefit from mistakes and successes. These are key problems in machine intelligence studies. The virtue of the automatic speech recognition problem (referred to henceforth as the ASR problem) is that it brings together a machine for which there is a real commercial requirement, and principles of machine intelligence which will be needed for the implementation. This happy conjunction provides a basis for discussion.

AUTOMATIC SPEECH RECOGNITION—THE PROBLEM

We have said that the problem is to specify the recognition process economically. This involves knowing about the speech signal we wish to recognise in order to specify the processing required by the machine. The most striking aspect of speech, which achieves its full impact when ASR is considered, is appalling complexity. The most effective overall picture of speech in relation to the machine and the recognition process is by means of a block diagram such as appears in Fig. 1. The connecting arrows stand for 'entails', and the circles, with numbers in, are conventional threshold gates, used to compound the entailments. In the top left area of the figure is shown the nature of the speech we wish to recognise. The bottom left area shows the main characteristics of a machine which might be built to recognise the speech. The right-hand side of the figure outlines some of the operations that the machine would need to perform.

The purpose of the next two sections is to amplify the diagram. The fifth section will describe briefly some work which is being carried out at Standard Telecommunication Laboratories towards implementing a real machine, and the final section will contain conclusions.

SPEECH

General

Speech is generated by a human being for the purpose of communicating with another human being, and is usually transmitted and received as an acoustic signal. This, like most other signals, is subject to various sorts of noise and distortion. These disturbances degrade the information in the signal. Because it is generated by a human being it is subject to a number of constraints due to his physiology, his intention and his previous linguistic experience. Finally, that a human being can recognise speech, despite the disturbances, is evidence of redundancy in the transmission/recognition process.

Disturbances

The first disturbance is channel noise, which comprises reduction of information by added noise, by attenuation of the signal (equivalent to

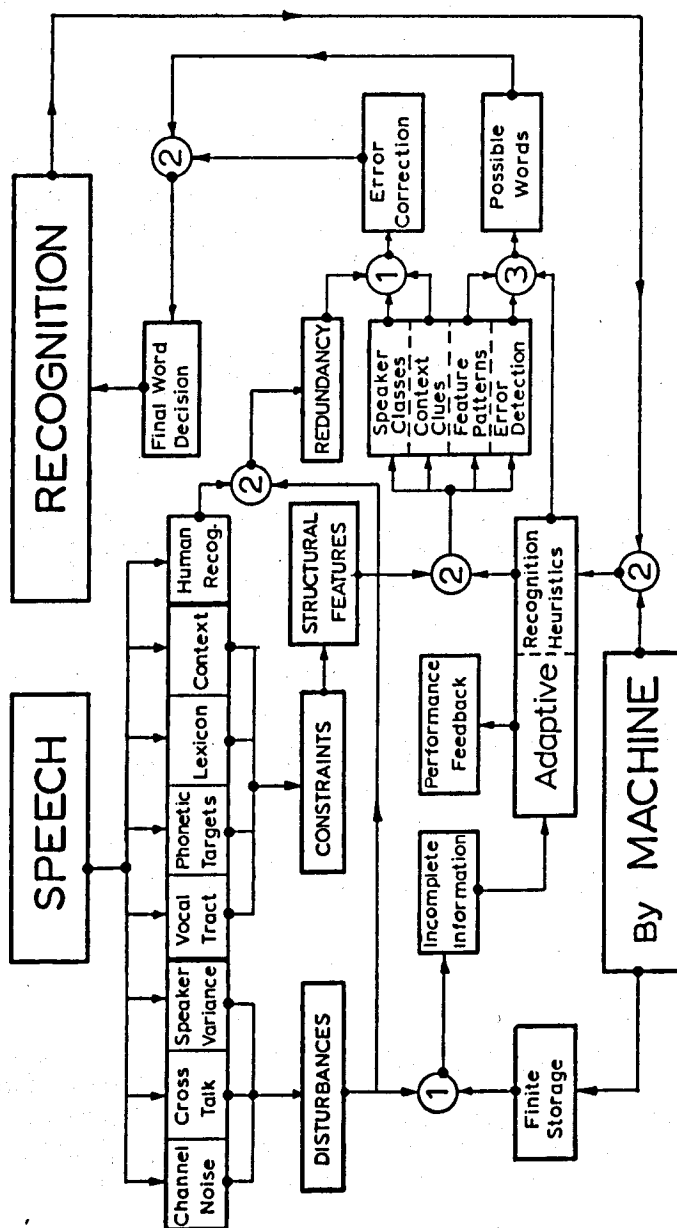


FIG. 1. The ASR problem.

adding noise), or by distortion of the signal because of the nature of the transmission path. If, as is highly probable for ASR, the speech is transmitted through a telephone link the problems of noise and distortion can be quite severe and include noises due to handling the handset, clicks and hisses in the speech band, limitation of the bandwidth to the range between 300 and 3400 cps, and pre-emphasis of the signal. With an air-only transmission path there is distortion because of the acoustics of the environment, and attenuation.

Another form of disturbance is cross-talk. In the most severe form this is referred to as the cocktail party problem. The effect on recognition performance is considerable, for the competing noise has the same general form as the signal. Miller (1947) has shown that a babble of four or more voices is one of the most effective masking noises available, given a single communication channel. The cross-talk mean level on commercial telephone systems (assuming no cocktail party at sending or receiving end!) is normally no worse than 30 dB (voltage) below the signal, which could give a target for machine recognition.

Finally, and this form of disturbance is the one most often considered in ASR studies, there is a great deal of speaker variation. Even the same speaker attempting an identical series of utterances will produce signals which differ noticeably from utterance to utterance (Fry 1959), and the words he utters in isolation will be different from the words he utters as part of connected speech (Truby 1958). Between different speakers there is considerable variation, and, for example, the vowel and fricative categories of one speaker are very likely to overlap different vowel and fricative categories for other speakers (Fry 1959, Stevens 1960). He may also use the same vowel categories in a different way because he has a different accent.

Constraints

At the lowest, most general, level there are constraints on the signal structure because of the nature of the generating apparatus. The human vocal apparatus consists in essence of a main tube through which a modulated stream of air may be blown, and a secondary tube which may be connected in parallel with the second half of the main tube. The configuration of the main tube may be varied by constricting the walls, moving the tongue—or parts of it—back and forth, and up or down. The secondary tube may be connected by lowering the velum. The tongue may produce a sufficiently narrow constriction to produce turbulent noise (fricative sounds) or even cut off the flow of air altogether (stop sounds). If the velum is lowered nasalisation occurs which, with the main air flow stopped by the tongue, leads to nasal consonants. The modulator at the lower end of the main tube may impress periodic (voiced) or aperiodic (aspirated) variation in the air flow, or may stop it altogether to give a glottal stop. In the case of voiced modulation the energy distribution, in terms of frequency versus amplitude, will show a harmonic series which falls off at about 12 dB per octave above

500 cps. The acoustics of this relatively complex vocal system have been the subject of classic studies by Fant (1960). Briefly, however, they are as follows. The source energy excites standing waves in the tract just as standing waves are formed in an organ pipe. These resonances modify the energy distribution in the basic spectrum leading to bands of enhanced energy in the frequency domain called *formants*. In theory there is an infinite series of these *formants* but in practice only five are important (neglecting nasal sounds) and of these only the lowest three are significant for intelligibility. At the frequency of F1 (the first and lowest *formant*) the standing wave pattern is $\frac{1}{4}$ of a full wavelength. For F2, and F3, the standing waves are $\frac{3}{4}$ and $\frac{5}{4}$ of a full wavelength respectively. For an average vocal tract, 17.5 cm long, this leads to formant frequencies of about 500 cps, 1500 cps and 2500 cps. Inside the tract pressure and velocity have a phase difference of 90° , and at the frequency of a formant, there is always a pressure minimum at the lips, while the opposite is true at the larynx (the modulator). The simplest rule governing the relations between vocal tract configurations and F-patterns is that if a uniform tube is constricted at a place where one of its formants has a velocity maximum the formant frequency will decrease; a constriction at a velocity minimum leads to an increase. In addition to the static acoustic constraints on the signal due to the vocal tract, there are dynamic and neurological constraints. The parts of the tract have inertia and particular forms of attachment, and the messages which activate movement have limitations. Thus the rate at which changes occur, and the possible configurations, are restricted in a manner general to human speakers. The fastest changes are due to the velum and larynx, next those due to tongue movements, next those due to pharyngeal constriction, with the slowest changes resulting from lip and jaw movements.

Constraints at a less general level result from the acquired articulatory habits of the speaker, ignoring the differences due to speaker physiology. In attempting an utterance, the speaker will be aiming at certain phonetic targets related to the language and dialect he speaks. These habits of utterance are acquired at an early age, and are difficult to break, as evidenced by the difficulty that speakers of English have in producing really good French vowels.

A further constraint on the speaker is his vocabulary, or lexicon. Speakers of the same language, even the same dialect, will use different vocabularies. Miller (1951) has shown that, taking 50 per cent success as the intelligibility criterion, there is an 18 dB variation in speech power threshold when the choice set for intelligibility testing is changed from 2 to 256 words. Normal people have vocabularies well in excess of this, some estimates put the size as high as 30 000 words, but this figure is drastically reduced if derivatives are not counted as different words.

The last set of constraints are contextual. Some are due to the syntax and semantics of particular utterances, some are situational in that the topic of conversation will restrict the likely use of the vocabulary. The contextual

constraints are hard to partition, since they range from quite general rules about how utterances may be constructed from lexical items, and what is meaningful, to quite particular restrictions on what is likely to be talked about next, and what interpretation should be placed on what was said a few moments ago in the light of what is said now.

Redundancy

That a human being can recognise speech despite the disturbances is evidence that there is redundancy in this form of communication. The redundancy is embodied in all aspects of speech and results from the constraints which exist. Certain signal-component structures simply are not allowable because of the generative and phonetic target constraints, and certain recognition possibilities are not likely due to the lexical, syntactic and semantic constraints. The redundancy can be increased by restricting the lexicon and imposing stricter rules on the construction of utterances from the lexical items.

A very simple recognition machine might work on input strings of the form *number, operator, number*, allowing only twenty or so recognition possibilities. Syntactic constraints would then inhibit the recognition of an operator for the last item, while semantic knowledge would allow the machine to reject the recognition of 'zero' as the last item if the operator happened to be 'divide by'. This level of using syntactic and semantic redundancy already exists in computer programs—it is extending the principle to the whole of language which is likely to prove so difficult.

Structural Features

If there are constraints on the signal, it will have structure; the more restrictive the constraints the less will be the variety of the structure. In speech we wish to describe the structure for two reasons; first so that a general knowledge of the structure may be built into the recogniser to allow it to utilise the redundancy to aid the recognition process, secondly so that the input may be adequately described in terms relevant to the recognition process. The term *structural features* refers to the descriptors or 'clues' which define the structure, so this section is concerned with the description of speech, and the form and nature of the descriptors. Whatever level of description is aimed at, the evidence for the description of a particular utterance must either come from the acoustic signal itself, or be built into the recogniser and be accessed by clues from the acoustic signal. However useful and easy it is to describe an utterance in terms of non-acoustical features, it does not help ASR if these cannot be related to the acoustic signal.

There are two ways of arriving at a description of the structure (analytic and synthetic) and two ways of formulating the description (segmental and parametric). The analytic determination of structure depends on analysis and classification of data from signals exhibiting the structure; the synthetic

approach depends on setting up models of the structure to form the basis of synthesis, followed by testing the goodness-of-fit of the results of synthesis with reality. Very often both methods are used together. The difference between the segmental and parametric formulations is that the former divides the signal with respect to time, using multi-dimensional descriptors, whereas the latter considers temporal variations in unidimensional descriptors. In general the two will be strongly related, for the segmental descriptors may be compounded from the parameters.

In speech there are only two significant segmental approaches below the level of words, one relating to phonemic segmentation, and the other to the detection of distinctive features. Let us start by considering these two segmental approaches in relation to phonetic targets. In the former, sounds of a language are grouped into categories called phonemes, and any utterance in the language may then be represented by an appropriate series of these. Individual sounds are classed in the same phoneme category if the substitution of one sound for the other never distinguishes two words in the language, and so a particular phoneme structure is specific to the language it describes. The phonemes may be classed into a number of broader categories, for instance voiced/unvoiced. During the whole of voiced sounds the larynx generates periodic excitation to the vocal tract, which impresses its configuration on the signal radiated from the talker by modifying the energy distribution in the frequency-time domain. From the point of view of intelligibility the emitted signal has three main peaks of energy (in addition to the peak around the fundamental frequency) somewhere in the range from approximately 300 cps to 3500 cps, and these are formants (see also 'Constraints', p. 203). Higher formants may be observed, and, if the nasal passage is connected to the main resonant tract by lowering the velum, more than three formants may be observed in the lower range, but in what follows *formants* refers to the three main formants. The voiced sounds in normally spoken English include all the vowels together with some of the consonants, such as /m, b, w, z/—these latter being examples of nasal, voiced stop, semi-vowel, and voiced fricative consonants respectively. The vowels are characterised by the steady state values of the formants (which in connected speech may never be achieved) and the consonants by the transitions of the formants to or from a characteristic steady state value (this latter, in the case of the stops, being of negligible or non-existent duration) together with the amplitude, type and duration of concurrent hiss or aspiration noise. The other ends of the transitions will depend on the adjacent phonemes. The voiceless sounds, in general, consist of a period of hiss type noise, generated by forcing the breath stream through some constriction in the tract, preceded and/or followed by formant transitions in the tract modified signal. There may be two distinctly different types of hiss, as in the case of /p/ for instance where one type could occur due to the release and relate to the constriction at the lips during the release, and the other could occur after the release and relate to the constriction at the larynx prior to the onset of voicing for

the succeeding sound. The place of constriction can vary between these two extremes, and the further down the tract the hiss is produced, the more the tract will be able to impress its configuration on the signal by modifying the energy distribution. The basic spectrum of the hiss is characteristic of the place of constriction. The only consistent difference between voiced stops and voiceless stops lies in the time of onset of voicing (Lisker 1965), though the release is usually much more pronounced in initial and medial voiceless stops than in the corresponding voiced stops. There is a close correspondence between the series of voiced consonants and the series of voiceless consonants, of the stop and fricative varieties, but reasonably intelligible speech may be produced without making the distinction, as in whispered speech. This is another piece of evidence for the existence of redundancy in speech, and is in part due to the redundancy of coding with respect to voicing.

For this reason among others, many workers prefer a sub-phonemic segmental analysis of speech sounds. An important contribution to this approach was made by Jakobson, Fant & Halle (1961). Since the phonemes may be dichotomised in a number of ways, the features used for the dichotomies may be used to classify the phonemes and fewer units are likely to be necessary. These units they termed *distinctive features*. Each comprised the opposition of two polar qualities of the same category, or the opposition between presence and absence of a certain quality. A concurrent bundle of distinctive features defined a phoneme. The difficulty with the original distinctive features is that they were largely based on articulatory considerations and are not easy to relate to the acoustic signal. However, variations on the theme of distinctive features lie behind quite a few approaches to ASR, and the engineer, free from linguistic inhibitions (but, hopefully, with some linguistic knowledge) happily modifies the idea to his own ends, and can also use the features to classify whole utterances. Hughes (1961), now at Purdue University, developed a machine which, under certain constraints, could recognise nonsense syllables a little better than a human being, using a modified series of distinctive features. These new features are the most explicit example of the structural features with which this section is concerned, and a particular starting set is suggested in 'The hardware' (p. 219).

Parametric descriptions at this level also exist. The best example of acoustical parameters occurs with Lawrence's (1953) Parametric Artificial Talker (PAT) (Antony *et al.* 1962, Ingemann 1960) and its successors, which approach the problem from the synthetic side. The acoustical parameters used are the three formant frequencies, the rate and amplitude of periodic excitation to the tract, the amplitude of aspiration, and the amplitude and frequency region of hiss noise, a total of eight parameters. In their work on speech by rule, using slightly different parameters, Holmes *et al.* (1965) have shown that generative rules in terms of such parameters can lead to highly intelligible speech. He also showed, while at Stockholm working with OVE II—a synthesiser very like PAT—in Fant's laboratory, that real speech could be copied parametrically accurately enough to be

almost indistinguishable from the original. Holmes' speech by rule, despite the fact that it was parametric, was based on phonemic segmentation, and the rules took care of the continuity. This supports the view expressed on p. 206 that segmental and parametric approaches are strongly related. Particular configurations of, or changes in, the parameters may constitute an event, and the events may provide a more economical description than the parametric description.

Use of the event approach, which is a variant of the distinctive feature approach, has proved the most successful means of attack on the determination of structural features. By systematically varying the events in speech synthesised partly on the basis of events and partly on the basis of parameters, the workers at the Haskins Laboratories have added extensively to our knowledge of the essential structure of speech. A classic interpretive paper by Liberman (1957) summarises the work up to 1957, and nominates events such as spectral quality of sounds at constant constriction, spectral quality in transient sounds—at or near the time of maximum constriction, transitional events in the parameters—indicating movement of the articulators (transitions), and events characteristic of the introduction of the nasal passage. Further data appears in other papers (for example, Liberman *et al.* 1952, 1955, 1957, Lisker *et al.* 1965).

On the analytic side surprisingly little data has been published. The classic study is that by Potter and his colleagues (1947) at Bell Telephone Laboratories. Wells (1963) has made a study of the formants in British English vowels, Lehiste (1962) published a monumental study of allophonic variations in /l, r, w, y, h/ and included whispered speech, Stevens (1960) has investigated the spectra of fricative sounds, and Green (1958) has made an extensive study of second formant transitions. These are a sample of the most informative.

A further class of features deals with the sequence of events. The incorporation of the time clues has provided a constant hindrance to ASR schemes and at a phonemic level of recognition results in the 'segmentation problem'. Most efforts to take time into account have either done what is effectively a template matching procedure, with some allowance for expanding and contracting the time template, or have attempted to quantise time in a manner exactly related to the spectral changes on the grounds that this leads to an identical scaling regardless of the rate utterance. This is by far the most popular basis and is termed segmentation by a stability criterion. When the signal changes from a previous state, to a sufficient extent, it is presumed that some new phonetic target should be evaluated. What seems to escape most workers is that what is really important are the sequential relationships between the parts of the message. If one expects a sequence ABC, and the B gets lost or changes to D, the sequential relationship 'A before C' is still there, which may be sufficient to recognise the group. This strategy, as well as others, is used by people attempting to decipher illegible hand-writing, but here the process becomes a little more explicit.

One can also notice separation of content and order in children learning to speak. Sometimes parts of the intended sound are lost—but what remains is in the right order, and sometimes all the right sounds are there—but the sequential relationships are distorted. Moray at Sheffield University is investigating the relation of sequence and content information to the information processing abilities of the human, though at the slightly higher level of words. Almost no work has been done on sequential features of words. Marril & Bloom (1963, 1965) have tackled the analogous problem of 'rules for combination' for the primitive features used in their 'CYCLOPS' picture-pattern recogniser. Their work stimulated this author to the idea of sequential features for ASR.

We are now in a position to return to a consideration of the features which describe the action of the vocal apparatus. A considerable upsurge of interest in this topic has taken place recently. What has been said before about approaches and formulation is applicable, but the idea behind the interest is that utterances may be more usefully described in terms of articulatory descriptors than acoustic descriptors. On the analytic side, by relating electromyographic measurements to the production of utterances, and classifying the data, it is hoped to formulate descriptors. Truby (1959) is responsible for some of the best cineradiographic pictures taken of the articulatory process, while MIT, in conjunction with the Haskins Laboratories, are examining both kinds of data. Similar work is also being started at Fant's Laboratory at the Royal Institute of Technology in Stockholm. On the other hand, Ladefoged and his colleagues at the University of California, Los Angeles, are not only taking measurements on speakers (see, for example, Fromkin 1964), they are also attempting to build a controllable physical model of the real vocal apparatus of the human. MIT are using a computer controlled electrical analogue of the physiology of the vocal tract, which has about 15 parameters (Lindgren 1965). In England, Abercrombie (1965) has suggested a preliminary set of parameters for describing the vocal generating process and emphasises the difference between the parametric and segmental formulations. He suggests three broad divisions—respiratory parameters, phonatory parameters and articulatory parameters—and, in the last category, includes velic valve action, tongue body movements, tongue tip movements, lip movements and jaw movements.

The descriptors for the lexicon fall into several categories—statistical, syntactic, and semantic. First there is a straight specification of the words in the lexicon, which for the average talker changes with time. Next there is a frequency of usage of those words, also changing with time. The lexicon also specifies phoneme transition probabilities and in addition there will be word transition probabilities. These comprise the statistics. The last two feature or descriptor categories reach up to the last level of description, namely context, for the words in the lexicon may be categorised into a relatively small number of *parts of speech* and they have *meaning*. The statistics may be measured reasonably easily, though again there is a lack

of recent data. Hultzen (1964) has published the most recent study of the transition probabilities of phonemes, based on General American. Ungeheuer, at the University of Bonn, who has made one of the most interesting recognisers so far (Tillman *et al.* 1965), is working on the statistics of German using the whole of Kant's work reduced to punched cards. One difficulty here (working from the printed word) is that there is no ready conversion from the orthography of a language to the spoken word, though dictionaries do specify standard pronunciations, which has enabled Bhimani (1964) to develop rules for the conversion process for some dialects of English. All this is limited to a phonemic segmental formulation and I do not know of any similar work on distinctive features or parameters, except that a dynamic plot of F1 against F2 shows little structuring of data even when duration is used as a weighting factor (Holmes *et al.* 1961).

An interesting illustration of the reality of the characteristic structure of the words of a language is given by the ability of someone like Michael Bentine to produce nonsense utterances which are readily 'identifiable' as belonging to particular languages. Fry *et al.* (1959) and Denes (1959) built such statistics into their phonetic typewriter to demonstrate the resultant improvement in phoneme recognition, with considerable success.

This is the most appropriate stage at which to consider speaker features. Some very successful speaker identification has been achieved by examining details of contour spectrograms (where intensity is given by contour lines rather than intensity of marking), looking, for example, for characteristic shapes of transitions, overall energy distribution and fricative energy distribution. Kersta (1962) originated this 'voice-printing' technique. Sebesteyen *et al.* (1962), at Litton Systems, says features including formant 3 and formant 4 frequencies, rates of change of the formants and first order probability function of the pitch interval are good clues to the speaker. Pruzansky (1963) describes a talker recognition procedure based on comparing the array points in a quantised spectrogram for chosen words, which is literally 'voice-printing'. Speaker features also include the words he uses, and the overall statistics of his acoustic feature production. The speaker features even extend to the syntactic and semantic levels, in the sense that certain speakers tend to say certain things in certain ways. If any features can be used to identify the speaker (or more likely, for reasons of economy, the speaker category) then knowledge of the other features of the speaker, or speaker category, may be used to aid recognition.

Finally, we consider contextual features, which are the descriptors of the rules governing the allowable sequences of words which will constitute *grammatical*, *meaningful* and *relevant* utterances. There are two major ways of looking at the grammar or syntax of a language, either in terms of a phrase structure grammar, or in terms of a transformational grammar. The former is analytic, in the sense that observed data is taken as it stands, but it still has to be fitted to a modelling of the presumed rules of the language syntax and there is a predictive element in it. The latter is much more

closely synthetic, for the syntax of the utterance is described in terms of the transformations required to produce the utterance from a *kernel sentence*. Work by Hanne at the University of Michigan, and by Newcomb at General Dynamics, Rochester, New York (neither published), is typical of the application of phrase structure techniques to ASR. In both these studies the string of sound elements in an utterance is used in conjunction with the lexicon to produce possible strings of words, and with each word is associated the part(s) of speech into which the word can be categorised. The various strings of parts of speech which are thereby possible are compared to the general schema of phrase structures and the impossibilities eliminated. Some ambiguity may still remain which can only be eliminated by semantic considerations. The transformational approach can lead to more accurate descriptions of the syntax of the utterance, but it seems much less suitable for actually applying to ASR directly, as there is the problem of deciding what the kernel sentence should be. Work is in progress at MIT Research Laboratory of Electronics under Chomsky (1957) and Yngve (1960), and under Thorne at Edinburgh University in Britain, to quote only a few examples. The meaning of language is, for the ASR worker, the most forbidding hurdle of all, for it implies not just a knowledge of the real world and what sort of things can and cannot happen in it, but also the whole human ability to use a language as a means of thought. Miller's sentence illustrates one of the difficulties. It starts (and this should be spoken rather than written) 'Writing it rapidly with his undamaged hand he saved from destruction the contents of the . . .'. Thus far we have detected no error, though we may be a little puzzled. However, when it ends with 'capsized canoe', we realise that the word at the beginning of the sentence should have been spelt 'righting', at least this is the most probable intention of the speaker. In attempting to use semantic features, which are the descriptors of the world we live in, machine intelligence will face its greatest test.

THE MACHINE AND THE TASK

General Characteristics

At this point the purpose of this report has perhaps been achieved, and the problem for machine intelligence has been formulated. It would not be fair, however, to pose a problem without at least some suggestion of how the problem might be solved, and what steps are being taken towards implementing the proposed solution. It will be necessary to become rather more particular in order to make these suggestions, and it is with some trepidation that a scheme is put forward at all, for there are many question marks and much that ought also to be considered will be ignored. The scheme will consist of suggestions for machine operations, some described in more detail than others. If the underlying assumptions are correct, and the parts could be designed to perform the operations suggested, this would deal with a fairly complex input of connected speech, with allowance for

PATTERN RECOGNITION

reversion to word by word operation when in difficulties. A primary objective has been to make each descriptive level of operation of the machine as independent of the operation of higher levels as possible. Some degree of recognition may be achieved with a simpler version of the machine, and the higher levels could be added progressively to improve the performance. 'Level' is perhaps a misleading designation for much of the processing will be parallel.

The input to the machine will consist of an acoustic signal representing an utterance. The machine's output will be a code representing the word(s) in the utterance. The input signal is characterised by structural features, embodying redundancy, bearing some relation to the words uttered. The machine may be characterised by a task structure embodying recognition heuristics, and by finite size. If a machine having any degree of generality is to be built with finite storage, especially when there are disturbances on the signal, it will need to be adaptive. Adaption requires performance feed-back to control the adaptive process, so this must be given to the machine, and probably sets a minimum requirement of the recogniser prior to adaption. A block diagram, on which the description is based, is given in Fig. 2.

Description

A commonly used division of pattern recognition machines is into a fixed observation-taking pre-processor, followed by an adaptive or non-adaptive decision-taker. This is a useful division, provided it is understood that where the boundary is drawn depends on which decision is being considered. The only requirement is that adaption in what is being considered as the pre-processor should either consist of logic switching according to previously acquired knowledge, or should be slow enough to be quasi-static with respect to adaption in the decision taker. Each decision stage utilises more knowledge of the constraints in the speech, and, in so doing, reduces the number of bits of information passed on to the next stage.

The speech to be recognised enters the machine by means of a linear transducer (microphone) which reproduces the pressure variations in terms of voltage variations. This signal is analysed by means of filters and special-purpose circuits which determine when certain predetermined features are present and when they are not. These we may call *primitive acoustic features*, or *PAFs* (see 'Structural features', pp. 205-11). They will comprise such features as 'silence', 'relative time of voicing onset', 'vowel quality', 'hiss', 'friction quality', 'relative amplitude', 'relative duration', 'occurrence of transitions', 'voicing', together with some features relevant to speaker identification (see p. 210). There is some evidence that features of this type are extracted by biological systems, though this is not necessarily a good reason for doing the same in a machine. Evans & Whitfield (1961) have demonstrated single cell responses in the primary auditory cortex of the cat to such events as 'tone present', 'tone absent', 'tone starts', 'tone ends',

'tone frequency rising' and 'tone frequency falling'. Hubel & Wiesel (1963) have demonstrated analogous responses in the visual system of cats, and Lettvin and his colleagues (1959) have shown some feature extraction in the frog's visual processing system. Effects have also been observed in psychophysical experiments on humans which could be explained using a similar model.

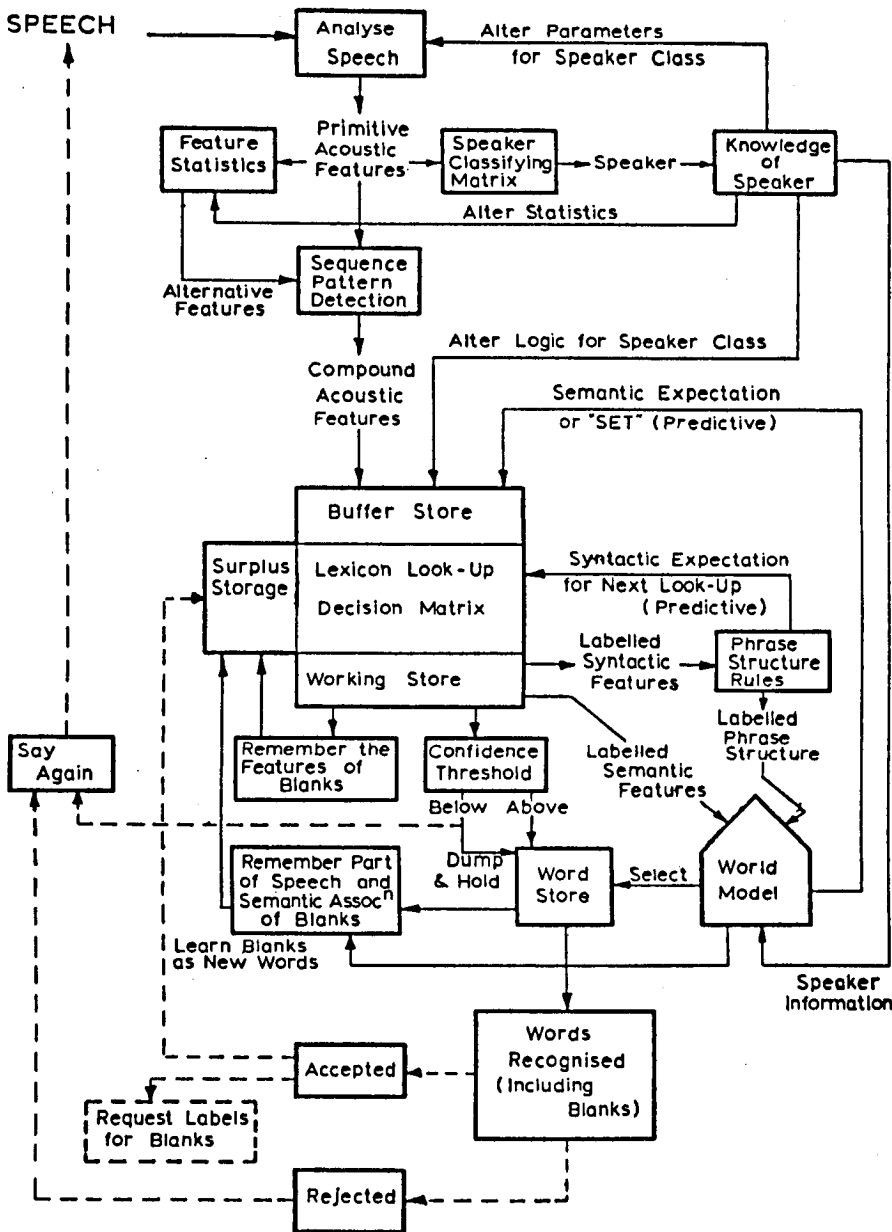


FIG. 2. A hypothetical ASR machine.

The PAFs could provide evidence for recognising speakers. In parallel with the feature extraction process, there could be a predictor. This would embody the statistics of the features, and its predictions would be combined with the features detected in an attempt to take advantage of the known properties of the language to correct errors. The predictor would be very slowly adaptive, if at all, but recognition of the speaker, or speaker type, could lead to a modification of the information used in the predictor. Speaker recognition could also lead to slight alterations in the feature extraction circuit parameters as well as switching of vowel quality channel connections between the detectors and the outputs to allow for dialectical variation in the use made of the vowels. These changes would be based on acquired knowledge of the speaker category characteristics.

Sequential patterns would then be detected in the PAFs on which word decisions were to be based; the output of the detector would be called *compound acoustic features*, or *CAFs*. It seems likely that the CAFs would be rather syllabic in nature and would form the main basis of word decisions. The relation between PAFs and CAFs would be determined as a result of five basic operations on the PAFs—occurrence, simultaneous occurrence, *X or Y*, *X before Y* and iteration. The output would include an indication of the number of occurrences of the CAFs. As indicated on p. 208 some caution would be required in the selection of the particular CAFs to be output. If they are chosen to be too simple they will not be discriminating enough, if they are made too complex then they will be so discriminating that each will be evidence for the occurrence of only one word; either alternative is uneconomical and therefore undesirable. Ideally each should occur in about 50 per cent of the vocabulary, but this is unlikely to be achieved in practice. A further constraint of the CAFs chosen as categories for recognition is the 'necessary and sufficient' condition for word recognition, for however well they are chosen there can still be a very large number of them, and this may be a critical area for the use of adaption to keep the working set as small and useful as possible. What strategy could be used for this is simply not known at present and a great deal of work is required even to assess the feasibility of a manual determination of a sub-set for recognising a small vocabulary. The basis for generating a new CAF would be comparison of the PAFs of a word with the stored CAFs of the words with which it was confused. The merit of the CAF would then have to be evaluated in the light of further new word discriminations which it enabled.

Information concerning the occurrence of CAFs would be passed to the buffer store of the word decision taker. It would be necessary to assume at this stage that any important order information was explicit in the CAFs, and the word decision would therefore be a straightforward maximum likelihood estimate. This is a degenerate form of the Bayesian strategy resulting from the assumptions that correct decisions cost nothing, incorrect decisions all cost the same, and the *a priori* probabilities of the decisions are known. If $P(A)$ represents the absolute probability of A occurring, $P(A/B)$

represents the probability of A occurring—given that B has occurred, E_j is the occurrence of the j th event, and X is a set of elementary observations having binary outcomes, then the maximum likelihood strategy may be summed up in the formula:

$$P(E_j/X) = \frac{P(X/E_j) P(E_j)}{P(X)} \quad (1)$$

The realisation of this is covered in a little more detail on p. 221. If the speaker had been recognised, and there was stored knowledge concerning the lexicon of the speaker—either required changes in the items, or required changes in the features expected for the items—the necessary adjustments could be made. In addition certain semantic features would be fed in concerning the associations of the preceding utterance, to bias the decisions taken in favour of words associated with the previous utterance. These would be provided by a later stage (see p. 216). When there was space in the working store, the new information would be fed successively into the working store and continue the process of generating likelihood estimates on the words. When the likelihood of a word or words exceeded a confidence threshold then a maximum detection would be made and the most likely word would be checked by comparing the stored features 'expected' with the features actually in the working store; those items in the working store actually used would be marked. If the marked items could form part of a longer word (and this information could be stored with the word recognised) then a check would be made of the CAFs immediately following to determine whether the longer word was present or not. If the longer word were detected the process would repeat until the longest word admissible had been determined. Finally a check would be made to see if the longest word determined could be *exactly* split into smaller words. If it could then the alternatives would also be considered as responses and the word(s) would be put into the output store, with label(s). At the same time information concerning the part(s) of speech of the word(s), and the semantic features if any, would be passed to other sections of the machine, suitably labelled to correspond with the word label(s). The marked items would then be discounted for recognition purposes and the process would continue. If there was a gap of unmarked features preceding the recognised item and these were sufficiently unlikely to be associated with a word then a labelled blank would be put into the output store, prior to the recognised word, and the unused features put into a secondary working store with the same label as the blank. These features would then also be marked. If these unknown features led to an equivocal decision then a partial output of the words recognised so far would be made with a request to repeat the doubtful section. If the output included the correct words thus far the operator could merely repeat as requested and the process would continue, the new features being inserted in place of the doubtful string. If the partial output were not accepted the input would have to start afresh. If the input stopped for more

than a certain period a definite boundary would be inserted. Unused features between the last word recognised and the boundary would be treated in the same way as those between two recognised words.

When a definite boundary occurred, or if the output store filled up, a selection of the words in the output store would be made on the basis of information from a parsing program, using in addition any relevant semantic constraints. (At this point the ice becomes even thinner, if it is there at all.) The parts of speech put out to the parsing program would lead to the determination of a phrase structure which fitted one of the limited number of *consecutive* combinations of parts of speech derivable from the given alternatives. Newcomb's program is of this type. The labels associated with the parts of speech would then allow the semantic relationships to be checked on the basis of the phrase structure against the allowable relationships in the machine's model of the world. The only purpose of both these activities would be for the resolution of ambiguities. If no phrase structure could be fitted to the given parts of speech, then there would be no selection from the alternatives in the output store of the machine on this basis. Likewise if there were no semantic homogeneity obtainable by selection, or no semantic inconsistencies detectable, then again there would be no selection on this basis. Both these activities would only be aids to the recognition, and in the absence of any aid all the words in the output store would be output, and the operator would be left to rephrase or correct the utterance in the event of there being ambiguity still present. The machine's model of the world need not be very complex to be of assistance, but even a modest complexity is outside the present state of the art, for reasons both of size and nature of the storage required. (One can only postulate the most trivial constraints for a small vocabulary, such as the one suggested on p. 205.) Any information concerning the likely part(s) of speech and semantic associations of blanks could, however, be stored in an auxiliary store, together with a knowledge of who said it (if known). Any semantic associations would be passed back to the input to the word decision taker, to bias the recognition of succeeding utterances, and the parsing program could pass back syntactic expectation to bias the recognition of the word immediately following the string compiled up to that point.

When the output occurred from the machine the operator could either accept or reject it. If he accepted it, and it contained blanks, the machine would take it that the blanks were new words and would request, one by one—referring to the labels—for the blanks to be named. This operation could consist of spelling out the word, specifying an output channel, or manually interfering with the machine, but in any case the features and other relevant information which had been held in temporary storage would be put into surplus locations in the word decision store (thus displacing least useful items if the store were full) together with the appropriate output connection. Finally the whole output would be made available for whatever

purpose the recognition had been intended. If the output were rejected the machine would ask for a repeat, and it would be up to the operator whether to try the whole utterance again, or proceed on a word by word basis.

Discussion

This section has done little towards proposing concrete ideas on the solution of the ASR problem; it would not be a research topic if it were possible to do so. Instead an attempt has been made to indicate the general lines of a solution to the total recognition problem in the hope that, however tentative, ambiguous and ill-defined the general scheme may be, it will at least give an idea of the complexity of the machine required to recognise a usefully large vocabulary spoken by speakers who differ significantly and use connected speech to converse with the machine. The core of the machine comprises the analyser, the sequence detector and the word recognition matrix with an output requiring verification. This could be non-adaptive at the start and would probably allow recognition of a limited vocabulary for a selected sample of speakers. The first task is to build these parts, or simulate them on a digital computer in order to show that they are feasible, and to allow an evaluation of the recognition performance. Work is already in progress on all these parts and an early version, lacking the sequence detector, was described at the IFIP 1965 Congress in New York (Hill 1966), and is described briefly in 'Building the machine', p. 219. Until the feature extractor is working there is no data which can be used to program the CAFs detection logic and therefore none to train the decision matrix on. The building of the feature extractor is therefore a first goal, though the design may well have to be modified in the light of recognition trials. Some of the features, such as silence, are reasonably easy to detect, but there are very serious difficulties when it comes to the treatment of spectral energy information for vowel quality detection and fricative quality detection. It seems that the processing for a number of features, particularly these, must proceed on a relative basis. Forgie† has recognised both fricatives and vowels for a fair spectrum of talkers using a straightforward statistical decision technique which required two dimensions for the fricatives and three for the vowels, but this has not been repeated by other workers, and the newcomer finds that he must start from scratch.

The other parts of the machine, with the exception of the parsing program rules, are highly speculative, and indicate the general lines of research for many years to come. They show the lines that machine intelligence studies will have to pursue in order to improve the speech recognition machines which will exist in five years' time.

Very little has been said about the details of adaptive processes for inclusion in the scheme. Probably the design of adaptive algorithms represents the most unexplored problem of all for the designer of an ASR machine.

† The two and three dimensions referred to came out during personal discussions with J. W. Forgie.

Adaption will be required both to allow the machine to benefit from past experience, and also to enable it to generalise from the experience over a wide range of potential input. The first objective could be achieved simply by recognising the re-occurrence of the previous situation and making parameter adjustments according to stored knowledge as suggested for speaker identification in the description of the machine. As an example, suppose there was a vowel quality categoriser which, given information about the spectrum of a sound, would classify it into one or other of a number of vowel categories. If it were known from previous experience that a particular speaker had an accent which substituted one vowel quality for another, then, when the speaker was recognised, the substitution could be made for the whole vocabulary simply by changing the connectivity between the categoriser class and the output line. If, on the other hand, it was desired to build an adaptive categoriser, which could take examples of the vowel sounds and build up a generalised categorisation, a far more subtle strategy would be required.

Let us consider a possible strategy. Suppose that the information pertaining to the vowel qualities was presented to a machine in the form of a binary pattern, which we shall call an I-pattern. And suppose that the machine comprised a number of storage rows containing a number of weighted digits equal to the number of digits in the I-pattern together with a category label, and means for comparing the I-pattern with the rows—to detect differences between the weighted patterns stored and a given I-pattern—together with a device which summed the weights of the digits which differed. The device might then operate in the following manner. Initially examples of each vowel quality would be given to the machine, and the patterns would be stored with the weights equal to unity and labels corresponding to the given labels. Further examples of the vowel quality would then be presented to the machine which would attempt to classify them. With each row would be associated a sum of the weights which differed. Some rows would differ, or deviate, more than others and the machine would try to choose the row most similar to the I-pattern. To do this a set latitude would be allowed for the deviation. The labels of the rows whose deviation fell within the latitude would be called 'similar' and would be considered as responses. First the label of the row with the least deviation would be given as a response. If this were correct then the weights of the digits which differed would be decreased and the categoriser would be more likely to make the same generalisation the next time. If it were incorrect then the weights which differed from the I-pattern would be increased, so that the response would be discriminated against, and then the row having the next smallest deviation would be tried, iterating the process. If none of the rows within latitude had the right label, then a new pattern-label pair would be stored in unoccupied storage. If the store were full then the least useful row would be displaced and the new row stored in its place. The usefulness of a row would be determined by two factors. First, how often it had been used successfully

(useful age) and, secondly, how often it had been used successfully compared to the number of times it had been used unsuccessfully (reliability). This would lead to competition between the rows and ensure a tendency to keep the stored patterns up to date while retaining reliable useful patterns. The amounts by which the row pattern weights were decremented and incremented, and the balance between the measures of the usefulness of a row would be critical parameters for the success of this adaptive strategy, which was inspired by one of those in Andreae's STELLA machine (Gaines *et al.* 1966) and is related to the maximum likelihood strategy.

BUILDING THE MACHINE

General

A recogniser is being implemented at Standard Telecommunication Laboratories. It represents a first step towards the sort of recogniser outlined above, and will comprise the analyser, the sequence detector, and the word decision matrix. Parallel work on adaption is also being carried out, some directed at specific ASR problems. What is being built is the greater part of the analyser. The remainder, including adaptive vowel recognition strategies, and similar complex feature extractors required for the analyser, will be simulated. The machine has been described (Hill 1966), but without the sequence detector. Until the feature analyser is working little can be done towards implementing the sequence detector, since some knowledge of the features statistics is required as a basis for design.

The hardware

The analyser may be regarded as a very special-purpose analogue to digital converter. There is a more or less permanent basic system (i) for providing power, (ii) for sequentially storing the outputs of the feature extracting circuits, (iii) for providing a real time, broad band frequency analysis of the input signal, and (iv) for providing other ancillary equipment such as amplifiers, a tape punch for output, a visual display of stored information, monitoring of signal levels and the like. Using this framework, various feature extracting circuits may be tried out, and their effectiveness evaluated in conjunction with the computer simulation. It will also be possible to use the machine to provide binary coded data to the computer in the cases where the complexity of the feature decision makes it more profitable. The sequence of the features and data will not be determined in detail, instead various sampling techniques may be tried out and evaluated. For instance pulses could be generated at a relatively slow rate according to the broad structure of the word and note taken only of the pulse interval into which a feature, or group of features, fell. Thus sequence information within the pulse intervals would be lost, and also the effects of variation in the rate of utterance would be reduced before any decision-taking started. Two trial schemes of this nature are being evaluated at first, one based on

PATTERN RECOGNITION

the amplitude envelope of the word, pulses being generated at the beginning and end of a rise, and at the beginning and end of a fall in amplitude, and the other on the distinction between high and low frequency energy sounds. Any of the pulses may be used in combination to control the sampling.

The features being extracted at first are relative duration, voicing, friction noise, fricative quality, vowel quality, transitions of energy bands, silence and relative amplitude. Fig. 3 shows a block schematic of the feature extractor.

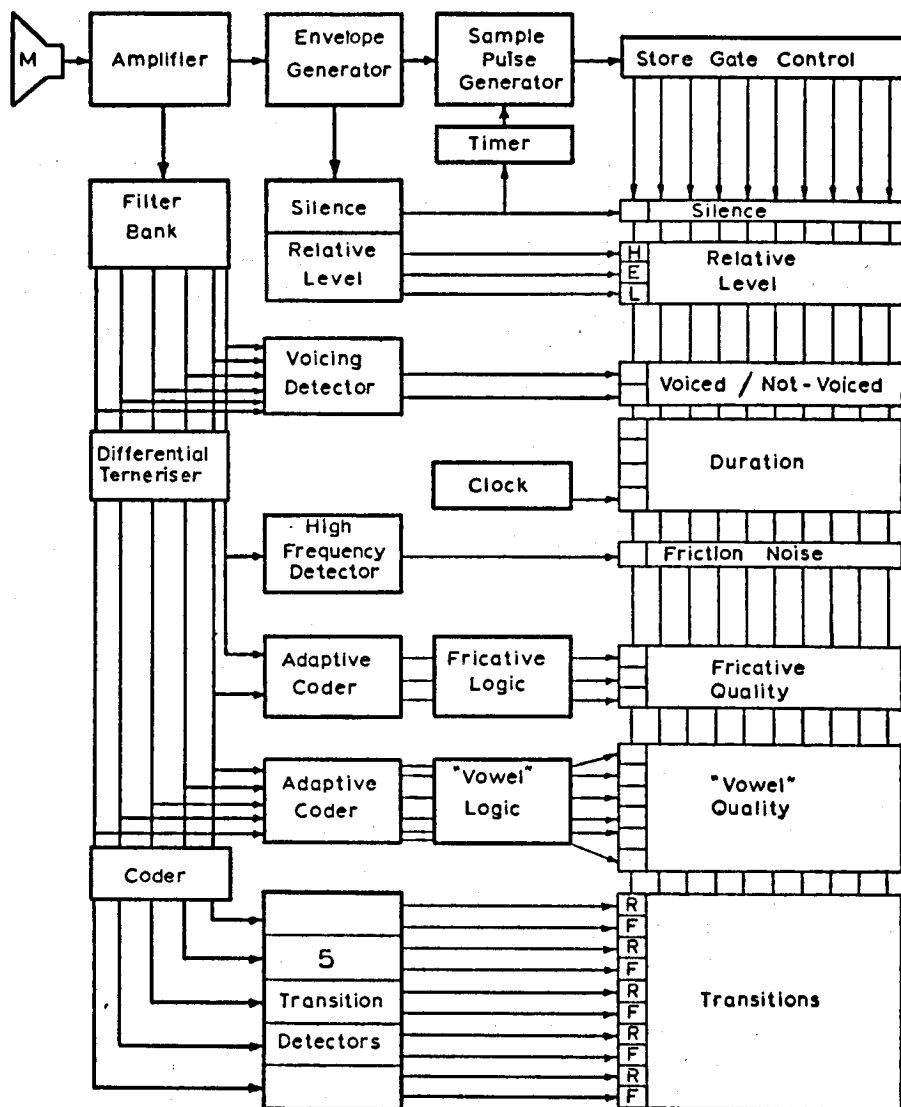


FIG. 3. A processor schematic.

The computer simulation

The rest of the machine will be simulated on a digital computer. One important variation from the scheme described at the IFIP congress concerns the manner in which the sequential features are taken into account, the revised scheme operating along the lines suggested on p. 214.

Where decisions are to be taken on binary data a variant of the maximum likelihood estimate will be used. Referring to equation (1) (p. 215) we derive a second equation similar to the first, concerning the probabilities when the event E_j does not occur:

$$P(\bar{E}_j/X) = \frac{P(X/\bar{E}_j) P(\bar{E}_j)}{P(X)} \quad (2)$$

then dividing (1) by (2) $P(X)$ is eliminated. Making the usual independence assumption on the features, and using an abbreviated notation we may derive the following formula:

$$R_{post} = R_{prior} \prod_{i=1}^n (W_{ij}) \quad (3)$$

where R_{prior} is the *a priori* probability ratio of the event to the NOT-event, R_{post} is the *a posteriori* probability ratio (i.e., that, given the set of observations), and W_{ij} is the ratio of the probability of observing the i th feature—given the occurrence of the event—to that given the NOT-event. These probabilities may be estimated empirically. Thus the *a priori* probability that the event will occur, rather than not occur, is modified by such positive evidence as is available, on the basis of previous observations of the probability of observing elements of that evidence under the occurrence and non-occurrence of the event, to give the *a posteriori* probability that the event has occurred rather than not occurred. This is the general decision strategy which is employed, and the strategy outlined on p. 218 is related, given suitable incrementing and decrementing rules for the weight changes. The results obtained from the computer simulation could be embodied in hardware as indicated in Fig. 4, where logs are taken to facilitate the computation.

CONCLUSIONS

The first set of conclusions to be drawn concerns speech. We conclude that, although individual utterances can be described very accurately by means of present analytical techniques and synthesis can be equally accurate, there is a lack of means for general description of *all* utterances of the same class, where 'same class' means 'leading to the same, or a strongly similar, phonetic transcription'. For this reason there is currently much emphasis on physiologically based description of speech, on the grounds that a description of the generating process should be simpler than describing in general terms what may be generated. This approach will not be fruitful

PATTERN RECOGNITION

for ASR unless either physiological parameters are given to the recogniser (Hillix 1963) or such physiological parameters can be obtained from the acoustic signal. The former is not reasonable for most ASR applications, and the latter is likely to prove as difficult as recognising words using the acoustic signal. Approaches at the physiological level may, however, lead to a better understanding of the underlying structure of the acoustic signal,

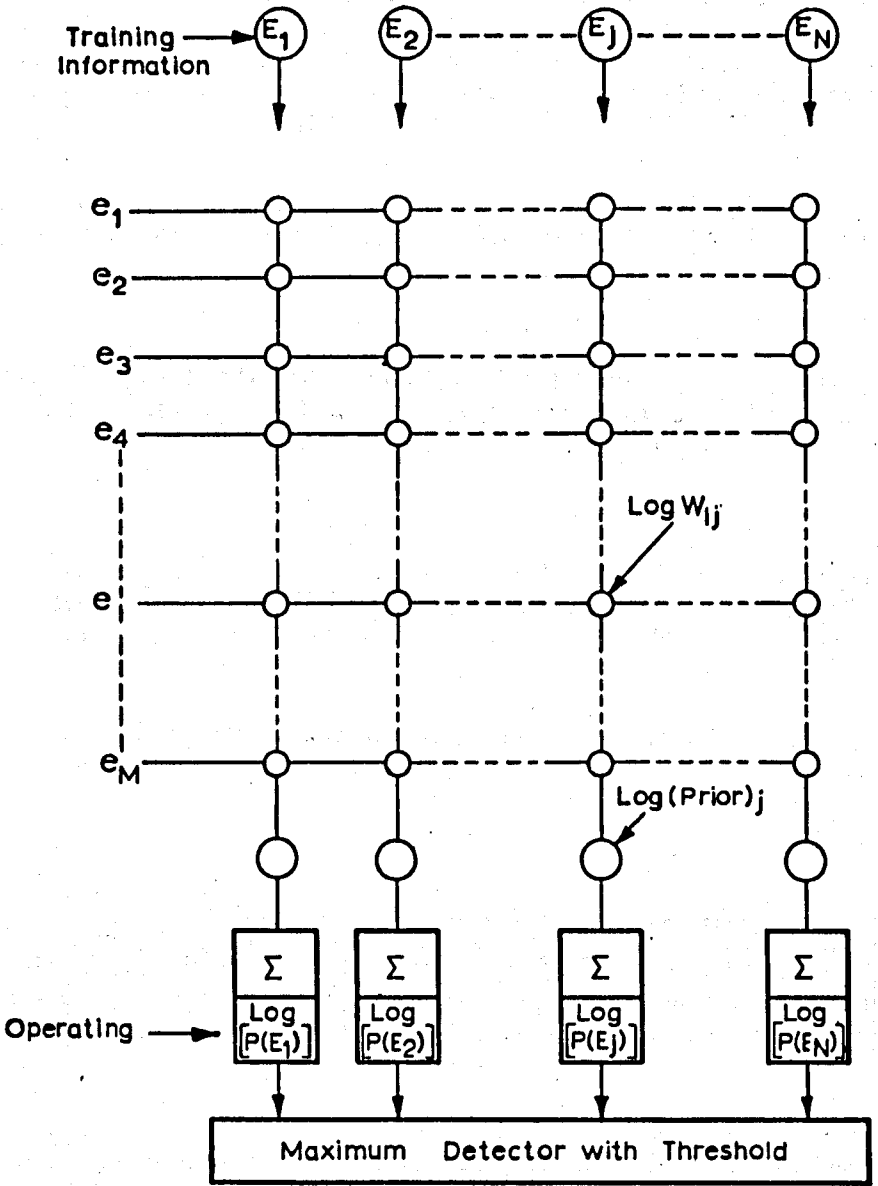


FIG. 4. Decision matrix.

and hence improve our acoustic description. Better descriptions of speech at all levels are essential to progress in ASR, both to describe the input signal prior to recognition, and to describe the general structure of speech so that useful knowledge of it may be built into an ASR machine. Above all there is an immediate need for quantitative work on speech data and statistics at all levels of description, based on descriptors considered important at present. Only by attempting recognition on present knowledge can the inadequacies of this knowledge, with respect to ASR, be fully illuminated.

Conclusions concerning the machine are even more difficult to draw. Certainly we do not know at present how to build a machine to recognise even the digits for just any speaker who happens by, though perhaps on such a small set of words *ab initio* training of the recogniser for each new speaker would be acceptable. Admittedly this would not be recognition in the sense that humans recognise the words of new talkers. Such an *ad hoc* solution should, however, be completely independent of the language of the talker, which perhaps lies behind Tribus' claim† that Dartmouth College's GE 235 will have an ASR input for any language within two years. To recognise a large vocabulary, when, for one reason or another, *ab initio* training would not be possible, a more powerful approach is required. It is likely that a first, primitive, recognition scheme, deciding entirely on the basis of acoustic phenomena, must be implemented. This will require good descriptions of speech at the acoustic level, a good description of the sequential properties, or 'grammar', of these primitive acoustic features, and a means for taking decisions, on the basis of this evidence alone, as to what word(s) could have led to the evidence. On to this basic structure must be added procedures which utilise inbuilt knowledge of the general structure of the speech signal at all the levels suggested above, in order to detect and correct errors, and to resolve ambiguities. The gradual introduction of these higher level procedures, and the embodiment of necessary adaption, will form the basis of ASR research over the next decades and constitutes the problem for machine intelligence.

ACKNOWLEDGEMENTS

The author wishes to acknowledge his debt to the many people who contributed directly and indirectly to the completion of this manuscript. In particular he would like to thank his colleagues at Standard Telecommunication Laboratories for helpful criticism and guidance, especially Dr J. H. Andreae who read the original manuscript and made many valuable suggestions at all stages of the preparation. The author has had fruitful discussion with many of the authors quoted, and would like to acknowledge his debt to them, while accepting full responsibility for any misunderstandings and misquotations which remain.

† Verbal statement during time-sharing, multi-access, computer demonstration at the Heriott-Watt College, Edinburgh, September 21, 1965.

REFERENCES

- Abercrombie, D. (1965). Parameters and phonemes. In *Studies in Phonetics and Linguistics*. Oxford: Oxford University Press.
- Antony, J., & Lawrence L. (1962). A resonance analogue speech synthesiser. *Proc. 4th ICA Copenhagen*.
- Bhimani, B. V. (1964). Multidimensional model for automatic speech recognition AD 437 324 (1964). (This is a rather general reference to his speech recognition work, specific papers on the orthography-to-pronunciation programme, apart from abstracts to the 1965 Washington meeting of the ASA which bear some relation, have not apparently been published. The author's main source was from personal discussion.)
- Chomsky, N. (1957). *Syntactic Structures*. The Hague: Mouton.
- Clapper, G. L. (1963). Digital circuit techniques for speech analysis. *Instn. elect. Engrs. Trans. Comm. Electronics*, **110**, 296-305.
- Denes, P. (1959). The design and operation of the mechanical speech recogniser at University College, London. *J. Inst. Rad. Eng.*, **19**, 219-234.
- Evans, E. F., & Whitfield, I. C. (1961). Classification of unit responses in the auditory cortex of the unanaesthetised and unrestrained cat. *J. Physiol.*, **171**, 476-493.
- Fant, G. (1960). *Acoustic theory of speech production*. 'S-Gravenhage: Mouton.
- Fromkin, V. (1964). Parameters of lip position. In *Working Papers in Phonetics*. University of Los Angeles, California. (Also as 'Lip positions in American English vowels', *Language and Speech*, **7**, 15-21.
- Fry, D. B. (1956). Perception and recognition in speech. In *For Roman Jakobsen*, pp. 169-173. The Hague: Mouton and Co.
- Fry, D. B. (1959). Theoretical aspects of mechanical speech recognition. *J. Brit. Inst. Radio Engrs*, **19**, 211-218.
- Fry, D. B., Denes, P., Blake, D. Y., & Uttley, A. M. (1959). An analogue of the speech recognition process. In *Proc. Symp. Mechn. Thought Processes*, **1**, 375-395. London: HMSO.
- Gaines, B. R., *et al.* (1966). A learning machine in the context of the general control problem. (Not published.)
- Green, P. S. (1958). Consonant vowel transitions. *A spectrographic study*. *Studia Linguistica* **12**, pp 57-105.
- Hill, D. R. (1966). STAR—a machine to recognise spoken words. *Proc. Int. Conf. Inf. Process. 1965 Congress*, Vol. II, Spartan/MacMillan.
- Hillix, W. A. (1963). Use of two non-acoustic measures in computer recognition of spoken digits. *J. Acoust. Soc. Amer.*, **35**, 1978-1984.
- Holmes, J. N., & Shearme, J. N. (1961). An experimental study of the classification of sounds in continuous speech according to their distribution in the formant 1-formant 2 plane. *Proc. 4th Int. Cong. Phonetic Sciences*. Helsinki 1961, Mouton 1962.
- Hubel, D. H., & Wiesel, T. N. (1962). Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex. *J. Physiol.*, **160**, 106-154.
- Hughes, G. W. (1961). The recognition of speech by machine. *MIT Tech. Report No. 395*. ASTIA AD 268 489 (does not cover subsequent results referred to in text but indicates Hughes' general approach).
- Hultzén, L. S., Allen, H. D., & Miron, M.S. (1964). *Tables of Transitional Frequencies of English phonemes*. U. Illinois Press.

- Ingemann, F. (1966). Eight parameter speech synthesis. *Edinburgh University Phonetics Department Progress Report*, Sept.-Dec. 1960.
- Jakobson, R., Fant, C. G. M., & Halle, N. (1961). *Preliminaries to speech analysis*. MIT Press (4th printing).
- Kersta, L. G. (1962). Voiceprint identification. *Nature*, 196, 1253-1257. Also *Bell Monograph*, 4485.
- Lawrence, W. (1953). The synthesis of speech from signals which have a low information rate. In *Communication Theory*, ed. Willis Jackson. Butterworth.
- Lehiste, I. (1962). Acoustical characteristics of selected English consonants. *ASTIA Report AD 282 765* (1962).
- Lettvin, J. Y., Maturana, H., McCulloch, W. S., & Pitt, W. (1959). What the frog's eye tells the frog's brain. *Proc. Inst. Radio Engrs*, 47, 1940-1951.
- Lieberman, A. M. (1957). Some results of research on speech perception. *J. Acoust. Soc. Am.*, 29, 117-123.
- Lieberman, A. M., Delattre, P., & Cooper, F. S. (1952). The role of selected stimulus variables in the perception of unvoiced stop consonants. *Am. J. Psychol.*, 65, 497-516.
- Lieberman, A. M., Delattre, P., & Cooper, F. S. (1955). Acoustic loci and transitional cues for consonants. *J. Acoust. Soc. Am.*, 27, 769-773.
- Lieberman, A. M., Gerstman, L. J., Delattre, P., & Cooper, F. S. (1957). Acoustic cues for the perception of initial /w, j, r, l/. In *English Word*, 13, p. 24.
- Lindgren, N. (1965). Machine recognition of human language (in three parts). *Instn. elect. Engrs, Spectrum*, April/May/June.
- Lisker, L., & Abramson, A. S. (1965). Stop categorisation and voice onset time. *Proc. 5th Int. Cong. Phonetic Sciences*, pp. 389-391. Karger: Basle. 1965.
- Marril, T., & Bloom (1963). CYCLOPS—a second generation recognition system. *Proc. AFIPS Fall Joint Comput. Conf.*, pp. 27-33.
- Marril, T., & Bloom (1965). CYCLOPS—2 system. *Comput. Corp. Amer. Tech. Rep. RT65-RD1*.
- Miller, G. A. (1947). The masking of speech. *Psychol. Bull.*, 44, 105-129.
- Miller, G. A. (1951). The intelligibility of speech as a function of the context of test materials. *J. Exp. Psychol.*, 41, 329-335.
- Miller, G. A. (1962). Decision units in the perception of speech, *Inst. Radio Engrs Trans. Inf. Theory*, 8, 81-83.
- Miller, G. A. (1964a). The psycholinguists (on the new science of language). *Encounter*, 23, 29-37.
- Miller, G. A. (1964b). Communication and the structure of behaviour. *Disorders of Communication*, 42, 29-40.
- Miller, G. A. (1965). Some preliminaries to psycholinguistics. *Am. Psychol.*, 20, 15-20.
- Potter, R. K., Kop*, G. A., & Green, H. C. (1947). *Visible speech*. New York: van Nostrand.
- Pruzansky, S. (1963). Pattern matching procedure for automatic talker recognition. *J. Acoust. Soc. Am.*, 35, 354-358.
- Sebesteyen, G., et al. (1962). Voice identification, general criteria. *ASTIA Rept.*
- Stevens, P. D. (1960). Spectra of fricative noises in human speech. *Language and speech*, 3, 32-49.
- Tillman, G. H., Heike, G., Schnelle, H., & Junghever, G. (1965). DAWID I—ein Beitrag zur automatischen 'Spracherkennung'. *Proc. 5th Int. Cong. Acoustics*. Liege.
- Truby, H. M. (1958). A note on visible and indivisible speech. *Proc. 8th Int. Congress Linguistics*, pp. 393-400. Oslo University Press.
- Truby, H. M. (1959). Acoustico-cineradiographic analysis considerations with

PATTERN RECOGNITION

- especial reference to some consonantal complexes. *Acta Radiol. Suppl.*, **182**, complete. Stockholm.
- Wells, J. C. (1963). A study of the formants of British English. *Progress Report of the Phonetic Lab.*, University College, London.
- Yngve, V. H. (1960). A model and an hypothesis for language structure. *M.I.T. Tech. Report*, 369, 444-466.

PROBLEM-ORIENTED LANGUAGES

SIMPLY PARTITIONED DATA STRUCTURES: THE COMPILER- COMPILER RE-EXAMINED

R. A. BROOKER and J. S. ROHL

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF MANCHESTER

Languages such as Fortran and Algol are concerned with fairly simple types of data, mainly *integers* and *reals*, and with rectangular arrays of these quantities. In this paper we consider some of the problems which arise with more complex types of data and data sets. Interest in this aspect of computer languages arose partly from a reappraisal of the Compiler-Compiler (Brooker, MacCallum, Morris & Rohl 1963, Rosen 1964) and partly from the need to introduce further numerical and non-numerical types of data into our algebraic compiler. Some features of the Compiler-Compiler, particularly the facilities for processing the tree produced by the syntax analysis, are useful enough to be included in a general purpose programming language. It is also relevant to the implementation of some of the facilities of Iverson's *Programming Language* (Iverson 1962).

The discussion relates to a simple language model with an Algol-like nested block structure. We assume that the reader understands what is meant by the 'scope' of a declaration in this context. Ignoring input-output there are four main types of statement in the language:

- data declarations,
- assignments,
- control statements,
- procedure declarations.

The distinction between them is not always clear cut. For example a conditional expression can be regarded both as a control statement and an arithmetic assignment. We propose to ignore these niceties, however, and concentrate on the first two classes of statement.

DATA DECLARATIONS

These can take three forms

$$\langle \text{type} \rangle \left\{ \begin{array}{c} : \\ \text{name} \\ \text{type} \end{array} \right\} \langle \text{name list} \rangle$$

(1) The first of these

$$\langle \text{type} \rangle : \langle \text{name list} \rangle$$

is the most familiar. It allocates storage space for a set of objects of the given $\langle \text{type} \rangle$, these objects to be known by the names in the list. (The $:$ can be omitted if no ambiguity results.) For example,

real matrix (m, n) u, v, w

allocates space to three $m \times n$ matrices with **real** elements. The matrices are to be referred to as u, v, w . In general the $\langle \text{type} \rangle$ has both a static and a dynamic component. In this case, if m, n are computed quantities, storage allocation cannot take place until the statement is 'encountered' at run time. The usual solution is to reserve space in the stack, or section of stack, associated with the current block. The dimensions thus constitute the dynamic component of the $\langle \text{type} \rangle$; the static component, i.e., the part known at compile time, being the fact that it is a **real matrix**. If the language restricts one to assigning constant dimensions, for example,

real matrix (10, 10) u, v, w

then everything is known at compile time, although it may still be convenient to allocate storage space dynamically. (Note: the Algol declaration

real array a, b, c [1 : n], u, v, w [1 : m])

is simply a multiple declaration in which a common, static, component has been factored out.)

Another way to specify $\langle \text{type} \rangle$ is by reference to a previously declared name. For example,

$u : a, b, c$

allocates storage space for three objects a, b, c of the same $\langle \text{type} \rangle$ as u , the dynamic characteristics, if any, being those which u possesses when the declaration is encountered at run-time. In this case, of course, u must refer to an actual object, but if the $\langle \text{type} \rangle$ is purely static then u can be a name defined as in (3) below.

(2) The second form of declaration is

$$\langle \text{type} \rangle \text{ name } \langle \text{name list} \rangle$$

and will be referred to as a **name list**.

For example, **t name** p, q, r means that the identifiers p, q, r will be used to name objects with the same static characteristics as t . Thus if t has previously been declared a **real matrix**, then p, q, r can be dynamically assigned (see later) the addresses of similar objects (though not necessarily with the

same dimensions, if these are a dynamic characteristic). Similarly **real name** p, q, r would mean that p, q, r are to be used for the names of **reals**.

(3) The third form of declaration

$\langle \text{type} \rangle$ **type** $\langle \text{name list} \rangle$

attributes the given static $\langle \text{type} \rangle$ to the names in the list. It cannot be used for a $\langle \text{type} \rangle$ possessing dynamic characteristics. For example, **real type real** does not allocate any storage space but allows us to write *real* : a, b, c as an alternative to **real** a, b, c provided it is within the scope of '*real*'. By defining static types in this way one may save a good deal of underlining. The main purpose, however, in introducing this form of declaration is a theoretical one—to allow names to be given the characteristics of an object without at the same time allocating space for the object itself.

ASSIGNMENTS

(a) Data evaluation:

Examples of statements for evaluation and assigning data are:

$x := y := z := 1 + \cos(2 \times \pi i \times x/n)$	Algol
$z := 1 + \text{Cos}[2\pi i \ x/n]$	CPL
$2\pi x/n \cos + 1 \Rightarrow x \Rightarrow y \Rightarrow z$	Anon

We use the last of these styles for our illustrations. The assignments will be called value assignments to distinguish them from those of the next section.

(b) Name assignment:

We shall sometimes need to manipulate data names. For example, given the declarations

real x ; **complex** z ; x **name** y

the statement

$z \text{ im} \Rightarrow x$

transfers the imaginary part of z to x

but

$z \text{ im} \Rightarrow y$

transfers its **name** to y . The latter operation allows y to be used as a name for the imaginary part of z , so that, for example,

$y \Rightarrow x$

is then equivalent to

$z \text{ im} \Rightarrow x$

Similarly $x \Rightarrow y$ allows y to be used as an alternative name for x . One may only transfer names in this way if the source and destination have the same static type characteristics.

Why should we wish to make name assignments? One use of the facility is simply to change a name, possibly to introduce an abbreviation, for example:

real *temperature*; *temperature* **name** t
 $\text{temperature} \Rightarrow t$

PROBLEM-ORIENTED LANGUAGES

Related to this is the use of **name lists** and **value lists** to define the formal parameters of a routine, for example,

```
routine <name> integer name i; real a, b;
begin ..... end
```

The formal parameters are those between the routine <name> and the first **begin**. A call for this routine might be

$\langle \text{name} \rangle (n, u+3, y)$

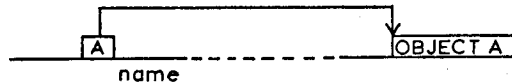
The calling mechanism consists of making name or value assignments between the actual parameters and the corresponding formal parameters according to whether the latter are **names** or **actual locations**. In this case they amount to

$n \Rightarrow i; u+3 \Rightarrow a; y \Rightarrow b$

Another and important use of name assignments is to record as intermediate quantities names which would otherwise need to be recomputed, possibly at some considerable cost, for example, array elements $a(i, j, k)$. Such a need is likely to arise when analysing deeply structured data such as trees.

PHYSICAL REALISATION

The names in a **name list** are represented by cells which hold the addresses of the objects they refer to, for example,

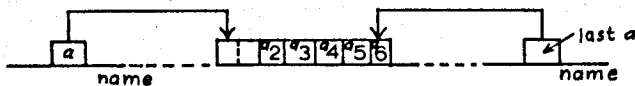


It is convenient to assume, for the sake of uniformity, that all names including those given to objects when they are first declared refer to the associated objects in this way. (It is not, of course, conducive to efficiency, since it introduces an extra store access operation.)

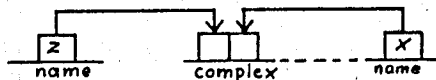
The object shown above is pictured as occupying a consecutive set of storage locations. This need not always be the case provided the structure contains within itself the information necessary for the compiler (with the aid of the static type information) to construct appropriate processing instructions.

Two examples illustrate the physical connections established as a result of name assignment:

(i) **real array** (2 : 6) *a*; **real name** *last a*; $a(6) \Rightarrow \text{last } a$



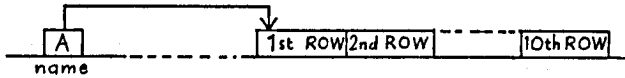
(ii) **complex** *z*; **real name** *x*; $z \text{ im} \Rightarrow x$



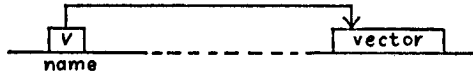
The data structures used in these examples can easily be partitioned into their basic units (real numbers). In designing more complex structures, however, it is necessary to bear in mind how we shall want to partition them. Consider, for example, the case of a matrix and a vector

matrix $A(10, 10)$; vector $v(10)$

If the matrix is stored 'by rows' we have



while the simplest representation of a vector is

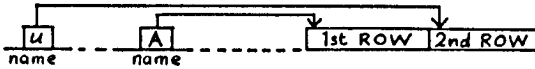


(It is assumed in both cases that dimensions are constant and form part of the static type characteristics.)

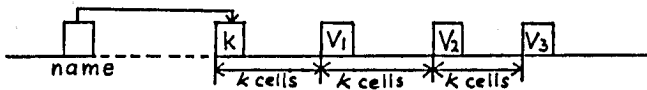
The result of the name assignment given by

$v \text{ name } u; A(2, *) \Rightarrow u$

is to copy the address of the 1st element of the 2nd row of A into the name cell u , thus



Note that we cannot write $A(2, 0) \Rightarrow u$ because $A(2, 0)$ is the name of an element not a vector. For this reason a language involving operations of this kind (e.g., Iverson's *Programming Language* (Iverson 1962)) requires a distinguishing notation such as that used here. The representations just described do not permit a matrix to be partitioned (in our sense) into column vectors. We can overcome this by representing a vector as follows:



The name cell points first to a cell containing an integer k which gives the spacing of the elements, i.e., k is added to the address of each element to get that of the next. Similarly the first element, v_1 stands k cells beyond that of k itself. If $k=1$, the elements stand in consecutive cells.

If now a matrix is bordered by a row and a column, thus

0	$n+1$	$n+1$	.	$n+1$
1	a_{11}	a_{12}	.	a_{1n}
1	a_{21}	a_{22}	.	a_{2n}
.	.	.	.	.
1	a_{m1}	a_{m2}	.	a_{mn}

and then stored by rows in the usual way, both the rows and columns appear

PROBLEM-ORIENTED LANGUAGES

as particular cases of the vectors just described, with $k=1$ and $n+1$ respectively.

Consider, for example, the case of a 3×3 matrix.

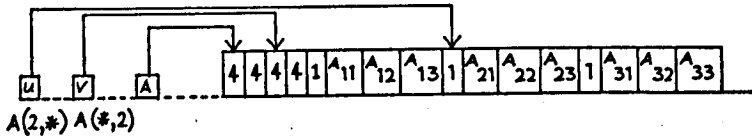
matrix $A(3, 3)$; vector $a(3)$; a name u, v

The assignments

$$A(2, *) \Rightarrow u$$

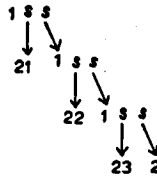
$$A(*, 2) \Rightarrow v$$

name the 2nd row and column of A as u and v respectively. The physical connections involved are as follows:



TREES

Another example of a data structure that can be simply partitioned is a tree. Consider, for example, the structure



which results from analysing

u, v, w

with reference to the phrase $\langle \text{variable list} \rangle$ defined as follows:

phrase *variable list* = *variable* . *rest of variable list*

phrase *rest of variable list* = ' , ' . *variable* . *rest of variable list*, nil

phrase *variable* = 'a', 'b', 'c', . . . , 'x', 'y', 'z'

The second of these is a recursive definition.

Here '.' denotes concatenation and ',' is used to separate alternative formats. These are listed in order of preference, i.e., it is the order in which the recognition routine tries to match them with the source string. If a match is found the remaining alternatives are abandoned. For further details of this algorithm see Rosen (1964). These 'phrase definitions' might be better called 'tree definitions' and in general they could be used to declare trees occurring in any application, and not just trees obtained in a compiler by syntax analysis.

The format eventually chosen is indicated by the numbers in the analysis diagram, 1 denoting the first alternative, 2 the second, and so on. If there is only one alternative the number is 1. The 'S's are address words which point to subtrees. Each node of the tree thus consists of an integer word followed, if necessary (i.e., if the structure goes any deeper), by S words.

These words at each node are contiguous in the store but the nodes themselves can be located anywhere in the store. However a tree is usually stored in a compact form when it is first produced.

A single address is all that is necessary to locate the subtree associated with any particular phrase. With the help of the associated phrase definitions the subtree yields to the compiler all necessary information about the phrase. We shall now describe some operations for analysing the structure of a given tree. Consider the routine:

```

routine read variable list; variable list name vl; begin
  variable name v; rest of variable list name rovl
  resolve vl into v. rovl
  1: read variable (v)
  → 1 if rovl resolves into '; v. rovl
end

```

There are three local **name** locations associated with this routine. First the parameter *vl* which on entry points to a tree with the structure of *<variable list>*, i.e., it points to the 1 at the head of the tree. The other **name** locations will be associated with *<variable>* and *<rest of variable list>*.

The operation

resolve *vl into v. rovl*

isolates the *S* words which point to the two principal subtrees, *<variable>* and *<rest of variable list>*, and records them in the **name** locations *v* and *rovl* respectively. The operation is performed with the help of a 'template' constructed by the compiler from the original phrase definitions, which are indicated in the **name** declarations. As a result *v* and *rovl* now point to the principal subtrees. The process can be repeated to resolve these subtrees into further subtrees and so on. Before we do this, however, consider the instruction

read variable (v)

This is a call for a routine

routine *read variable; variable name v; begin*

B O D Y O F R O U T I N E

end

(Note: the use of *v* in both the call sequence and the routine heading is purely a coincidence.)

The body of the routine processes a *<variable>* tree in some way. We have in mind a 'language' for which we are writing a 'compiler', and that *read (<variable list>)* and *read (<variable>)* are input statements in this language. The task of these routines is to compile instructions to read the data in question. The former calls on the latter to compile the equivalent

of *read* ($\langle \text{variable } k \rangle$) for each of the variables in the list. How the latter routine works is irrelevant for the present discussion.

To return to the main routine, the operation

$\rightarrow 1$ if *rovl* resolves into $\langle \cdot \rangle$. *v. rovl*

is a conditional version of the first instruction. If *rovl* has the first of the two alternative formats in the definition of $\langle \text{rest of variable list} \rangle$, the instruction locates the subtrees corresponding to $\langle \text{variable} \rangle$ and $\langle \text{rest of variable list} \rangle$ and records their addresses in *v* and *rovl*. It then transfers control to label 1. (The repeated use made of *v* and *rovl* is the usual device of using a single location for several purposes.) In this way we unravel the recursive structure of the original *vl*, exposing a new $\langle \text{variable} \rangle$ *v* at each stage, which is then handed on to a subroutine, *read variable*, to process. The tree for a $\langle \text{variable} \rangle$ in this case is utterly trivial consisting as it does of a single 'I' word only. In actual practice it would be much deeper. Finally we reach a stage with this conditional operation where *rovl* takes the second alternative form, namely $\langle \text{nil} \rangle$. In this case we do not jump to 1 but pass on to the *end* (which also serves as a *return* exit).

The foregoing routine would appear in Compiler-Compiler language as:

```
ROUTINE [ss]  $\equiv$  read ([VARIABLE LIST]) [EOL]
LET [VARIABLE LIST]  $\equiv$  [VARIABLE][REST OF VARIABLE LIST]
1) read ([VARIABLE])
 $\rightarrow 1$  IF [REST OF VARIABLE LIST]  $\equiv$  [,] [VARIABLE][REST OF VARIABLE LIST]
END
```

It will be seen that the name location *vl* is implied by the heading, while *v* and *rovl* are implied by the identifiers appearing in the subsequent statements. If more than one subphrase of the same type has to be identified, they would take the form, for example,

[VARIABLE/1], [VARIABLE/2], . . . , corresponding say to *variable* name *v1*, *v2*

The tree manipulations just described are essentially name assignments. We have said nothing so far about the actual generation of trees from other data, and their assignment to 'tree locations' in the store. A phrase definition, for example,

phrase *index* = $\langle 'i', 'j', 'k', 'l', 'm', 'n' \rangle$

is essentially a type declaration, it does not imply the existence of a 'location' for a tree named 'index'. Likewise only type names (or, as in the above case, basic characters) can appear on the right-hand side of a phrase definition. If we wish to actually store trees of a given type then we write a declaration of the form, for example,

index: *i1*, *i2*

The precise mechanism by which local storage is allocated for a tree does not concern us here provided the tree takes the general form described earlier. The difficulty is that it is impossible to predict at compile time what the size of the tree will be. One can determine the maximum extent

of each node and hence, *provided it is not recursive*, the maximum size of the tree assuming the nodes are packed contiguously. If it is recursive, however, it is not possible to say anything until the tree is actually generated, and in this respect it is worse than dealing with an **array** declaration where we can at least allocate storage on entering the block in question. (It is somewhat similar to the problem of storing multilength integers in which all digits are retained.) One way to deal with objects of this type is to use the sliding store principle (see Rosen 1964). In this system storage is only allocated when the objects are generated, and when it is necessary to replace an old 'value' by a new one, the former is deleted by 'closing the ranks' and adding the new 'value' at the end.

There are two methods by which trees can be generated. The first is by analysis of a character string (stored say as coded integers in an **integer array**) with respect to some phrase definition. This can be done by an instruction of the form

analyse $A(i)$ wrt p

where $A(i)$ indicates the character at which scanning is to start, and p the destination of the resulting tree. (p will be given in a value list together with the associated phrase definition.) The analysis algorithm is described by Rosen (1964).

The second method is by an instruction of the form, for example,

let $ss = 'r' . 'e' . 'a' . 'd' . '(' . v . ')'$

Here the right-hand side is analysed (at compile time) with respect to the phrase definition associated with ss and a provisional tree is constructed into which a subtree of the type (associated with) v can be plugged when the instruction is eventually executed (and the subtree computed). More precisely the provisional tree, which forms part of the instruction itself, is copied to ss , and at the same time the connection with v is made. The latter is in effect a name assignment: no physical copy of v is made. (In the Compiler-Compiler trees constructed in this way were kept in the stack and simply allowed to advance the working area.) However, one can run into trouble with this approach if tree generation is allowed to become repetitive, without some attempt being made to recover the space occupied by trees which are no longer required.

Another very useful tree function is

p category

which yields the value of the principal category word as an **integer**. This is more or less essential for dealing with phrases in which there are many possible formats, for example, the class of source statements in Fortran. The category number can be used (via a switch) to transfer control directly to the relevant processing sequence or routine. The only alternative would be a sequence of conditional **resolve** instructions which would be very slow indeed.

PROBLEM-ORIENTED LANGUAGES

To sum up, the new features proposed for tree processing are the introduction of phrase definitions, the instructions

```
resolve . . . into . . .  
if . . . resolves into . . .  
analyse . . . wrt . . .  
let . . . =
```

and the function category.

CONCLUSION

Discussion has been confined to data which can be 'named' by a single address word which points to the 'first' of the locations occupied by the item. The information found in these locations is interpreted according to rules built into the compiler, although certain decisions, involving the dynamic characteristics of the type, may be postponed until run-time. This type of data reference is sometimes called a simple name (a term introduced by Strachey and Wilkes in connection with the call-by-name facility of Algol procedures) to distinguish it from the more general (but far less efficient) device of using an algorithm to define either the precise locations involved (the left-hand-value) or the information itself (the right-hand-value).

In some compilers a machine address is synonymous with an integer. (If not, one can declare a sufficiently large vector, for example, **array store** (1 : 10,000), and use subscripts as addresses.) In this case data systems of the type discussed here can be simulated by imputing an association between certain integers, the 'names', and certain storage locations (either absolute or array elements), the 'data'. Some Compiler-Compiler facilities recently introduced into Atlas Autocode (Brooker, Rohl & Clark 1966) are partly of this character and partly built-in.

The trouble with such interpretive systems is that we throw away the possibility of checking the legitimacy of the operations we attempt to perform. This is one of the objections to the use of **array fn s** in Atlas Autocode (at least as implemented on the Atlas). The declaration **array fn A(s, p, q)** implies that a subsequent operand $A(i, j)$ refers to the number standing in location $s + p \times i + q \times j$. It is usually employed to remap an existing matrix, *B* say, *s* being associated in some way with *B*. There is, however, no provision for checking this, or even that the final location is sensible. We must try to ensure that in providing more facilities we do not at the same time give the user more opportunities to make a fool of himself.

ACKNOWLEDGEMENTS

Mr R. B. E. Napper read the first draft of this paper and made some very helpful comments.

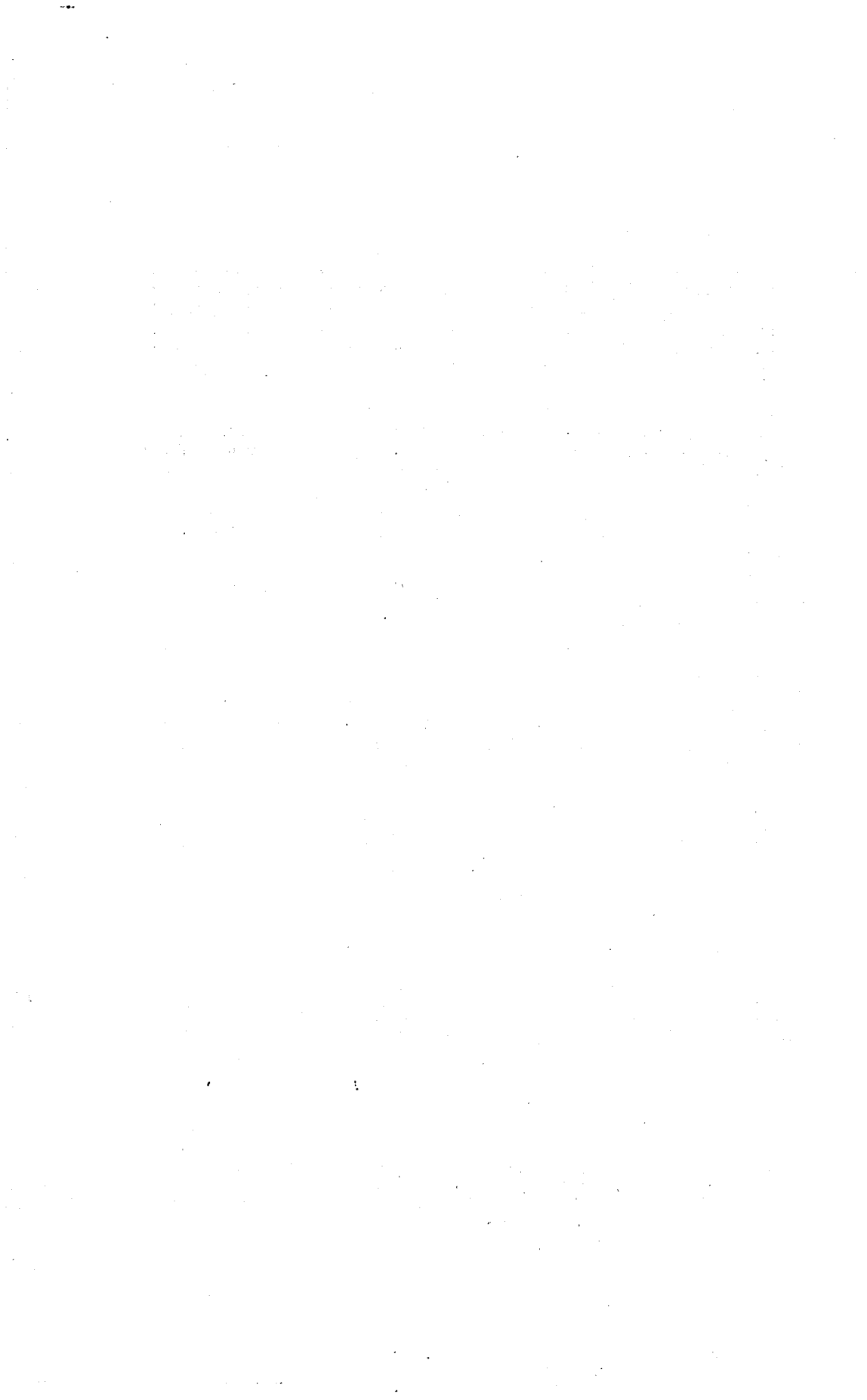
APPENDIX: PHRASE DEFINITIONS AND FILE STRUCTURE

There is a close analogy between phrase definitions and the file structure of data processing records. The latter could be defined in the same way as phrases (although at most levels there is usually only one possible format) but physical separability of the data, particularly at the lowest levels, is not consistent with efficiency. The latter dictates that information must be packed, at least down to character units, and possibly to bits, in order to minimise scanning of magnetic tape on which thousands of records may be stored. The use of machines with byte storage would help in this matter.

In any case phrase definitions corresponding to basic units of information, e.g., integer, real, are implemented as subroutines, which recognise the appropriate data formats in the source string, and convert the information to the appropriate internal form. For example, the phrase for real would simply call in a subroutine to 'read' a decimal number from the 'input' source string, convert it to standard floating binary form and terminate the tree with it in place of the usual 's' words which would otherwise lead to something further.

REFERENCES

- Brooker, R. A., MacCallum, I. R., Morris, D., & Rohl, J. S. (1963). The Compiler-Compiler. *A. Rev. autom. Progrmg*, 3, 229-275.
- Brooker, R. A., Rohl, J. S., & Clark, S. R. (1966). Main features of Atlas Autocode. *Comp. J.*, 8, 303-310.
- Iverson, K. E. (1962). *A Programming Language*. New York: John Wiley.
- Rosen, Saul (1964). A Compiler building system developed by Brooker and Morris. *Communs Ass. comput. Mach.*, 7, 403-414.



THE THIRD-ORDER COMPILER: A CONTEXT FOR FREE MAN- MACHINE COMMUNICATION

R. B. E. NAPPER

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF MANCHESTER

THE COMPILER-COMPILER

The Compiler-Compiler of Brooker and Morris is a special-purpose compiler for writing compilers. It was designed specifically to facilitate the writing and development of the Systems compilers for the Manchester Atlas computer, including the new language Atlas Autocode. A *compiler* is a computer program whose job is to accept any (*source*) program written in the designed language and translate it into machine code, so that a computer can then execute this program to carry out the particular operation required by the programmer. The compiler is itself a computer program, for which the data is the source program, and it has to be written in some language or other, usually a primitive machine-code assembly language.

In the case of the Manchester Atlas it was considered that as a number of compilers had to be written it would be more convenient to devise a special language for the compiler-writers instead. This special compiler had of course to be written first, using machine code. It was called the 'Compiler-Compiler', but it can be used in the normal way as a compiler, particularly for data-processing where the data is in the form of language. However it will be described only in its original context, as a compiler for compiling compilers (a *compiler-compiler*).

The compiler-compiler is fully described in Brooker, MacCallum, Morris & Rohl (1963), which is in effect a manual for a user of the system, and in Napper (1965), which is designed as a self-contained appreciation of the system for the lay programmer.

The language of the compiler-compiler system can be broadly divided into two classes as follows:

PROBLEM-ORIENTED LANGUAGES

(i) *Master-instructions.* A compiler description is made up of a collection of master-instructions. The most important of these are the PHRASE, FORMAT and FORMAT ROUTINE declarations, which provide a language for language description.

(ii) *Directives.* There is a simple language for use inside format routines, in which the compiler-writer describes the actions required to translate source instructions into the appropriate machine code according to the conventions of the autocode he is implementing. This language includes some special instructions for manipulating source language.

The compiler-compiler automatically incorporates in each compiler machinery for recognising and translating source language. This machinery works in conjunction with the language-description and language-manipulation facilities to provide a powerful mechanism for processing source language.

A PHRASE definition associates a suitably named *class word* with a phrase or set of alternative phrases, to define the set of symbol strings that are members of the class. The class word can then be used in a phrase or format definition to represent this set. A *phrase* is a string of symbols, or a class word, or in general a string of symbols and class words which defines a set of symbol strings direct, for example:

PHRASE [EQUALS]=equals, =, [IS] [EQUAL-TO], [IS]

PHRASE [DOES-NOT-EQUAL]=does not equal, $\neq$, [IS] not [EQUAL-TO], [IS] not

PHRASE [IS]=is, are, was, were, will be

PHRASE [EQUAL-TO]=equal to, same as

A FORMAT is a phrase which defines an instruction type, for example:

FORMAT: set [VARIABLE]=[EXPRESSION]

FORMAT: go to [LABEL] [if, unless] [EXPRESSION] [COMPARES-WITH] [EXPRESSION]

Some class words have recursive definitions, and so the set of possible symbol strings (i.e., instructions) satisfying a format may be infinite.

Each format has an associated FORMAT ROUTINE. When the compiler is operating on source program, the built-in language-recognition machinery deals with successive instructions by testing the head of the symbol string of the rest of the program against the list of formats of the source language. When it finds a matching format it passes control to the associated format routine, which describes what action is then to be taken to translate this next instruction. These routines carry out the backroom organisation required, e.g., building up or searching the list of parameter names of the source program, and in particular they form the appropriate machine code instructions to add to the compiled program.

When a source instruction is matched to a format, a record is kept of the substrings matching each class word in the format. These symbol strings are the natural parameters of the format routines, as they define the particular variant of the format to be dealt with. They are therefore treated as a

special kind of variable and they can be referred to inside format routines by special *phrase-identifiers*. Special *phrase-handling* directives are provided to operate on these phrase-identifiers, in particular to extract information about their associated symbol string.

In addition, phrase-identifiers can always be included in ordinary instructions in the format routines, provided they match a corresponding class word in the instruction format. In this case the current value of the phrase-identifier is substituted into the instruction when it is obeyed, i.e., when its format routine is entered as a subroutine.

As a trivial example, consider:

```

FORMAT: note that [VARIABLE] [EQUALS] [EXPRESSION]
ROUTINE: note that [VARIABLE] [EQUALS] [EXPRESSION]
          set [VARIABLE] = [EXPRESSION]
          END

```

Here the words in square brackets, which in the format are class words, in the routine heading and the routine are phrase-identifiers. If the instruction

note that a is $b + c$

was met in the source program, the routine would be entered with phrase-identifier values $[VARIABLE] = 'a'$, $[EQUALS] = 'is'$, and $[EXPRESSION] = 'b + c'$, and thus the instruction obeyed inside the format routine would be a call for the format routine of the more basic format 'set . . .', which will compile in effect:

set $a = b + c$

The language defining machinery is available to the compiler-writer to allow him to define special auxiliary formats and routines for use in format routines, to give an effective subroutine mechanism for his compiler program.

Formats are in fact divided into *format classes*, and each format definition must state which class it belongs to. There are four format classes built in to the system:

- (i) *MP (Master Phrase)*: the set of master-instructions of the compiler-compiler.
- (ii) *BS (Basic Statement)*: the directive language of the compiler-compiler.
- (iii) *AS (Auxiliary Statement)*: the set of subroutines of the compiler program.
- (iv) *SS (Source Statement)*: the source language.

BS and AS instructions can only occur inside routines. SS instructions can also occur inside format routines, but SS format routines are more commonly entered at top level from the built-in recognition machinery during compiling.

The compiler-compiler translates phrases and formats into stored information for subsequent use by the language-processing machinery, and it translates format routines into machine code routines of the compiler program. Inside format routines, BS instructions are usually translated into the appropriate machine code, but for AS and SS instructions unconventional cues are planted to the associated format routines.

THE SECOND-ORDER COMPILER

The process of using a compiler-compiler to write a systems compiler to translate source programs is generally done in two separate stages. That is, when the compiler has been developed, it is translated by the compiler-compiler and transferred to magnetic tape by a special master-instruction. It is then a conventional compiler in machine code, stored on magnetic tape, that can be called upon to translate any program written in the designed language.

In developing his compiler the compiler-writer does not want to go through this double process in order to introduce or correct one of the formats of the language and then test it; it is convenient to be able to carry on as part of the same run and compile a section of source program to test the compiler. Equally he does not particularly want to precede each such development run with all the satisfactory phrases, formats and format routines already developed. So it is convenient for him to combine the facilities on hand, so that the incomplete compiler is stored on magnetic tape during development together with a copy of the compiler-compiler. At the beginning of each development run it can then be called down as if it were an ordinary compiler, but the compiler-compiler can first be instructed to translate further phrases, formats and format routines, and to incorporate them into the compiler, before passing control to the compiler to translate a test piece of source program, finally passing control to the compiled source program to obey the test. If however it is required to make an already tested addition or correction to the compiler, instead of compiling a test program, a special master-instruction can be used to put the amended compiler back on tape.

The Brooker-Morris compiler-compiler is designed so generally, using bootstrapping techniques, that this procedure is quite natural to it. It has been stated that the compiler-compiler automatically incorporates language recognition and translation machinery into the compiler program. But of course the compiler-compiler is itself a compiler program, and most of this machinery is equally relevant to its task of compiler-compiling. Therefore the machinery is a carefully integrated whole that is used both when compiler-compiling a compiler and when the compiler is compiling a source program. It is even used when developing the compiler-compiler itself, to add on the less basic facilities (e.g., most of the directive language) using the more basic facilities. In fact a compiler is grown on to (a copy of) the compiler-compiler just as if it was a further development of it—the translated phrases, formats and format routines are not stored separately. When the compiler is fully developed, a special master-instruction removes all the machinery not required for just compiling source program, and the compiler is finally stored as a production compiler.

An interesting question is now raised: what kind of object is the partly developed compiler? It is a compiler, because (if it is sufficiently complete) it can be called upon in the standard way to translate source program. It is

equally a compiler-compiler, because the original compiler-compiler is a subset of it, and one could start defining a completely new language for it to translate, provided it did not clash with the language already defined. In fact, apart from looking to see if there is a dictionary of ss formats on hand, it is impossible to tell from the internal structure of the compiler whether or not it is just the original compiler-compiler.

In this case the compiler-cum-compiler-compiler is perhaps best called a *Second-Order Compiler*, that is, a compiler that can compile any program written in any language, provided the source program is preceded by a definition of the language in the (fixed) compiler-compiler language. It can do all this process in one run, starting with the basic compiler-compiler. Or at any point a subset of source language definition can be stored with it on tape, subsequent calls on it only requiring language outside this set to be defined before compiling the source program; and additions to the subset on tape can be made at any time.

Now the language of a conventional compiler is fixed: that is, the compiler can only translate a program written using the fixed set of ss formats defining the language. Extensions to the language in the program can be done indirectly using the conventional routine mechanism that is usually built-in to the source language. This means that provided the correct format for specifying routines and calling them is adhered to, library routines, or routines specially written by the programmer for his program, can be included in his program and called by instructions that read like basic ss instructions in the language.

However the second-order compiler allows a programmer to extend the language any program is written in indefinitely, by preceding the source description with a definition (in the compiler-compiler's language) of any previously undefined language. In principle each programmer can have his own breeding of the second-order compiler on magnetic tape, comprising the basic compiler-compiler, and a subset of useful basic source instructions (together with the AS routines involved in their definition) which are common to all programmers in his field of work; to this he adds his own personal selection of library and system format routines, maybe with adaptations, together with some private definitions, to give a personal language in which he can most easily express himself when programming. Then for any particular program, any further language particular to the program can be defined before it is used in the source program, as part of the program description. Thus any program can be written in a language uniquely adapted to the program itself, its writer(s) and his field of work.

Current work by the author, described in Napper (1966), is implementing such a system, providing a basic source language subset, and devising suitable behind-the-scenes organisation and establishing certain conventions to enable the ordinary programmer to make simple additions to his language safely and without detailed knowledge of compiler-compiler theory. The language of the basic subset is in 'English Mode' with instructions reading as

English and combined in a clause and sentence structure. There is a 'Mathematical Mode' similar to a simple conventional autocode language, but the more interesting use of defining language is expected to be in English Mode. So the organisation provided is such that definitions of the language are as simple as possible, and all English Mode formats will fit automatically into the clause and sentence structure. With this freedom of definition it is possible to write programs which at higher levels read as a computer-independent description of the job being programmed, in free and 'natural' English. Further, the phrases, formats and format routines of the language description, with suitable choice of English class word names, can be made to read as a computer-independent description of format and meaning.

Such natural language is not particularly suited to purely mathematical programming, but in those sections of program where English is a suitable medium of expression, this second-order compiler allows very complicated formats to be built up, which retain complete clarity (in the context of the job on hand) because they correspond to the original description of a more complicated task (including clauses, options of repeated phrases or clauses, or free strings that are implicit routine calls). These can be used safely instead of breaking them down into a set of more basic instructions connected by a control complex.

For example, consider the format:

```
FORMAT [ORD] = reorder[s]list[s][OF][s][PROPERTY][&PROPERTY* ?][s][OF][s]
                EACH[s]
                [item][WHICH ?][LIMITS ?][s]in[s][a,de]scending[s]order[, ?][s]
                according[s]to[s][CRITERION][IF-EQUAL..* ?][PAUSE]
```

where [IF-EQUAL..] = ;[s]if[s]equal[,][s]order[s]in[s][a,de]scending[s]order[, ?]
[s]according[s]to[s][CRITERION]

[Note: [s] denotes a general space (including comment or noise words), '?' denotes an optional phrase, '*?' an optional phrase maybe repeated; [ORD] is the format class (a subclass of ss) of ordinary English Mode instructions as opposed to conditional clauses.]

This has been used in a program as follows:

Reorder the list of the ALLOCATION, IDEAL-COLUMN-OF-THE-ALLOCATION, and REFERENCE for EACH hole which is affected by the current priority class, from the bottom-hole-in-the-column up to the top-hole-in-the-column, in descending order according to the ((priority-of-the))ALLOCATION; if equal, order in ascending order according to the difference between the IDEAL-COLUMN-OF-THE-ALLOCATION to the hole and the current column.

In this particular example 'which is affected by the current priority class' is not one of the formally defined alternative forms of a 'which' clause, being the last alternative, the phrase [ANYTHING] which accepts everything up to the next piece of punctuation. Whenever it is required to discriminate

the subset of [item]s to be operated on, this alternative is transformed into the call for a conditional clause 'if the [item] [ANYTHING]', e.g., 'if the hole is affected by the current priority class: . . .'. Note however that this transformation is not carried out in the reorder format; it is carried out automatically and generally, at a lower level, as part of the standard loop control machinery, and it is transmitted through the level of the format routine automatically when loop control formats are used, for example 'REPEAT for EACH [item][WHICH?] from the current [item] back to the [STARTING-POINT]'.

It is hoped that such a transformation of the power of expression in programming will enable experienced programmers to write large programs with great accuracy and clarity of description—so that they can be easily understood and even amended with safety by someone other than the writer. However, leaving aside such as yet nebulous and unproven advantages, the second-order compiler provides three significant practical advantages compared to existing autocodes with their conventional routine mechanism:

(i) New units of language can have completely free formats (within reason) whereas in an autocode every routine call must obey a set format. In both cases the format of *routine description* must be as laid down by the compiler.

(ii) The source program compiled using the second-order compiler is more efficient. Every instruction in the second-order compiler's source program will be compiled straight into tailor-made machine code without resorting to a cue-subroutine mechanism unless this is more appropriate; and in a complicated routine with many variations of format (e.g., 'reorder'), only the subset of instructions required to deal with the variant on hand need be compiled, and hence obeyed, whereas if a subroutine is dealing with such a general job, the particular relevant subset must be isolated at execution time.

(iii) No declaration of parameter type is required, as the parameters involved are those of the block in which the instruction appears—the parameters referred to here are the conventional parameters of the source program, and they will usually be represented in format routines by phrase-identifier parameters of the compiler program. Directives can be used to establish parameter types if required, and in particular local parameters can have their types set by association, e.g., 'LOCAL NAME: list-value-being-ordered; type as for [PROPERTY] [OF] [item]'.

As a simple example, consider the following routine as defined in a conventional autocode:

```
routine form (<type> name sum, <type>: x, <type>: y)
    sum = x + y
end
```

where the <type> has to be specified in each of the three cases (e.g., all 'real' or all 'integer')—hence the routine could only be used for one combination of types throughout the program.

PROBLEM-ORIENTED LANGUAGES

This routine is called by, e.g., '*form (sum1, a, b)*'.

With the second-order compiler in English Mode, we have:

```
FORMAT [ORD] = form[s]sum[QUALIFIED ?][s]of[s][NUMBER][s]and[s][NUMBER]  
[PAUSE]
```

where [QUALIFIED ?] allows any further symbols up to the next space.

ROUTINE [ORD] $\equiv$ form the sum[QUALIFIED ?] of [NUMBER/1] and [NUMBER/2].

Set the sum[QUALIFIED ?] = [NUMBER/1] + [NUMBER/2]. END

This format can then be called by, e.g., '*form the sum:1 of a and b*' and the effect of the machinery will be the same as if '*set sum:1 = a + b*' had been written instead.

In both cases x & y or the [NUMBER]s could be any arithmetic expression, but this form of language would not be 'natural' with complicated expressions.

Assuming that all the numbers involved are simple items, then the second-order version would compile direct into the two basic operations, an assignment and an addition. In the autocode version, in the cue the *values* of a and b and the *address* of *sum1* would be copied, and the instructions obeyed for jumping to the routine and going down another level of the 'stack'; then the two basic operations would be obeyed; then the stack pointer would be reset and control passed back to the block containing the cue. Thus three or four times as many instructions would be obeyed in execution for each call of the routine.

For longer routines the balance of advantage in efficiency moves towards the routine form, as the number of instructions obeyed in the routine begins to swamp those involved in the entry-exit mechanism, and compiling a large block of instructions for each call on the format (instead of just a cue) outweighs the advantage of quicker execution. Then of course a formal routine mechanism in the basic source language of the second-order system can be used, or the format routine could organise its own subroutine to do the bulk of the procedure, compiling a cue and the subroutine on the first use of the format and subsequently compiling only a cue.

It should be emphasised that this ('macro') facility for compiling nonbasic instructions direct into machine code causes limited increase in efficiency when used where a routine mechanism is commonly used in a conventional autocode program, to save repeatedly writing out a sizeable block of similar instructions in the program. By implication the block of instructions is likely to take long enough in execution to render the relative time spent in entry and exit insignificant. The facility is however critical when it is used in 'trivial' definitions where the chief object is to write an instruction representing perhaps only one or two basic instructions simply to achieve a greater measure of self-description in the instruction in the context of the job being programmed; the basic language of an autocode must of course be very general and therefore neutral in meaning, and the main opportunity for self-description is usually only in the programmer's choice of parameter names.

THE THIRD-ORDER COMPILER

While the compiler-compiler is a great improvement on a machine code assembly language, it only places the compiler-writer in a similar position to the programmer of a conventional autocode. It would be nice to confer on the compiler-writer the same relative advantage that the second-order compiler confers on the ordinary programmer. And whereas it is debatable how far programs containing much mathematical calculation are suited to expression in a nonmathematical language (e.g., 'English', as encouraged by the author's implementation of a second-order system), there is little doubt that compiler-writing is.

The compiler-compiler can be extended and adapted, just as without it compilers can be; but this entails altering a machine code program (or one written in a primitive language), which is difficult and dangerous even if done by a member of the original compiler-writing team. So in practice the compiler-writer using a compiler-compiler is restricted to the language-processing machinery available. Further, he has to use the fixed (AS) routine mechanism provided, and although it technically enables him to extend his language, in practice (as in current autocodes) extensive use of it is prohibitively inefficient. So he has to program in the simple autocode-type language provided for him. Relative to the current autocode user he has the advantage that routine calls do not have to obey a set format, but the disadvantage that the top-level routine of his program and all the compiler-compiler machinery used in compiling (i.e., that is part of his program) is written in a primitive language and hence is in practice unalterable and its precise effect obscure.

In order to effect the similar revolution in the compiler-writer's lot that the second-order compiler does for the ordinary programmer, we require a *third-order compiler*. This would provide the compiler-writer with the analogous extra facilities required to make programming easier. In addition it would formally recognise the three modes of operation of the system and provide three suitable basic languages for each, with appropriate facilities for each.

The three modes, with a summary of the existing situation and the more obvious extensions required, are as follows:

1. *Compiler-compiler mode*. This reads the definitions of language written in the compiler-compiler language and converts them into machine code sections of the compiler to translate source program. There are only a handful of master-instructions commonly used, e.g., FORMAT, PHRASE and ROUTINE. All the machinery involved in language definition and recognition is written in machine code or primitive autocode (by the compiler-compiler writer). Every nonbasic routine call recognised in a format routine is planted in cue form.

We require from the third-order compiler a wider range of master-formats, with associated routines written in an understandable simple (as opposed to

primitive) language, so that the compiler-writer can adapt the machinery he is provided with, and easily define more of his own; in particular it should be easy for him to incorporate existing machinery as subroutines of his new machinery.

For example, it might be convenient to have different forms of ROUTINE format, e.g., for AS routines only used by the compiler-writer, for AS routines which can be used by the source programmer, and for SS routines, one each for ordinary instructions, conditionals, and directives (i.e., instructions that take effect at compile time not run time). In PHRASE definitions, different forms may be required, e.g., to automatically construct full definitions from incomplete specifications; there is an obvious need illustrated in Napper (1966) for a master-format, e.g., KEY-WORD [hole], to construct a phrase definition for the infinite set of 'composite words' containing the word 'hole', e.g., hole, hole:A, ideal-hole, hole-above, top-hole-in-the-column, BUT NOT whole. It might also be necessary to adapt the language-processing machinery to deal with such special forms.

To revolutionise the compiler-writer's freedom of expression, it is required to be able to *enter* format routines for AS instructions recognised in compiler-compiler mode, instead of just planting a cue so that they are obeyed interpretively in compiler mode; so that every AS instruction that does not involve a phrase-identifier can be compiled direct into compiler-mode machine code (if required). This is analogous to the tailor-made advantage of the second-order compiler over conventional autocodes in compiling non-basic instructions. To meet this requirement involves a considerable addition to the theory and structure of the second-order compiler. In particular it has the same problem as in the source program analogy, of wanting two different forms for each definition of language—a cue-subroutine form and a 'macro' form depending on whether it is possible or economically desirable to compile a separate set of instructions. It is expected that this problem will be formally solved in the second-order system for source program, so that the same definition can cover all cases in the most effective manner without any special help from the programmer; it is therefore hoped that the analogous problem will prove soluble for the third-order system in compiler mode.

2. *Compiler mode.* This reads the source program, recognising successive instructions as members of the SS format class, setting up the appropriate phrase-identifier values for each class word occurring in the format, and entering the appropriate format routine to compile the requisite source program. Compiler mode uses some of the built-in compiler-compiler machinery, but most of the programming involved is in the organisation written by the compiler-writer who has provided the basic subset of source language and established the corresponding conventions with which he can conveniently build up less basic permanent and library routines, and in particular which enable the source programmer to write his own definitions of language easily and safely. The compiler-writer's language is the basic directive subset provided by the compiler-compiler, and his routine mechan-

ism is the AS routine, which although it provides freedom of format is uneconomic if used extensively for small routines.

We require from the third-order compiler a basic English Mode programming language similar to that the author's second-order system provides for the source programmer, to enable the compiler-writer to describe his basic programming in compiler mode clearly and accurately in English. On top of this he can define his own language freely and efficiently using the more powerful AS machinery described for compiler-compiler mode. It is not convenient to use the same language as the source subset, as this can cause confusion where a format routine contains both AS and SS instructions (as the former usually take effect at compile time and refer to the compiler organisation of data, and the latter usually take effect at run time and refer to the source program organisation). Also the conventions of compiling machine code may be different; for example, the compiler's routines may have to be relocatable but the source's may not have to be.

3. *Source mode.* Having compiled the source program in compiler mode, the compiler passes control to the source program, which executes the job described by the source program, independent in general of the compiler.

The source programmer already has the benefit of the second-order facilities which the third-order compiler was designed to provide for the compiler-writer. The second-order system gives him a very powerful means of expression, and it compiles more efficient programs (but takes longer over compiling). What further benefit can he hope for from a third-order system?

A CONTEXT FOR FREE MAN-MACHINE COMMUNICATION

At present about the only context for a compiler-compiler has been the introduction of a new large computer, where a number of compilers are required quickly, mostly adaptations to the machine of existing languages, but also one or two new ones. For new languages it might also be desirable to revise the compiler substantially after a year or two's practical experience. Such calls for a compiler-compiler are rare enough; what immediate prospect is there of wanting sufficient repetition of the compiler-writing process to justify a further refinement of the facilities?

The answer to this question derives from the answer to the previous question. Any further advance in programming beyond a second-order system must surely be in the communication from the machine back to the programmer, to help him write, test and run his programs. The second-order system provides him with a powerful means of communication *to* the machine—any further increase in power will only aggravate the existing imbalance. For at present any communication *from* the compiler tends to be very primitive both in language and in content. This is partly due to economic considerations, but chiefly it is because there is no subadaptation of the communication from the compiler; the same programmed responses apply equally to all programmers, and with few exceptions to all programs.

PROBLEM-ORIENTED LANGUAGES

The two most obvious ways of narrowing the context within which communication is made, and hence increasing its relevance and variety, is to make communication easily specifiable from individual non-basic routines (e.g., library routines), and to make it adaptable to the individual programmer. There is clearly great scope for communication in large and complex routines, e.g., in 'reorder . . .' or in mathematical routines where there are many ways of carrying out the operation depending on the characteristics of the data.

The second-order system goes part of the way towards providing this narrowing of context by encouraging a great build-up in routines, and allowing adaptations of existing language to the individual. Many of these routines may be trivial, apparently saying the same thing in a different way. But the fact that we want to say something different usually implies that the context is different.

However it is not ideal for the programmer to have to specify all the communication back to himself to achieve the required adaptation. Neither could the compiler-writer deal personally with all the users of his compiler. The answer may be to split ordinary programmers into groups each looked after by a *master-programmer* who provides them with a comprehensive programming service, providing teaching and advice as well as help in writing, testing and running their programs. In a large University, say, the structure suggests itself of one or more master-programmers in each department supervising a group of pupil programmers. Each department would have its own adaptation of the most suitable language(s) based on the common basic language together with a set of library routines tailored to the requirements of the department—a 'dialect'. Each ordinary programmer would have a set of private formats (now fairly small and mostly written in collaboration with the supervisor), and these together with his supervisor's dialect would be called in to translate all his programs.

The master-programmer system provides the justification for a third-order system. For it would bring about a proliferation of compilers (but not necessarily of languages). It also places a heavy burden on such a system. For the emphasis is not so much on a compiler-writer providing a working language, as on a master-programmer, given a basic compiler system, providing a comprehensive computing service for a set of ordinary programmers. To do this he will require a very powerful and accurate language, particularly as his system will be subject to continual amendment.

There is not space to outline the potentialities of such a system in newer fields that might become possible, such as teaching programmers, error detection and correction, and monitoring in general (in particular running programs under compiler control—not necessarily interpretively). All these newer possibilities provide a context for language generation, and if all the higher-order systems programming is also in expressive English, all communications should be immediately understandable in their context.

The following brief examples will illustrate two or three more obvious opportunities for language generation. These are applicable both to straight-

forward compiling and in an off-line system, although of course they are equally applicable in the contexts mentioned above, and are more suited to an on-line multiple-access computer system. *Directive English* refers to the projected free English programming in compiler mode of the third-order system, where the parameters are frequently phrase-identifiers as well as more conventional variables.

The machinery of the compiler-compiler is already adaptable to providing easy generation of language. A format is a generalised sentence, reading (with imaginative choice of class-word names) as a general English sentence, which represents an indefinite set of source sentences; its precise meaning can be derived from breaking down the general structure, using phrase-handling instructions that read in directive English as a computer-independent analysis of meaning. In the same way, new sentences can be built up in appropriate contexts in format routines, that read as a relevant generalised communication in class-word form, but which on phrase-identifier substitution are output as source string immediately understandable in the particular context of the original variant of the format.

Apart from giving information and printing out errors in a more sophisticated way, there are two immediate contexts for language generation opened up by second-order programming. The easiest is for a format routine to print out the source string of the variant it is dealing with. This is of little interest if it is an ss routine being obeyed at top level, as the sentence already occurs in the source program. But if a switch is set so that this printing out only occurs in the level one below top level, or only at the level of basic English, the compiler will then give a continuous translation of the source program into English at one level down or into basic English. For example, when 'form the sum of $a+b$ ' was being compiled, 'set the sum = $a+b$ ' would be printed. The 're-order' instruction quoted previously would translate into about 35 English instructions in the level of the format routine (i.e., one level down) or about 80 at the level of basic English.

Similarly, if each format routine gives a REPORT directive, the compiler can arrange for a program to print out a continuous report of the instructions it is obeying in a section of program (at top level, or a lower level if required). The report format would be basically the same format as the instruction together with a specification of which values were to be printed as well. Thus the report on 'form sum' if a was 2 and b was 3 would be 'form the sum (5) of a and b '. Conditional clauses would require two report formats, one if the condition was satisfied ('since . . .') and one if not ('note that . . . not . . .'). In reporting directive English for master-programmers, the values of the phrase-identifiers would of course be source strings. Such reporting is quite impressive, since it reads as the master-programmer's thought-process in translating higher-level language into the requisite more basic language.

However the most interesting extension of the existing machinery is in 'conversing'. Obviously the system can answer any questions put to it, as

PROBLEM-ORIENTED LANGUAGES

the question would be recognised as a format in the normal way, and its format routine would establish the answer and generate a suitable output format to reply. However it is equally possible (particularly on-line) to formulate a question in a format routine where further information is required of the programmer. This will be printed out with source string substitution where phrase-identifiers are involved, and if the instruction specifies the format of the reply, it can then recognise the reply in phrase-identifier form, thus making new information available to the routine.

For example, a simple integral loop control format might be in its full form 'REPEAT for EACH [item] from a [STARTING-POINT] [in, ex]clusive [up, down] to a [FINISHING-POINT] [in, ex]clusive', but the actual format 'REPEAT for EACH [item][LIMITS?]' might allow all or any of the information about the [LIMITS?] to be missing. If no direction [up, down] is given, 'up' is assumed; 'inclusive' is assumed where no indication is given; if the starting-point or finishing-point is not given, the appropriate preset parameter FIRST-[ITEM'] or LAST-[ITEM'] is generated, where [ITEM'] is [item] with small letters raised to capitals.

If a programmer is using the format for the first time, or even a particular alternative of [LIMITS], it might be wise to check back that the correct assumptions have been made. So at the point when all the missing parameters have been generated, if necessary the routine could proceed:

ASK THE QUESTION: [REFERENCE]

I assume that the limits for the [item] are from the [STARTING-POINT] [in, ex/1]clusive [up, down] to the [FINISHING-POINT] [in, ex/2]clusive.

Is this correct?

WHERE THE [REPLY/1] SHOULD BE [YES], [NO], OR [MAYBE].

IF THE [REPLY/1] IS [NO]:

ASK THE QUESTION: what are they[please?]?

WHERE THE [REPLY/2] SHOULD BE [for EACH [item] ?][LIMITS]. THEN START AGAIN REPLACING THE [LIMITS?] WITH THE [LIMITS] OF [REPLY/2].

Note that the ASK format with its associated WHERE clause would be part of the standard compiler-compiler machinery that the master-programmer can use freely (maybe after adapting it slightly for his private use). The ASK directive prints out the symbols following, substituting the symbol string on hand for any phrase-identifiers involved. Then the WHERE clause, which locally defines the phrase (format) of the appropriate reply, reads in the reply, setting up a new phrase-identifier value that can then be tested or otherwise manipulated in the usual way by the phrase-handling directives. If the WHERE machinery fails to recognise a reply it might automatically request that the reply be repeated, giving the principal alternative formats of the required reply.

In this way, with standard machinery provided by the third-order system and maybe adapted for his own use, the master-programmer could freely specify any exchanges he requires to make in any situation. Routines of his

dialect would probably have separate memory for each programmer (rather than existing separately for each) and would be able to adapt to the individual, e.g., only asking the above question if his last use of the current alternative form had been incorrect. There would be scope for little learning programs to deal automatically with such situations and others. If the master wanted to carry out a certain action for a subset of programmers only, the requisite sequence would merely be prefaced by, e.g., 'IF THE PROGRAMMER IS Smith OR Jones:'.

Thus it could be possible for the master-programmer to communicate freely with his pupils, both direct and via his compiler, gradually building more and more of his actions into the system as he sees automatic patterns developing. And if programming in a more natural self-explanatory language is encouraged in source programming, this will enable him more quickly to see the context of source program when he is obliged to deal with it direct rather than automatically through generalised directive English. Similarly there would be analogous free communication in directive English between the master-programmer and the central master-master-programmer(s) (compiler-compiler-writer) to help him keep his master-program correct. This would give a double pyramid of communication, with additional communication existing laterally between master-programs (e.g., exchanging information and looking for someone else's format to match an unrecognised instruction), and between master-programs and the computer's Supervisor system, etc. All these communications would be in one or other of the modes of English, since all the programming concerned would be.

In this way it might be possible to have a computer installation that is a continuously evolving complex of people and mechanical systems all intercommunicating freely in English in a very large but economically viable context. In particular it should be possible to get some subsystems operating without a formal master, where a master-programmer has succeeded in mechanising a sufficiently high proportion of his actions, so that in the few remaining situations where the correct procedure is not formalised, little 'artificial intelligence' programs can be used to make decisions, or the system could ask for help from another master-programmer.

REFERENCES

- Brooker, R. A., MacCallum, I. R., Morris, D., & Rohl, J. S. (1963). The Compiler Compiler. *A. Rev. autom. Progrmg*, 3, 229-275
- Napper, R. B. E. (1965). An introduction to the Compiler Compiler. *Tech. Report, Dept. of Comp. Science, Univ. of Manchester*.
- Napper, R. B. E. (1966). A system for defining language and writing programs in 'Natural English'. *Formal language description languages for computer programming*, 231-248 (North-Holland).

PRINCIPLES FOR IMPLEMENTING USEFUL SUBSETS OF ADVANCED PROGRAMMING LANGUAGES

G. F. COULOURIS*

INSTITUTE OF COMPUTER SCIENCE
UNIVERSITY OF LONDON

The author has been concerned with the implementation of a subset of the CPL programming language (Barron *et al.* 1963) for use in a university (Coulouris & Goodey 1965). This paper describes the results of efforts to isolate those features of the language which may be implemented by relatively simple techniques. The subset eventually arrived at contains few limitations of importance in general programming. The techniques adopted generally result in a more efficient object code than would be possible with more sophisticated methods.

The implementation makes use of the compiler-compiler system (Brooker, MacCallum, Morris & Rohl 1963) and this paper contains some comments on the relative importance of the various features of that system in the implementation of syntactically complex block-structure languages such as CPL.

Although the discussion in this paper is largely in terms of the specific problems of the CPL language, the techniques adopted can be extended in a fairly obvious manner to other languages containing features which are semantically equivalent. The author apologises for any inadvertent use of local jargon; despite recent developments in the formalisation of programming languages, the discussion of implementation problems seems to be largely restricted to specific systems.

THE CHOICE OF A SUBSET

Practical limitations on time and programming resources suggested a corresponding restriction of aims to the implementation of a useful and

* Present address: Centre for Computing Automation, Imperial College of Science and Technology, University of London.

consistent subset of CPL. The following factors were considered relevant in the design of the subset (listed in order of decreasing importance); speed of the object code, efficient use of store in the object code, efficiency of the compiler and the ease with which later extensions could be made to the subset.

It was accepted as an *ad hoc* basic principle that the use of 'interpretive techniques' in the object code increased both the complexity of the translation problem and the time of execution of the object code.

For the purposes of this paper the term 'interpretive technique' includes those techniques which produce an object code containing references to data other than the explicit operands in the source code, or in which the flow of control does not necessarily correspond to that in the source program. One example of the latter would be an entry to a store allocation routine. Conversely, a non-interpretive code is one in which no reference is made to information other than explicit operands, and in which the logical flow of the object code is isomorphic with the flow implicit in the source program.

An added incentive for the acceptance of a 'principle of non-interpretation' was that the facilities of the compiler-compiler are not easily adapted to the implementation of routines for interpretation.

As we shall see in a later section, strict adherence to the principle would lead to hopelessly inefficient use of storage by the object program. Nevertheless, a compromise was reached by which it was possible to implement a very powerful subset of the language using object code less complex than that of many Algol implementations.

It is interesting to note that nearly all the omissions from the subset are semantic features of CPL. The many convenient syntactic features were easily handled by the very powerful syntactic classification facilities of the compiler-compiler.

In the following sections, the consequences of including various semantic features in programming languages are discussed.

THE STORE ALLOCATION PROBLEM

In terms of our 'principle of non-interpretation', there are at least three levels of complexity in schemes that may be used for the allocation of absolutely addressed storage to the operands in a source code:

- (i) the allocation of an absolutely addressed location to each element at translation time;
- (ii) the allocation of storage relative to one or more pointers varied at execution time (often known as a 'dynamic stack' system (cf. Randell & Russell 1964));
- (iii) the allocation of storage on an 'available space' basis at execution time. A store map is used and all references to the store are via the map. This technique is known as the 'Codeword' system (Iliffe & Jodeit 1963).

We will now investigate the potentialities of each of these schemes for the implementation of various features of CPL.

First, it is necessary to distinguish between *simple items*, namely those whose size and structure does not vary at execution time, and *compound items* which may change in size and structure. In CPL, the first category includes all items of types **index**, **integer**, **real**, **Boolean**, **logical**, **long logical**, **label**, **complex**, etc., while the second contains all **array** and **string** items. In Algol, a similar distinction can be made between items of types **integer**, **real** and **Boolean** on the one hand, and **array** on the other. The status of items of types **function** and **routine** in CPL is dependent on other factors and is discussed in a later section.

Consider the consequence of using store allocation scheme (i) for simple items. The scheme implies that each item uses a fixed store element, and that object code may be compiled containing direct references (without 'B-modification') to the absolute address corresponding to an item. Any resultant gain in speed of object code will derive from the fact that in many hardware systems the speed of absolute store referencing is appreciably greater than that for relative referencing. However, the allocation of a fixed store element to each item may produce inefficiency in the use of storage in some situations in comparison with the dynamic stack technique.

Unless sophisticated store optimising procedures are to be used we may take the following rule to characterise scheme (i):

Every store element allocated to an item remains associated with
that item throughout all portions of program during which the
item may be activated (1)

We shall call the portions of program mentioned the 'potential scope' of the item.

To understand the application of the rule, consider the following segments of a CPL program.

§ 1 let a, b be real let $F[x]$ be value of § 1.1 Input [a, b] . . . § 1.2 let $z = 3a \uparrow 2 + 2b \uparrow 2$ Output [$F[z] + 2F[z \uparrow 3]$] . . § 1.3 let $Z = 3a \uparrow 3 - 2b \uparrow 2$ Output [$F[Z] + F[Z \uparrow 2]$] . . . finish	let y be real Input [y] result is $x \uparrow 2 + y \uparrow 2$ § 1.1 § 1.2 § 1.3 § 1 259
---	---

PROBLEM-ORIENTED LANGUAGES

Consider first the variables z , Z declared in § 1.2 and § 1.3 respectively. The actual scope of z is § 1.2. It may only be activated by normal entry to the block through the heading. Accordingly, the potential scope of z is the same as its actual scope. Similar reasoning applies to Z . Thus z and Z may share the same store element under rule (1).

Now consider the variable y . The actual scope of y is § 1.1 (i.e., the block expression defining $F[x]$). Each application of F causes a reference to y . Thus the potential scope of y must be the entire actual scope of F . By (1) the store allocated to y is unavailable for other purposes throughout the remainder of § 1.

The rule corresponding to (1) which characterises the dynamic stack scheme (ii) can be stated:

Every store element allocated to an item remains associated with that item throughout the actual scope of the item (2)

A near optimal use of storage is achieved if scheme (ii) is used for simple items. (Still greater optimisation might be achieved by flow analysis methods.) For example, the variable y in the last example would now require a store element only during each application of the function F . The additional cost incurred by scheme (ii) derives from the need to make all store references relative to a 'stack pointer' and to manipulate the stack pointer during execution.

We will now enumerate the ways in which simple items may be activated during the execution of a CPL program and consider the consequences of using a scheme such as (i) in each case. Items can be activated in the following situations:

- (a) on entry to a block which is outside any routine or function body and which contains declarations. Each simple item is allocated a location for the scope of the block. The location is available for the other items outside that block;
- (b) on a transfer from outside a block to a labelled command within the block. The case is equivalent to the last one, since a transfer into a block may be transformed into a normal entry to a similar block with a multi-way transfer inserted immediately after the declarations;
- (c) on entering the scope of a **where** clause. Once again this may be transformed into case (a), this time by the introduction of a block. Thus

$$Y := ax + bx \uparrow 2 \text{ where } x = F[y]$$

is equivalent to

$$\begin{aligned} &\S \text{ let } x = F[y] \\ &Y := ax + bx \uparrow 2 \S \end{aligned}$$

- (d) the object program may need items as working space. These can be considered as concealed **where** clauses and obey the same rules as explicit **where** clauses;

COULOURIS

- (e) on a call for a routine or function whose body contains activations of simple items. This final case is the only one where the potential scope does not correspond to the actual scope. By rule (1) the locations allocated to all items used in a routine or function must be protected throughout the scope of the routine or function. This is analogous to the Algol treatment of 'own variables'. Additional action is necessary if a routine or function is called recursively. This is discussed in a later section.

We conclude that inefficiency in the use of store resulting from allocation by scheme (i) occurs only in the case of local items of routines and functions. This is not usually important, since the number of non-compound items used in a program is roughly equivalent to the number of named variables, and is generally small in relation to the number of elements of compound items. For these reasons scheme (i) has been used for the allocation of store to all non-compound items.

DYNAMIC ARRAYS AND OTHER COMPOUND STRUCTURES

It is not feasible to deal with dynamic arrays of the sort contained in CPL by absolute store allocation methods like scheme (i), because the size of the array may change between one activation and the next. To make matters worse, the size of an array variable may vary during its activation. It turns out that an optimal implementation of this feature is not possible even by the 'dynamic stack' technique which is adequate for the arrays of Algol.

Let us investigate to what extent a single dynamic stack is adequate, and where it fails in the implementation of the 'fully dynamic' arrays of CPL.

Arrays in CPL may be assigned in the same way as items of fixed size. Thus a program segment such as:

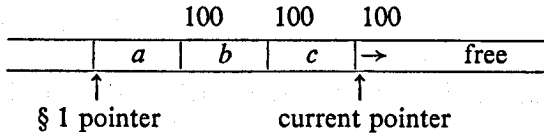
```
§ 1 let  $a, b = \text{Newarray} [\text{real}, (1, 100)]$ 
    and  $c$  be real vector
     $c := a$ 
    .
    .
    .
     $c := b$  § 1
```

is quite permissible, the assignments causing all the current values of the elements of a and b respectively to be copied and given the name c dynamically.

The problem is that such an assignment involves the provision of space for the copy, and the size of the space is unknown during compilation, or even at the time of entry to the block containing the assignment. If a single dynamic stack were used, we would make the copy appear at the next available space on the stack when the assignment occurs. On exit from a block all the space used during the execution of the block is made available again in the usual way. In the example given, the stack representations in Fig. 1

show that the result would be the one required, since the scope of c is precisely the smallest block containing the array assignments.

(i) after ' $c:=a$ '



(ii) after ' $c:=b$ '

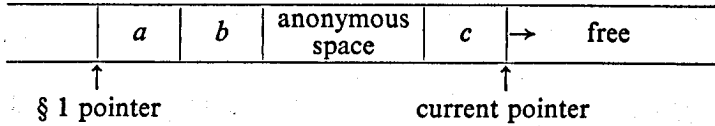


FIG. 1

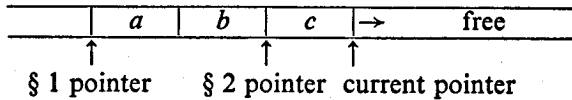
But consider the following:

```

§ 1 let a, b = Newarray [real, (1, 100)]
    and c, d be real vector
§ 2 .
    .
    .
    c:=a § 2
    d:=b § 1
    
```

We can see immediately from Fig. 2 that the result would be incorrect.

(i) after ' $c:=a$ '



(ii) after ' $d:=b$ '

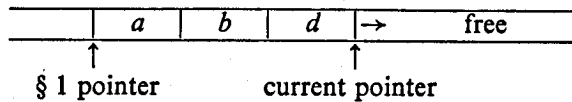


FIG. 2

The space used in the assignment to c becomes available on exit from the inner block and thus is overwritten by the assignment to d . This difficulty might be alleviated in several ways, at the cost of some inefficiency in the use of store. One possibility is to protect the space used in a block if an assignment has occurred during the execution of the block. Unfortunately, this method is not effective unless each array variable contains information about its scope in order to determine when the space associated with it can be freed. Even then the use of store is not very economical, since all the

COULOURIS

stack space used in the block must be reserved until the assigned array is freed.

It is apparent that no method short of a full dynamic store allocation scheme of type (iii) is adequate for the implementation of the compound data structures of CPL. However, the overheads of such a scheme, both in implementation and in execution speeds, are great, so the limitations of a simple stack have been accepted. The restriction imposed might be worded: 'Any assignment to an array becomes invalid on leaving the innermost block containing the assignment'. This is a restriction on entire arrays, not on the elements of arrays, which may be evaluated and assigned without restriction. A further penalty incurred by the adoption of a single stack is that on assignment to an array variable, the space containing the previous value of the array does not become available until the end of the array's scope.

RECURSION

In our discussion of store allocation we have not so far considered the possibility that variables may be re-activated during their own activation. This occurs when a routine or function is called recursively. Fortunately it can be easily handled both for the absolute store allocation scheme adopted for simple items and the single stack system used for compound items. In fact the operation of the dynamic stack is such that no special action is required for compound items. When a block is recursively entered, the space for the new activations is obtained from the top of the stack in the same way as before, and on exit from each recursive level, the store used is automatically made available. This is the property of stack storage which makes it so attractive for the implementations of block structure languages containing recursion.

The absolute store allocation scheme adopted for simple items raises few additional difficulties. We have already noticed that the space allocated to items local to a routine or function must be protected throughout the scope of the routine or function. If the routine or function may be recursively entered, we must provide for the preservation of a copy of the values of all the local variables before a recursive entry and for the restoration of the values on exit. In practice this requires the copying of the portion of absolute store allocated to the items local to a routine or function on to the stack whenever the routine or function is entered with its 'recursive level marker' greater than zero. All such vectors on the stack are linked in the form of a backward chain in chronological order of copying. Whenever we return from a recursive call we need only copy the last vector on the chain into its original position in the absolute store.

THE SCOPE OF LABELS

In our discussion of store allocation schemes, we mentioned that a transfer to a labelled command is possible from a position outside the block containing

PROBLEM-ORIENTED LANGUAGES

it. This is a feature of CPL which is not contained in most block structure languages, but as we pointed out this inclusion does not raise further problems in store allocation. However the inclusion of variables and expressions of type *label* further complicates the position; the destination of a transfer command cannot be determined during compilation. This produces considerable implementation problems. Every transfer command must perform an analysis of the position of its destination in the block structure and initiate the necessary activations of the blocks entered. This interpretive feature is eliminated if the scope of a label is restricted to the block containing it.

FURTHER PROBLEMS OF FUNCTIONS AND ROUTINES

In the implementation of the comprehensive facilities for the declaration and use of functions and routines within CPL, several features appear to demand some degree of interpretation. The problems of routines and functions are parallel so the discussion here is restricted to the case of functions for the sake of clarity.

CPL provides for the existence of functions as data items in their own right, appearing in any semantically meaningful context. We may, for example, construct and assign functions.

Functions may behave in one of two ways with respect to their free variables (i.e., variables referred to in the function body which are neither formal parameters of the function nor declared within the function body):

- (i) A *fixed* function evaluates the free variables at the time of activation of the function and associates an unchangeable set of values with the function throughout its activation.
- (ii) A *free* function uses the values of the free variables current when they are referred to during the evaluation of a call of the function. The values may vary from one call to the next. This treatment is analogous to the treatment of free variables in Algol procedures.

Since the free variables may be arrays or even further functions, and since a function may be changed by assignment during its activation, the storage problems associated with *fixed* functions are at least as great as those of arrays. The implementation of *free* functions on the other hand requires no special handling of free variables and the amount of store required by the function is fixed at compile time. *Free* functions may be handled within the absolute store allocation scheme described for non-compound items. *Fixed* functions have been excluded from the subset for this reason and because some degree of interpretation is required in their activation.

The assignment of functions, and the use of functions with functional results, is still possible within the subset, but in order to avoid the interpretation of actual parameter types and modes of call on application, the types and modes of the formal parameters are considered a part of the type of a function. This permits type matching during compilation.

CONCLUSION

The outcome of the project from which this paper stems has been the construction of a translator for a restricted subset of the CPL language. The most important restrictions imposed by the subset are the following:

1. The scope of labels does not extend outside the block in which they occur.
2. The use of array assignments is restricted.
3. Declarations of function and routine variables must include the types of all their parameters.
4. All functions and routines are *free*.

The subset is nowhere less powerful than the full Algol language, and includes all the syntactic features of full CPL.

This paper does not do justice to the many powerful concepts behind the structure of the CPL language. Nor does it examine all the problems of implementing the subset selected. It is hoped that the essential features of the techniques used, and their relevance to the selection of a subset, have been made clear. It is interesting to note that some of the methods used are often considered inadequate even for Algol implementation.

The achievement of at least some measure of success was due in no small part to the availability of the very powerful compiler-compiler system. The suggestion has sometimes been made that this system is not suited to the implementation of syntactically complex block structure languages. In the author's opinion, the ease with which this can be achieved is quite surprising, and in particular the project would not have been possible without the facility for recursive use of routines controlled by syntax analysis records. The other advantages of the compiler-compiler include the more obvious ones such as automatic syntax checks of source program, ease of documentation and correction of the compiler and useful monitoring facilities. The total effort involved in the project, including the provision of a library of basic routines and functions, has been of the order of two programmer years. This figure would probably have been trebled if any other available system had been used.

REFERENCES

- Barron, D. W., Buxton, J. N., Hartley, D. F., Mixon, E., & Strachey, C. S. (1963). The main features of CPL. *Comput. J.*, 6, 134-142. A fuller and more up-to-date description of CPL is available in *The CPL Reference Manual* (1966). Pergamon Research Group, Oxford.

PROBLEM-ORIENTED LANGUAGES

- Brooker, R. A., MacCallum, I. R., Morris, D., & Rohl, J. S. (1963). The Compiler Compiler. *A. Rev. autom. Progmng*, 3, 229-275.
- Coulouris, G. F., & Goodey, T. J. (1965). The CPL 1 System Manual. *ULICS Prog. Inf. Doc.*, 12.
- Iliffe, J. K., & Jodeit, J. G. (1962). A dynamic storage allocation scheme. *Comput. J.*, 5, 200.
- Randell, B., & Russell, L. J. (1964). *Algol 60 Implementation*. London: Academic Press.

INTERROGATION LANGUAGES

J. M. FOSTER

MINISTRY OF AVIATION
ROYAL RADAR ESTABLISHMENT
GREAT MALVERN, WORCS.

When a computer is working on-line with a man, each needs to understand the other quickly. Though men can learn new notations and computers need programming, there are some reasons why the machine should humour the man, perhaps making use of his ability to recognise a picture, to draw, to read or to write his own language. If the machine can read English, an English speaker can use it without training which he may be unwilling to afford, and use it more instinctively than if he had to cast his thoughts in an unusual way. It is natural to ask questions in English, if the information the computer has is English text. And if a computer can be made to understand, though very imperfectly, the English which a man uses, a modest step in our understanding of language may have been made.

Several programs have been written to read and to attempt to answer questions posed in various subsets of English about various forms of data. Simmons (1965) has an excellent review paper which discusses them and brings out the particular purpose and character of each. The purposes and problems which they try to solve are very variable. A program has been written for RREAC which will read a system of languages each of which is a subset of English. This paper is an account of its properties. There has been no attempt to understand English as a complete natural language. That problem is far greater.

The program has been written using the list-processing extensions to ALGOL available on RREAC.

FEATURES OF THE PROGRAM

The program has to read many languages and the emphasis has therefore been on isolating the problems in general and attempting to solve some of

them rather than on producing a particular example. The problems considered are listed below.

1. A large vocabulary is needed. In order to ask a reasonable variety of questions and to phrase them in a variety of ways natural to the user a vocabulary of 1000 words is rather small and several times this would be useful. This raises problems of dictionary look-up, size of computer store, organisation of backing store and associating a meaning or meanings with the words. This problem has not been treated in the program. Room exists for more than 1000 words, which is sufficient at present.

2. A grammar capable of defining a complex language is needed and a way of associating the meaning of the words and the constructions to extract the meaning of the sentences. This analysis process needs to be fairly fast and to obtain all the meanings of a sentence if it is ambiguous. The method used will be described below.

3. The program needs to interact with the user in order to ask for more data or for a clarification of the question or to point out that the sentence is not understood and to give some guidance about how it might be corrected. The need to clear up ambiguities is very great since English is a remarkably ambiguous language. As the man is on-line the computer can ask which meaning is wanted. If a man is to use the language with little training it will be necessary to accept wrong sentences and indicate why they are wrong. The program attempts to do these in a limited way.

4. The user should be able to extend the language by defining the meaning of new words and phrases in terms of old ones. Thus 'The age of a man means the time from his date of birth to the present date'. The program accepts definitions.

5. The program needs some deductive power. 'No man is wholly sane' and 'Jones is a man' implies 'Jones is not wholly sane'.

6. The program needs to be able to take things for granted. There is a vast mass of information in daily use by a man which could in principle be explicitly stored in the machine but which would tend to choke it. Monday is different from Tuesday, night and day are not simultaneous. It is particularly ideas of equality and classification that are important here. This is a very difficult problem.

7. Primitive basic procedures must be supplied for building blocks. The questions are really translated into programs which then examine the data to give the answers. These programs are made up from standard pieces which are known to be useful, such as deductive programs or programs for deciding whether an object is a member of a set.

8. Temporary and unused information should be forgotten.

Among these problems we have selected those particularly concerned with reading the questions (1 to 4) for study. For numbers 5 to 8, which are just as important and probably more difficult, fairly simple solutions have been accepted according to the example chosen. Thus for 5 a logical calculus

allowing only one quantifier is the most that has been used, in order to avoid the difficulties of theorem proving. Nothing has been done towards 6 and 8. For 7, simple set operations of intersection, union and mapping of sets on to sets have been used together with routines for printing answers.

In Simmons' review paper he classifies interrogation languages according to whether the data in the computer is English text or is represented in some way more easy to analyse. The information might be 'Jones is a man aged 21 who joined this organisation in 1964' or alternatively this might be represented in the store in some list-processing notation as:

```

name—Jones
sex  —male
age  —21
date of joining—1964

```

The examples have all been of the latter class though this need not have been so. If they had been text-based this would have raised the very difficult problem of extracting the information. The reader can see some of the difficulties by considering how a program could be written to obtain a sensible answer to the question, 'What does on-line mean?', from this paper.

The two following sections describe the way in which sentences are analysed, their meaning found and the users' grammar corrected and the way in which fresh definitions of phrases are added to the language.

THE ANALYSIS OF THE SENTENCES

The algorithm for analysing the sentences must be reasonably fast; it must produce all the analyses of an ambiguous sentence; it must deal satisfactorily with incorrect sentences and it must produce a useful analysis as its result. The method chosen is based on predictive analysis (Kuno & Oettinger 1963). It is applicable when all the syntax productions defining the language start with a basic symbol.

```
class-name → basic symbol, class-name, class-name . . .
```

For example, if the particular rule

```
noun-phrase → article, noun, adjectival clause
```

is replaced by the three rules

```
noun-phrase → a, noun, adjectival clause
```

```
noun-phrase → an, noun, adjectival clause
```

```
noun-phrase → the, noun, adjectival clause
```

then these latter rules would be eligible as constituents of a predictive analysis grammar. The notation means that the sequence on the right is an example of the grammatical class on the left. When all the rules have this form a particularly simple algorithm will analyse sentences in all possible ways. The algorithm is very suitable for use in a list-processing system which allows common sub-lists. The process follows:

PROBLEM-ORIENTED LANGUAGES

1. Start with one pile containing the class-name 'sentence'.
2. Read in a word of the sentence. Then for each of the piles now in existence proceed as follows:

(a) If the head of the pile is a particular class-name look at the rules defining this class-name; if n of them begin with the word read in then make n new piles consisting of the rest of the rule and the rest of the pile being considered. If the head of the pile is a basic symbol and equal to the word just read in make a new pile consisting of the remainder of that pile.

(b) Do this for all the piles collecting the new piles generated.

3. If any new piles are empty these represent analyses of the sentence. Otherwise replace the collection of piles by the new collection and repeat 2. Consider as a simple example the grammar

sentence	→ a.	1
sentence	→ a + expression .	2
expression	→ a	3
expression	→ a + expression	4

and the sentence

a + a + a + a.

By rule 1 we start with the pile

sentence

and by rule 2 we obtain the piles after reading 'a'

·
+ expression.

on reading '+' this reduces by the second part of 2(a) to
expression.

then to

·
+ expression

and so on until finally in the position

·
+ expression.

the '.' is read which forms an empty pile and stops the analysis.

The algorithm works by carrying along in the piles all the possible continuations it can expect on the basis of the words it has read so far. Hence it will obtain all the analyses of an ambiguous sentence, as these will all reduce to zero simultaneously. An interesting example of ambiguity is given by Kuno (1965). In his grammar, the sentence, 'People who apply for marriage licences wearing shorts or pedal pushers will be denied licences', has 40 different analyses. The algorithm will not be fast unless care is taken. Two points are important, the analysis of phrases should not be repeated many times and words must initially be replaced by their class-names. If a sentence contains several ambiguous phrases separate from each other, say n of them,

there is a danger of repeating the analysis of the last one about 2^n times. Kuno (1965) gives some suggestions for partially avoiding this. And if several rules, while different in their later constituents, start with the same class-name then the analysis of this may be repeated several times. If these in turn start with the same constituent and this continues, one will run rapidly out of store. This has been avoided by care in writing the rules. The stringency of the requirement that all rules must begin with a basic symbol must be relaxed, since it would imply that if a rule began with a noun, the rule would have to be repeated with all possible nouns. This can be avoided by pretending that 'noun' is a basic symbol and arranging that the words are changed into the right parts of speech as they are read in.

Though there are good reasons for using the method we are particularly interested in the way in which it treats incorrect sentences. If the sentence does not belong to the grammar, then at some stage in the analysis process no new piles will be generated and yet none of the old piles will be empty. The point at which this failure occurs will not necessarily be the point where the human will think that he made the error, though it often will be. Since the process operates by carrying all possible continuations it is able at the stage where it detects an error to generate some further continuations on the basis of the errors that are likely to be made. These continuations are then used to continue the analysis. Some will be rejected but some may come to a successful conclusion. Alternatively the analysis may run out again and the process can be repeated.

The particular errors which are accounted for by the program in use are as follows:

1. The user has omitted a word. Then for each of the piles it can be assumed that the omitted word corresponded to the topmost basic symbol. The word read is then used again.

2. The user has written one word and meant another. Then for each pile it can be assumed that the word read was equal to the topmost basic symbol.

3. The user has written an extra word. It is ignored and the old piles used for the next word.

All these can be done simultaneously. It should be remembered that words can be replaced by parts of speech.

Consider again the simple grammar and the input sentence

$$a + b + a.$$

Then at the stage where the only pile is

expression

a 'b' will be read instead of an 'a'.

By Rule 1 it will be assumed that a word has been omitted and 'b' will be tried on

+ expression.

PROBLEM-ORIENTED LANGUAGES

These will fail. Rule 2 creates the piles

+expression

on the grounds that the 'b' was an 'a' and Rule 3 will leave 'expression' as an active pile. So analysis will proceed with

+expression

expression

as the piles, all but the second of these being removed by the '+' which is read next. The computer can say

'Did you mean a+a+a?'

This question will also be asked in reply to the input sentences

a+ba+a

and

a a

and many others.

Taking a more realistic example in English

'All mimsy were the borogroves.'

might provoke,

'Did you mean "All adjective were the plural-noun?"'

The program also can replace a group of words by a class-name, with an extension of Rule 3.

The way in which the analysis of a sentence is used to obtain a meaning for it depends on the fact that the syntactic groupings of words into phrases, clauses, etc., correspond to the groupings of meanings. In the sentence,

'Who is the Prime Minister?'

the syntactic analysis groups 'the Prime Minister' together. If there exist a correspondence between

'Who is X ?'

where X is a noun-phrase, and a piece of program which will obtain the answer given the meaning of X , then this will suffice to answer questions like,

'Who is the officer of the watch?'

and

'Who is the most senior officer present?'

provided that the programs associated by the analysis with the three noun-phrases all have their 'results' in the same form. For example, adjectives restrict the class of objects to which they refer.

'Who is the noun adjectival-clause?'

will always mean examine the set of objects called noun, say all men, and apply the function 'adjectival-clause' to each member selecting those for which

the result is true. So if in this construction, nouns always have sets as values and adjectival clauses always have truth-functions as values, the same analysis will work. Since we are working in a field where the value of an operation can be a function, as for the adjectival clauses, the notation of the λ -calculus is useful. All that is necessary here is to remark that

$$(\lambda xy. B)CD$$

for example, means substitute C for x and D for y throughout B . The λ -calculus is used in the program to represent meanings. The analysis of

'Who is the man who joined in 1964?'

is

$$(\lambda x. \text{Who is } x)((\lambda yz. \text{restrict set } y \text{ by function } z) \\ (\text{set of men})(\lambda t. t \text{ joined in 1964}))$$

where 'who is x ' means 'print out the set x '; 'restrict set y by function Z ' is the operation described above and ' t joined in 1964' is a truth function having the value true if the argument (in this case each member of the set of men) joined in 1964. The function, ' t joined in 1964', could in turn be expressed as

$$\text{equal}(\text{year of joining}(t), 1964).$$

The output of a predictive analyser very easily yields a representation like this, since the natural output is in a form which can be thought of as a Reverse Polish form of the analysis of the meaning of the sentence into functions and arguments. Rule 2 of the analysis algorithm can be modified as follows:

2. Read in a word of the sentence. Then for each of the piles now in existence proceed as follows:

(a) If the head of the pile is a particular class-name look at the rules defining this class-name; if n of them begin with the word read in then make n new piles consisting of the rest of the rule and the rest of the pile being considered, and associate with each of these piles a result pile formed by adding something corresponding to the rule to the top of the result pile associated with the old pile. If the head of the pile is a basic symbol and equal to the word just read in make a new pile consisting of the remainder of that pile.

(b) Do this for all the piles collecting all the new piles and their associated result piles.

Consider again the simple grammar and make the letters w, x, y, z correspond to rules 1, 2, 3 and 4 respectively. Then the analysis of

$$a + a + a + a$$

is $yzzx$, the top being towards the left. These letters actually correspond to functions which might be

$x = \text{print}(a + \text{result so far})$

$z = \text{add}(a, \text{result so far})$

$y = \text{let the result so far be the value of } a.$

Then carrying out these functions from left to right will print out the value of $a+a+a+a$. The analysis of

'Who is the man who joined in 1964?'

is in Reverse Polish as produced by the analyser

$(\lambda t. t \text{ year of joining } * 1964 \text{ equal } *) (\text{set of men})$
 $(\lambda yz \text{ restrict set } y \text{ by function } 2) * (\lambda x. \text{who is } x) *$

The method of evaluation is like that of Landin (1964) modified to a Reverse Polish form. It is roughly as follows:

1. Place the first λ -expression on top of the evaluation stack.
2. Place the set of men on top of the stack.
3. Place the next λ -expression on top of the stack.
4. Apply the function on top of the stack to its arguments, taken in this case from the next two places on the stack (* is used to mean apply) and leave the result at the top of the stack.
5. Place the next λ -expression on the top of the stack.
6. Apply it to its argument.

The result of analysing a sentence is an expression which when evaluated does what is required. This need not be to answer a question but could be to add new information to the store.

THE DEFINITIONS OF NEW MEANINGS

We have seen that in this system the meaning of a word or a phrase is either a function or something which has a value like a set, a number or a string of characters. To define new constructions we have therefore to give a syntactic specification of them together with a meaning, which must be expressed in terms of what already exists.

If it is a single word whose meaning is being defined then the definition must have a form

' X , which is a Y , means Z '

where X is being defined; Y is the class to which it belongs and Z is its meaning. It is only necessary then to analyse the phrase Z as a Y and take the analysis (without obeying it) as the meaning of X . Then if a sentence occurs with X in it, all will be exactly as if the phrase Z had been written there.

A more complicated situation exists when a phrase is being defined which may contain constituents which must not be explicitly specified. For example, we might define the meaning of average by

'The average of x over y where x and y are noun phrases, is a noun phrase and means the sum of x over y divided by the number of members of y '

Then if we say 'The average of salary over the set of men' we want this to be treated as

'The sum of salary over the set of men divided by the number of members of the set of men'.

This is slightly more complicated than the previous case because the definition must be analysed with any noun phrases for x and y . However, the important point is that the first phrase has two arguments and the second three, so the real equivalent of

'The average of x over y '

is

$(\lambda xy. \text{ the sum of } x \text{ over } y \text{ divided by the number of members of } y)^*$

APPLICATIONS OF THE PROGRAM

So far three examples of the use of the program have been made. One was merely to test out the ideas of the previous section, which were demonstrated on a very simple example. It is interesting to note that the method of definition can be used to define a new method of definition.

Another example was concerned with supplying information and answering questions about simple staff matters such as

'Who is in division x and joined before the men in division y joined?'

A vocabulary of about 50 words was enough for this simple example, together with about 50 syntax rules. The primitive operations, out of which the meanings were built, were set operations and operations to print answers.

The third example was much more elaborate and had an essentially unrestricted vocabulary. It accepted a sequence of statements, such as

'No nice puddings are wholesome'

'Every father is a man'

and would then attempt to answer questions like

'Is it true that Jones is a man?'

on the basis of the information it had. If it failed to do so it asked questions to determine the truth or falsity, such as

'Is it true that Jones is a father?'

The logical calculus was restricted to a single quantifier.

CONCLUSIONS

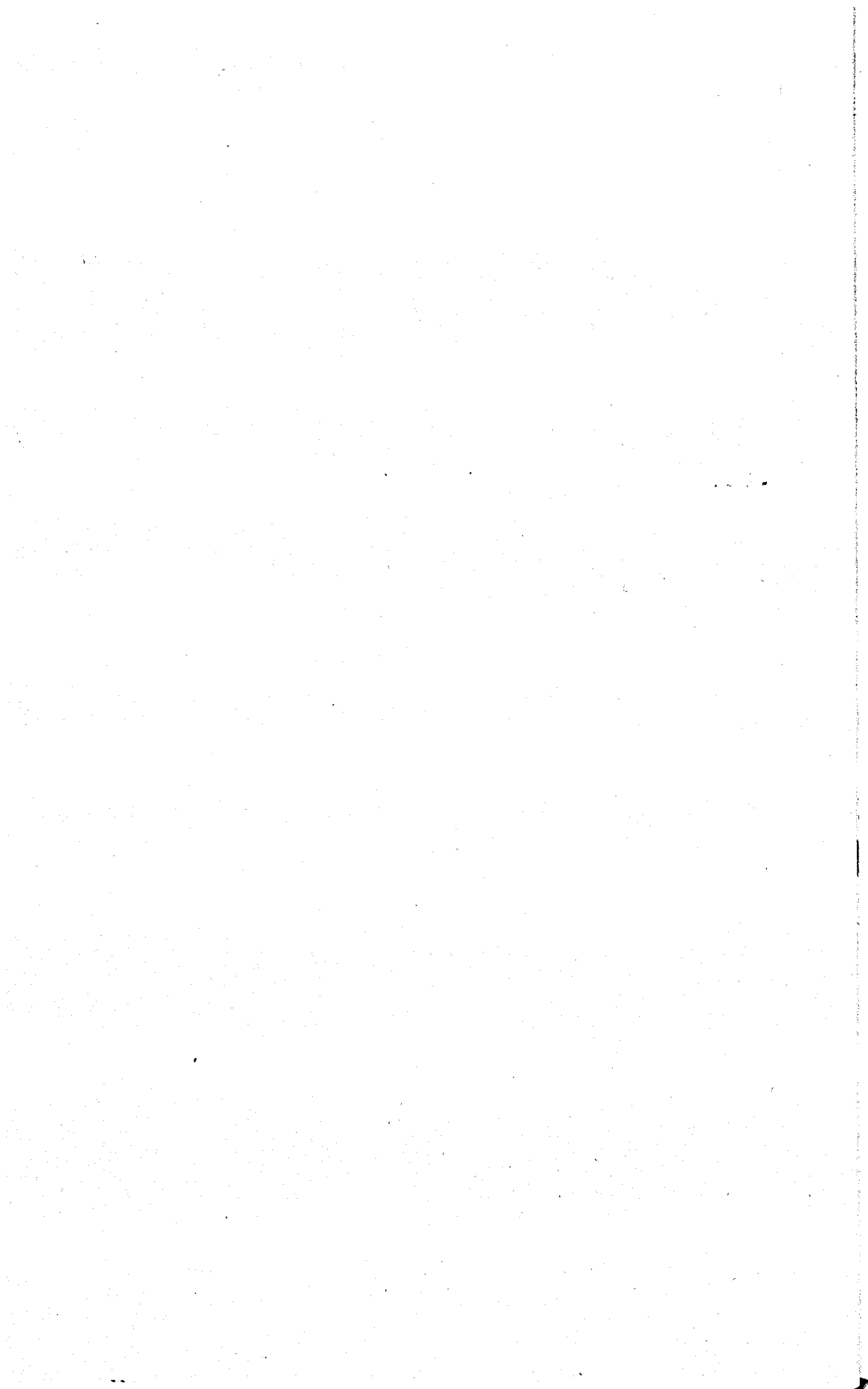
A way of interpreting the meaning of sentences in terms of λ -expressions has been shown to be useful in some cases, particularly for defining the meaning of new phrases. Predictive analysis has given a useful indication of errors in sentences.

The problems treated so far enable a reasonable input in English to a computer but a very large amount of work needs to be done on what to do with information in the computer.

REFERENCES

- Kuno, S. (1965). The Predictive analyser and a path elimination technique. *Communs Ass. comput. Mach.*, 8, 453-462.
- Kuno, S., & Oettinger, A. G. (1962). Multiple-path Syntactic Analyzer. *Information Processing 1962—Int. Conf. Inf. Process.*, pp. 306-311. Amsterdam: North-Holland.
- Landin, P. J. (1964). The mechanical evaluation of expressions. *Comput. J.*, 6, 308-320.
- Simmons, R. F. (1965). Answering English questions by computer: a survey. *Communs Ass. comput. Mach.*, 8, 53-69.

MACHINE
INTELLIGENCE 2



MACHINE INTELLIGENCE 2

EDITED BY
ELLA DALE

AND

DONALD MICHIE

DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION
UNIVERSITY OF EDINBURGH

EDINBURGH at the University Press

PREFACE

The readers of this book will very probably be familiar with its predecessor, *Machine Intelligence, I*, published under the same imprint and edited by N. L. Collins and Donald Michie, and so they will be aware of the fascination and familiar with the wide extent of the field of enquiry being investigated here, difficult as it may be to define it precisely.

The objectives vary exceedingly. We may start with: how to make a machine which can do what we could do (so that we will be free to do something else); how to make a machine which can do what we could do if we were more accurate and patient (so that we can eliminate human error); how to make a machine which can do what we could do, but do it more quickly (so that we can learn the answer to the problem before the problem changes).

From this we advance to: how to make the machine (and ourselves) more efficient on the average by defining sub-goals, by using sub-optimal search procedures, by throwing away information totally, partially, or temporarily, for the sake of getting something done within an acceptable time.

Next we have: how can we assist the machine to recognise limited successes; how can we use the machine's current experience to suggest more appropriate sub-goals; how in fact can we and the machine best act together?

We are now confronted with the questions: how can we converse with the machine during the period of operation; what is a suitable language for such a conversation; what can the machine best do, what can we best do, and how should our respective rôles be determined?

Thus a study of 'machine intelligence' leads after yet one more remove to the ancient imperative, 'Know thyself', and the universal problem of co-existence.

1967

DAVID KENDALL

CONTENTS

PREFACE	PAGE v
INTRODUCTION	ix
ABSTRACT FOUNDATIONS	1
1 Semantics of assignment: R. M. BURSTALL	3
2 Some transformations and standard forms of graphs, with applications to computer programs: D. C. COOPER	21
3 Data representation—the key to conceptualisation: D. B. VIGOR	33
MECHANISED MATHEMATICS	45
4 An approach to analytic integration using ordered algebraic expressions: L. I. HODGSON	47
5 Some theorem-proving strategies based on the resolution principle: J. L. DARLINGTON	57
MACHINE LEARNING AND HEURISTIC PROGRAMMING	73
6 Automatic description and recognition of board patterns in Go-Moku: A. M. MURRAY and E. W. ELCOCK	75
7 A five-year plan for automatic chess: I. J. GOOD	89
8 New developments of the Graph Traverser: J. DORAN	119
9 BOXES: an experiment in adaptive control: D. MICHIE and R. A. CHAMBERS	137
10 A regression analysis program incorporating heuristic term selection: J. S. COLLINS	153
COGNITIVE PROCESSES: METHODS AND MODELS	171
11 A limited dictionary for syntactic analysis: P. BRATLEY and D. J. DAKIN	173
PROBLEM-ORIENTED LANGUAGES	183
12 POP-1: an on-line language: R. J. POPPLESTONE	185
13 Self-improvement in query languages: J. M. FOSTER	195
14 POP-2 reference manual: R. M. BURSTALL and R. J. POPPLESTONE	205
INDEX	251



INTRODUCTION

What is Machine Intelligence? Is it a theory, a discipline, an engineering objective? Or is it a pretentious name for the more peculiar parts of computer science?

In our view the matter is quite clear. Since the objective is to bring into existence an intelligent machine, those engaged in the attempt must look on themselves as engineers. The relationship of this particular brand of software engineering to the disciplines of psychology, logic, mathematics and computer science is thus the same as that which bridge-building bears to the fields of geometry, mechanics, materials science and hydrodynamics. There is a simple test of success in bridge-building. Walk across it, and you know that the bridge exists. If you can drive a twenty-ton truck across, the odds are that it is also well built. The equivalent test of an intelligent machine is, as Turing pointed out long ago, the test of rational conversation. No contemporary computer system can yet stand this test, but the time is now fast approaching when the first intelligent dialogues between man and machine will commence.

Growing awareness of this finds a certain reflection in the pattern of this new volume. The previous year's themes are there—both formal theory of programs and programming, and new experimentation with deductive and inductive systems. But the emphasis in the last four chapters on vehicles of communication, is more prominent than before, and marks a significant trend.

Once again our thanks are due to Oliver and Boyd; also to the University of Edinburgh for rewarding the labours of this Second Machine Intelligence Workshop with a liberal issue of refreshments at half time.

ELLA DALE
DONALD MICHIE

ABSTRACT FOUNDATIONS



1

SEMANTICS OF ASSIGNMENT

R. M. BURSTALL

EXPERIMENTAL PROGRAMMING UNIT
DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION
UNIVERSITY OF EDINBURGH

FORMAL SEMANTICS AND THE LAMBDA-CALCULUS

The past two or three years have seen a growing interest in the formal semantics of programming languages. The aims of these semantic investigations are twofold. First they give a means of specifying programming languages more precisely, extending the syntactic precision which was first achieved in the Algol report to murkier regions which were only informally described in English by the Algol authors. Secondly they give us new tools for making statements *about* programs, particularly for proving that two dissimilar programs are in fact equivalent, a problem which tends to be obscured by the syntactic niceties of actual programming languages.

Like most terminological questions, the dividing line between syntax and semantics has been the subject of much arid debate. The sense in which the words are used in this paper is not mysterious. Suppose that the definition of a programming language 'XTRAN' is published in two parts, 'Part A: Syntax' and 'Part B: Semantics'. After reading Part A, the reader should be able to say, when he is presented with a piece of text, 'Yes, this is an XTRAN program' or 'No, it is not'. But the program, given data, would print some results. The reader should be able to predict what these results would be when, and only when, he has read 'Part B: Semantics'.

One of the important aspects of programming language semantics is that it tries to connect the very empirical subject of programming with the main body of mathematics, particularly with mathematical logic. The part of logic which is most closely related to programming is that dealing with functions

and computability. P. J. Landin (1964, 1965, 1966a and b) and C. Strachey (1965) have sought to explain programming languages in terms of Church's calculus of lambda conversion (Church 1941, Rosenbloom 1950) also using ideas from Curry's combinatory logic (Curry & Feys 1958). The relevance of the lambda-calculus to programming and the use of recursive function definitions in proving equivalences were first brought out by McCarthy (1962, 1963).

A function can be described in two ways: first by saying how it is built out of certain simpler functions by combining them in various ways, secondly by saying what values it would yield when applied to various arguments. The lambda-calculus gives an unambiguous notation for describing the way a function is constructed from simpler primitive functions, and gives conversion rules for converting a function into another which is differently constructed in terms of the primitives but which would nevertheless give the same result when applied to the same arguments. There are simple and fully formalised rules for evaluating expressions of the lambda-calculus; if we can translate the application of a program to its data into an expression of this calculus we have fully specified the outcome.

Landin's 1965 and 1966b papers and Strachey's paper already cited are devoted to the task of translating Algol and CPL respectively into the lambda-notation. The parts of programming languages furthest from the purely functional lambda-notation are the imperative features: assignment and jumps. Landin translated Algol into a version of the lambda-notation which was extended to include assignments; these assignments were defined in terms of a 'sharing machine'. Strachey showed how CPL (or Algol for that matter) could be translated into purely functional lambda-notation by introducing a 'store' with appropriate 'contents' and 'update' operations defined on it. Although these methods serve well as a formal definition of the languages concerned, they leave something to be desired as a means of simplifying programs and proving equivalences between them. Landin (1966) gives a number of equivalence rules but these do not apply to the assignment expressions in his system. Strachey's translations in terms of a store lead to elaborate and rather impenetrable expressions as the translations of quite straightforward piece of CPL text.

The first part of this paper gives methods of translating many instances of assignment in ordinary programming languages into purely functional notation, producing relatively simple and manipulable expressions as the result. In the second part a method similar to Strachey's is used to tackle the problems arising in procedures with non-local variables and in assignment to data-structures with shared components.

This second method of translation is more powerful and general, but it produces more complex translations. It uses the concepts of address and storage location, in an abstract form. For the simpler uses of assignment, such as assignment to a temporary variable to avoid repeated evaluation of a common sub-expression or repeated assignments to a variable in a loop,

the more general method is unnecessary; the same effect can easily be obtained by a functional expression not involving addresses or locations. But in more complex cases, such as certain forms of non-local variable, call by 'reference' ('simple name') and data structure manipulation, the ideas of addresses and locations cannot be dispensed with by a rearrangement of the program before it is run.

I use Algol as the programming language to be translated, because it is well known and well documented, and extend it in various ways to make points about features of other programming languages. I have talked above about Church's 'lambda-notation'. It is not really the notation which concerns us here, but the formal system of the lambda-calculus. In fact I will translate into a dressed up (syntactically sugared) notation for expressions of the lambda-calculus, due to Landin. This also includes conditional expressions and operations on lists, all of which may be reduced to pure lambda-notation if desired.

Although there are still a number of unresolved questions about the techniques used in this paper I hope that it may help to stimulate further discussion of formal systems for expressing the semantics of programming languages.

ISWIM—A MORE INTELLIGIBLE VERSION OF THE LAMBDA-NOTATION

Landin (1966a) has given a detailed formal definition of a functional programming language which he calls ISWIM (If you See What I Mean). This language apart from two features can be rewritten in the lambda-notation of Church in a quite straightforward way, using the relationship between definitions and the lambda-notation explained by Landin (1965, 1966a). The features which cannot easily be rewritten are 'program points' and 'assignment'. Program points are a means of translating labels and hence they are not relevant to the subject matter of this paper. My aim here will be to translate Algol, with some extensions, into ISWIM without assignment. This is equivalent to translating into the lambda notation but the translations will be more readable. As Landin (1966a) has pointed out, the purely functional ISWIM expressions which will be obtained are susceptible to various transformations which preserve equivalence. Thus the translation into functional ISWIM could provide a means for proving that two Algol programs are equivalent.

I hope that the following examples will make clear the meaning of expressions in this very simply structured language without repeating Landin's formal definition. In the interests of clarity I have taken one or two slight liberties with Landin's notation as follows:

- (a) Shape of brackets. Landin uses only parentheses. I will use parentheses for arguments of functions, diamond brackets ('<' and '>') for explicitly written lists and square brackets to indicate the scope of operators. A vertical stroke will be used above and below square

ABSTRACT FOUNDATIONS

brackets as a meaningless, purely decorative character to assist the eye.

- (b) Landin uses 'h' and 't' for the list selector operations (LISP 'CAR' and 'CDR'); I will use 'head' and 'tail'.
- (c) The meaningless, purely decorative word 'result' will sometimes be used when an expression is preceded by a number of definitions, e.g.,
[let $x=9$; let $y=u+v$; result $\langle x, y, u \rangle$]
- (d) I will abbreviate 'undefined' to 'undef'.

ISWIM expressions will be written on the left and their values on the right.

(i) Basic functions

I assume for purposes of illustration the arithmetic operators '+', '-' and 'x' and the list-processing functions 'cons', 'head' and 'tail'. The empty list will be denoted by 'nil'.

(ii) Conditional expressions

These are written ' $p \rightarrow x, y$ ' (cf. Algol 'if p then x else y ').

- (a) $1 < 2 \rightarrow 3, 4$ 3
- (b) $1 > 2 \rightarrow 3, 4$ 4

(iii) Definitions

Definitions are introduced by 'let' and a sequence of auxiliary definitions may precede any expression.

- (a) let $x=2$; $1+x \times x$ $1+2 \times 2$
- (b) let $x=1$ and $y=2$; $x+y$ $1+2$
- (c) let $x=1$; let $y=x+2$; let $x=4$;
result $x+y$ $4+3$

Note that the definitions may refer to variables previously defined. Remember that 'result' is a decorative word which may be omitted.

- (d) let $x=1$;
let $x=2$ and $y=x$;
result $\langle x, x+y, y \rangle$ $\langle 2, 2+1, 1 \rangle$

Note that in the second line x and y are defined in parallel and y takes the previous value of x . The result is a list of 3 items.

'let $x=2$ and $y=x$ ' is also written 'let $\langle x, y \rangle = \langle 2, x \rangle$ '.

- (e) let $f(x)=x-1 \rightarrow 0, x+1$;
 $f(3)$ 4
- (f) let $f(y)=y \times y$;
let $f(x)=x-1 \rightarrow 1, x \times f(x-1)$;
 $f(4)$ $4 \times 3 \times 3$

Note the definition of f is *not* recursive. It uses the previous version of f .

- (g) let rec $f(x)=x-1 \rightarrow 1, x \times f(x-1)$;
 $f(4)$ $4 \times 3 \times 2 \times 1$

BURSTALL

Note: 'rec' signifies that the f in the right of the definition is the same as that being defined.

(h) $\text{let } f(x) = [\text{let } g(y) = y \times y; \quad |$
 $\quad \quad \quad | \text{let } z = x + 1; \quad |$
 $\quad \quad \quad | \langle z, g(g(z)) \rangle \quad |$
 $f(3) \quad \quad \quad \langle 4, (4 \times 4) \times (4 \times 4) \rangle$

(i) $\text{let } x=0 \text{ and } y=0;$
 $\text{let } \langle x, y \rangle = [\text{let } x = x + 1; \quad |$
 $\quad \quad \quad | \text{let } y = y + 2; \quad |$
 $\quad \quad \quad | \text{result } \langle x, y \rangle \quad |;$
 $\text{result } x + y \quad \quad \quad 1 + 2$

It is important for what follows that the reader should grasp the sequence of definitions here. This expression could equally well be written:

$\text{let } x=0 \text{ and } y=0;$
 $\text{let } \langle u, v \rangle = [\text{let } p = x + 2; \quad |$
 $\quad \quad \quad | \text{let } q = y + 2; \quad |$
 $\quad \quad \quad | \text{result } \langle p, q \rangle \quad |;$
 $\text{result } u + v \quad \quad \quad 1 + 2$

(iv) Printing

The word 'print' followed by an expression causes it to be printed

(a) $\text{print } [\text{let } x=2; \langle x, x \times x \rangle]$ prints 2, 4

TRANSLATION OF SIMPLE ASSIGNMENTS AND CONDITIONAL STATEMENTS

What does the following piece of Algol program do?

$x := 1;$
 $y := 2;$
 $x := x + y;$
 $y := y + 3;$

If we run it and watch what happens, the answer would be 'nothing', at least nothing is printed. More usefully we may say that two new values are computed, and x and y are defined to take these values in what follows. I suggest that we should *not* try to find an ISWIM expression as a translation or explanation of this piece of code. It corresponds rather to a fragment of an ISWIM expression, thus

$\text{let } x = 1;$
 $\text{let } y = 2;$
 $\text{let } x = x + y;$
 $\text{let } y = y + 3;$

This is a sequence of ISWIM definitions which could equally well have been written

$\text{let } u = 1;$
 $\text{let } v = 2;$

ABSTRACT FOUNDATIONS

let $x = u + v$;

let $y = v + 3$;

The definitions only make sense if x and y are used in some following expression. Thus we have a correspondence between

<i>Algol</i>	<i>ISWIM</i>
begin integer x, y ;	print [let $x = \text{undef}$ and $y = \text{undef}$;
$x := 1$;	let $x = 1$;
$y := 2$;	let $y = 2$;
$x := x + y$;	let $x = x + y$;
$y := y + 3$;	let $y = y + 3$;
outinteger($1, x \times y$)	result $x \times y$
end	

This transcription seems so simple that one's first reaction is that it must be wrong. However, it is possible to push this correspondence between assignments and definitions a surprisingly long way. To do this we need the concept of the 'assignable variables' of a piece of Algol program. We define these as a list of all the variables which are, or might be, assigned to. We may then say that a piece of Algol program produces a list of new values for its assignable variables.

This is immediately useful when we consider conditional statements.

<i>Algol</i>	<i>ISWIM</i>
if $x > 0$ then	print [let $\langle x, y, z \rangle = [x > 0 \rightarrow$
begin $x := x + 1$;	[let $x = x + 1$;
$y := x + y$	[let $y = x + y$;
end	$\langle x, y, z \rangle$]
else	,
begin $x := x + 2$;	[let $x = x + 2$;
$z := x + 1$	[let $z = x + 1$;
end;	$\langle x, y, z \rangle$];
outinteger($1, x + y + z$)	$x + y + z]$

The conditional has $\langle x, y, z \rangle$ as assignable variables and it is translated into a definition of x, y and z which has, as right-hand side, a conditional expression denoting a triple. This may be clearer if we distinguish the different variables all called x above, by renaming them x_1, x_2 , etc.

```

print [let  $\langle x_4, y_4, z_4 \rangle = [x_1 > 0 \rightarrow$   |
          |[let  $x_2 = x_1 + 1$ ;  |
          |[let  $y_2 = x_2 + y_1$ ;  |
          | $\langle x_2, y_2, z_1 \rangle$   |]
          |,
          |[let  $x_3 = x_1 + 2$ ;  |
          |[let  $z_3 = x_3 + 1$ ;  |
          | $\langle x_3, y_1, z_3 \rangle$   |];
           $x_4 + y_4 + z_4]$ 

```

In general we may let X and Y be lists of variables and $X::Y$ be the list formed by merging the elements of X and Y without repetitions. We assume variables are in some canonical order, e.g., alphabetical or order of declaration.

We let $S\{X\}$ be an Algol statement or sequence of statements with X as its list of assignable variables, and $D\{X\}$ be a sequence of ISWIM declarations which translate it. We use E for an Algol or ISWIM expression.

(i) If $S\{X\}$ is ' $x_1:=x_2:=\dots x_k:=E$ ' where $\langle x_1, \dots, x_k \rangle$ is X then $D\{X\}$ is 'let $x_1=E$ and $x_2=E$ and $\dots$ and $x_k=E$ '.

(We will discuss later the case where evaluating E produces side-effects.)

(ii) ' $S1\{X\}; S2\{X\}$ ' corresponds to ' $D1\{X\}; D2\{X\}$ '

(iii) ' E then $S1\{X1\}$ else $S2\{X2\}$ ' corresponds to
let $X1::X2=[E\rightarrow[D1\{X1\}; X1::X2], [D2\{X2\}; X2::X2]]$.

(iv) ' $\text{if } E \text{ then } S\{X\}$ ' corresponds to 'let $X=[E\rightarrow[D\{X\}; X], X]$ '.

Notice that we have assumed so far that the left-hand side of an assignment is a simple variable and not a subscripted variable, a procedure identifier or a formal parameter called by name.

LOOPS

We have dealt with the simplest use of assignment, which is to evaluate once and for all an expression which would later occur in several places. Another use of assignment is to update a counter or accumulate a result in a loop. We will translate a loop by a recursive function.

$i:=1; x:=0;$	print let $i=1; \text{let } x=0;$
loop: if $i>10$ then goto out;	let rec loop (i, x)= $[i>10\rightarrow\langle i, x \rangle,$
$x:=x+i \times i;$	let $x=x+i \times i;$
$i:=i+1;$ goto loop;	let $i=i+1;$
out: outinteger(1, x)	loop (i, x)
	let $\langle i, x \rangle = \text{loop } (i, x);$
	result x

The body of the loop has as assignable variables i and x . We need to replace it with a piece of program which produces new values of these assignable variables. The function which produces them will also need to take the old values as parameters. In general

loop: if E then goto out;	let rec loop(X)= $[E\rightarrow X,$
$S\{X\}$ goto loop;	$D\{X\}; \text{loop}\{X\}$
out:	let $X=\text{loop}\{X\}$

We will not consider more elaborate loops as these would need a proper treatment of labels and this is outside the scope of the present paper, which deals with assignment. There seems to be no reason why the present treatment of assignment could not be combined with Landin's treatment of labels (Landin 1966), but this could bear further investigation.

ABSTRACT FOUNDATIONS

Let us consider for statements

for $i := E1$ step $E2$ until $E3$ do $S\{X\}$

```
let rec loop( $\langle i \rangle :: X$ ) =
  [let  $i = E1$ ;
   |( $i = E3 \times \text{sign}(E2) > 0 \rightarrow \langle i \rangle :: X$ ,
   |[ $D\{x\}$ ; loop( $\langle i + E2 \rangle :: X$ )]
  ]
let  $\langle i \rangle :: X = \text{loop}(\langle E1 \rangle :: X)$ ;
let  $i = \text{undef}$ 
```

Here $D\{x\}$ is the translation of $S\{x\}$ and we have assumed that $E1$, $E2$ and $E3$ have no side effects. If they do have side effects the translation has to be extended to redefine their assignable variables.

There is no difficulty in translating other kinds of for statement in a similar manner.

PROCEDURES

Let us see whether we can deal with procedures by the methods just elaborated. First we will try a non-type procedure with parameters called by value. Suppose that a procedure f has as formal parameters a list X and as non-locals a list Y .

Algol

```
procedure  $f(X)$ ; value  $X$ ;
   $S\{X :: Y\}$ ;
. . . . .
 $f(X1)$ ;
 $f(X2)$ ;
```

ISWIM (incorrect translation)

```
let rec  $f(X) =$ 
  [ $D\{X :: Y\}$ ;  $Y$ ];
. . . . .
let  $Y = f(X1)$ ;
let  $Y = f(X2)$ ;
```

In general this will not work correctly because of the non-locals. This becomes evident if we look at a specific example.

```
procedure  $f(x)$ ; value  $x$ ;
   $a := a + x$ ;
   $a := 0$ ;
   $f(1)$ ;
   $f(2)$ ;
outinteger(1,  $a$ )
```

```
print let rec  $f(x) =$ 
  [let  $a = a + x$ ; result  $a$ ];
  let  $a = 0$ ;
  let  $a = f(1)$ ;
  let  $a = f(2)$ ;
  result  $a$ 
```

The trouble becomes obvious if we number off the different variables which are all called a . This change in the identifiers used will not affect the meaning of the expression.

```
print let rec  $f(x) =$ 
  [let  $a2 = a1 + 2$ ; result  $a2$ ];
  let  $a3 = 0$ ;
  let  $a4 = f(1)$ ;
  let  $a5 = f(2)$ ;
  result  $a5$ 
```

The variable $a1$ takes on the value which it has when f is defined and not the value of $a3$ or $a4$ as it would have to in a correct translation of the

Algol text. The point is that the ISWIM function is a quite definite entity which always has the same effect whenever it is used, e.g., if it is the function 'add two' then it always adds two, whereas an Algol **procedure** may add two at one time when it is called and subtract three the next, because there has been an assignment to one of its non-locals meanwhile.

We can deal with the non-locals by letting them appear as extra parameters of the function in the ISWIM translation. Where assignments to some of these non-locals take place we will let the **procedure** produce as a result a list of the values of its assignable variables. This will give us the power to handle non-locals properly without introducing the concept of assignment into the ISWIM translation.

```

procedure  $f(x)$ ; value  $x$ ;
   $a := a + x$ ;
   $a := 0$ ;
   $f(1)$ ;
   $f(2)$ ;
  outinteger(1,  $a$ )

```

```

print let rec  $f(x, a) =$ 
  [let  $a = a + x$ ; result  $a$ ];
  let  $a = 0$ ;
  let  $a = f(1, a)$ ;
  let  $a = f(2, a)$ ;
  result  $a$ 

```

Algol type **procedures** present no difficulty. If they have side-effects they produce, as well as their obvious result, a list of the new values of their assignable variables.

```

integer procedure  $f(x)$ ; value  $x$ ;
  begin  $a := a + x$ ;
     $f := a \times x$ 
  end;
   $a := 0$ ;
   $y := f(1)$ ;
   $y := y + f(2)$ ;
  outinteger(1,  $a + y$ )

```

```

print let rec  $f(x, a) =$ 
  [let  $a = a + x$ ;
   result  $\langle a \times x, a \rangle$  ];
  let  $a = 0$ ;
  let  $\langle y, a \rangle = f(1, a)$ ;
  let  $\langle e, a \rangle = f(2, a)$ ;
  let  $y = y + e$ ;
  result  $a + y$ 

```

Note that where a type **procedure** with side-effects is called as part of an expression, as in ' $y + f(2)$ ' above, it is necessary to decompose the expression and introduce a subsidiary definition. If several such sub-expressions occur the sequence of the subsidiary definitions will correspond to the rule of evaluation chosen, e.g., evaluate from left to right. A rather similar thing occurs when the actual parameters of a **procedure** call are expressions which have side-effects. In this case we must also introduce subsidiary definitions before the call, ensuring that the values of any non-locals being fed to the **procedure** are updated by these definitions.

In this method of dealing with non-local variables we must know when we are translating a **procedure** call which variables are non-locals of the **procedure** and which of these are assignable. There is normally no difficulty about this in Algol, but it is not so easy for a **procedure** f which has another **procedure** g as a formal parameter. In this case the non-local variables of f will differ from call to call and must be taken to include the non-locals of g , similarly

for the assignable variables. This means that we can no longer write the translation for f in our previous manner, including the non-locals explicitly as extra parameters. We will return to this difficulty in the section 'Call by reference' (p. 17).

Parameters called by name present another problem. Landin (1965) has shown that they may be treated as functions with no parameters. By this means they can be handled in our scheme by the same method as **procedures** which are parameters of other **procedures**.

DATA STRUCTURES

We now consider the problems presented by assignment in a language which allows processing of data structures. We will take as an example an informal language essentially equivalent to LISP but with an Algol-like syntax. We will use as basic functions HEAD, TAIL and CONS, writing them in capitals to distinguish them from the related ISWIM functions *head*, *tail* and *cons*. The distinction is important, especially in the case of *cons*. The methods we shall use are general enough to extend to other languages which treat data structures.

We will call the language which we are using as an illustration 'List-Algol' and define it by example rather than give an explicit description. No particular virtue is claimed for this language, we are only concerned to find a formalism (other than a pictorial one) which makes the meaning of it clear.

<i>List-Algol</i>	<i>Explanation</i>
(i) begin list x, y ; $x := \text{CONS}(1, \text{nil})$; $y := \text{CONS}(2, x)$; $\text{HEAD}(y) := 3$; $\text{outinteger}(1, \text{HEAD}(y) + \text{HEAD}(x))$ end	x is a list of one item y is a list of two items The first item of y changes $3 + 1$ is printed
(ii) begin list x, y ; $x := \text{CONS}(1, \text{nil})$; $y := \text{CONS}(2, x)$; $\text{HEAD}(x) := 3$; $\text{outinteger}(1, \text{HEAD}(\text{TAIL}(y)))$ end	The first item of x changes. This also changes the second item of y , i.e., $\text{HEAD}(\text{TAIL}(y))$. 3 is printed
(iii) begin list x, y ; $x := \text{CONS}(1, \text{nil})$; $y := \text{CONS}(x, x)$; $\text{HEAD}(\text{TAIL}(y)) := 2$; $\text{outinteger}(1, \text{HEAD}(\text{HEAD}(y)))$ end	y is a list with two identical sub-lists This also changes to $\text{HEAD}(x)$ and $\text{HEAD}(\text{HEAD}(y))$ 2 is printed

BURSTALL

```
(iv) begin list x;
      x:=CONS(1, nil);
      TAIL(x):=x;           x is now a circular list
      outinteger(1, HEAD(TAIL(TAIL(x)))) 1 is printed
end
```

Of course it is tempting to simply translate CONS by *cons*, HEAD by *head* and TAIL by *tail*, but this will not do. The following examples show how this naive approach breaks down. To carry out the translations we will need a couple of auxiliary functions defined thus:

```
let updatehead(x, i)=cons(i, tail(x))
and updatetail(x, i)=cons(head(x), i);
```

List-Algol

```
(i) begin list x, y;
     x:=CONS(1, nil);
     y:=CONS(2, x);
     HEAD(y):=3;
     outinteger(1, HEAD(y)+HEAD(x))
end
```

ISWIM (naive translation)

```
print let x=undef and y=undef;
      let x=cons(1, nil);
      let y=cons(2, x);
      let y=updatehead(y, 3);
      result head(y)+head(x)
```

This translation is adequate. Both programs print '2+1'

```
(ii) begin list x, y;
      x:=CONS(1, nil);
      y:=CONS(2, x);
      HEAD(x):=3;
      outinteger(1, HEAD(TAIL(y)))
end
```

```
print let x=undef and y=undef;
      let x=cons(1, nil);
      let y=cons(2, x);
      let x=updatehead(x, 3);
      result head(tail(y))
```

This translation fails. The List-Algol prints '3' but the ISWIM prints '1' since it does not cater for the side-effect on y of the assignment to x.

```
(iii) begin list x, y;
       x:=CONS(1, nil);
       y:=CONS(x, x);
       HEAD(TAIL(y)):=2;
       outinteger(1, HEAD(HEAD(y)))
end
```

```
print let x=undef and y=undef;
      let x=cons(1, nil);
      let y=cons(x, x);
      let y=updatehead(tail(y), 2);
      result head(head(y))
```

This translation fails. The List-Algol prints '2' but the ISWIM prints '1', again because it does not cater for a side-effect.

```
(iv) begin list x;
      x:=CONS(1, nil);
      TAIL(x):=x;
      outinteger(1, HEAD(TAIL(TAIL(x))))
end
```

```
print let x=undef;
      let x=cons(1, nil);
      let x=updatetail(x, x);
      result head(tail(tail(x)))
```

The List-Algol prints '1', but the ISWIM result is undefined.

REPRESENTING DATA STRUCTURES BY GRAPHS

In our naive translations above we assumed that the correct translations of CONS, HEAD and TAIL were the ISWIM functions *cons*, *head* and *tail*. So long as we are translating a language analogous to pure (functional) LISP this is sufficient, but it is not difficult to see that once we allow assignment to components of data structures, e.g., '*TAIL(x) := y*', this translation is no longer adequate. The reason is that *cons*, *head* and *tail* are functions acting on trees, whilst the structures dealt with in a list processing language with assignment to components are much more complex objects than trees. In fact they are a certain kind of directed graph.

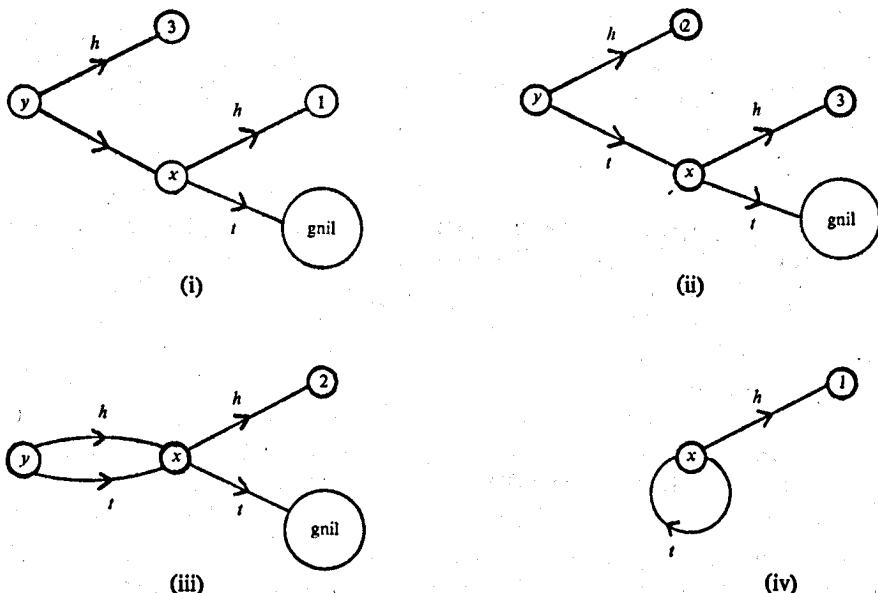


FIG. 1

Such a graph can be represented as a (potentially infinite) set of nodes N , a set of data objects (atoms) A , which might be character strings or integers, and a terminal object which we will call *gnil*, together with two functions h and t from N to $N \cup A \cup \{gnil\}$. We use the word 'item' for elements of $N \cup A \cup \{gnil\}$. The functions h and t are partial and defined only for a subset of the nodes N . Thus an arc in the graph corresponds to applying h or t to a node, and there are two kinds of arc corresponding to h and t respectively. Since N and A are fixed it will be sufficient to represent the graph by the pair $\langle h, t \rangle$ in our ISWIM versions of List-Algol programs. A list variable will have as its value a node in such a graph. The nodes correspond to addresses in a computer and the functions h and t find the contents of the address N .

Fig. 1 shows pictorially the structures produced in the four sample programs of the last section.

BURSTALL

We will need some functions to act on these graphs.

The function *gcons* takes two arguments which are any items, i.e., nodes, atoms or *gnil*, and also a graph as argument. It produces a new node and a new graph, i.e., an updated *h* and *t*.

```
let gcons(i1, i2, h, t) = [let n3 = anyfreenode      |
                           |let h(n) = n = n3 → i1, h(n); |
                           |let t(n) = n = n3 → i2, t(n); |
                           |result <n3, h, t>           ];
```

We write here 'let *n3* = *anyfreenode*' meaning any node of *N* for which *h* and *t* are not defined. This could be done more correctly by introducing a list of free nodes as an extra parameter of *gcons*. However, this would complicate the expressions without adding anything of real interest. We will simply assume in simplifying the ISWIM expressions that the new nodes introduced are all distinct.

There are two updating functions which produce just a new graph.

```
let gupdatehead(n1, i2, h, t) = [let h(n) = n = n1 → i2, h(n); |
                                |result <h, t>                  ]
and gupdatetail(n1, i2, h, t) = [let t(n) = n = n1 → i2, t(n); |
                                |result <h, t>                  ];
```

We also need functions for testing for atoms and *gnil*.

```
let gatom(x) = x ∈ A and gnull(x) = x = gnil;
```

We will use the functions *h₀* and *t₀* for the null versions of *h* and *t* which are undefined for all nodes.

```
let h0(n) = undef and t0(n) = undef;
```

We can now make a more satisfactory translation of our previous examples.

List-Algol

(i) begin list *x*, *y*;

```
x := CONS(1, nil);
y := CONS(2, x);
HEAD(y) := 3;
outinteger(1, HEAD(y) +
HEAD(x))
end
```

ISWIM

```
print let x = undef and y = undef
      and h = h0 and t = t0;
let <x, h, t> = gcons(1, gnil, h, t);
let <y, h, t> = gcons(2, x, h, t);
let <h, t> = gupdatehead(y, 3, h, t);
result h(y) + h(x)
```

Substituting the definitions of *gcons* and *gupdatehead* and then making the obvious simplifications we obtain successively the left-hand and right-hand versions below. (We change *n3* to *n4* in the second *gcons* to avoid clashes.)

```
let x = undef and y = undef
  and h = h0 and t = t0;
```

```
let <x, h, t> =
```

```
[let n3 = anyfreenode; |
 |let h(n) = n = n3 → 1, h(n); |
```

```
let n3 = anyfreenode;
let h(n) = n = n3 → 1, undef;
```

ABSTRACT FOUNDATIONS

<pre> [let $t(n) = n = n3 \rightarrow gnil, t(n);$ [result $\langle n3, h, t \rangle$]; let $\langle y, h, t \rangle =$ [let $n4 = anyfreenode;$ [let $h(n) = n = n4 \rightarrow 2, h(n);$ [let $t(n) = n = n4 \rightarrow x, t(n);$ [result $\langle n4, h, t \rangle$]; let $\langle h, t \rangle =$ [let $h(n) = n = y \rightarrow 3, h(n);$ [result $\langle h, t \rangle$]; result $h(y) + h(x)$ </pre>	<pre> let $t(n) = n = n3 \rightarrow gnil, undef;$ let $x = n3;$ let $n4 = anyfreenode;$ let $h(n) = n = n4 \rightarrow 2, h(n);$ let $t(n) = n = n4 \rightarrow x, t(n);$ let $y = n4;$ let $h(n) = n = y \rightarrow 3, h(n);$ result $h(y) + h(x)$ </pre>
--	---

Further substitutions give

```

let  $n3 = anyfreenode;$ 
let  $n4 = anyfreenode;$ 
let  $t(n) = n = n4 \rightarrow n3, n = n3 \rightarrow gnil, undef;$ 
let  $h(n) = n = n4 \rightarrow 3, n = n4 \rightarrow 2, n = n3 \rightarrow 1, undef;$ 
result  $h(n4) + h(n3)$ 

```

This has value $3+1$ (cf. Fig. 1(i)). Note that the second conditional clause in the definition of h is redundant, i.e., we would simplify to

```
let  $h(n) = n = n4 \rightarrow 3, n = n3 \rightarrow 1, undef$ 
```

Note that we could have made the same simplifications if the integers 1, 2 and 3 had been replaced by expressions involving variables or by formal parameters. In other words we have a means of simplifying programs without actually running them to evaluate the result.

We may now translate the second example which involves a side-effect.

List-Algol

```

(ii) begin list  $x, y;$ 

     $x := CONS(1, nil);$ 
     $y := CONS(2, x);$ 
     $HEAD(x) := 3;$ 
    outinteger(1, HEAD(TAIL(y)))
end

```

ISWIM

```

print let  $x = undef$  and  $y = undef$ 
    and  $h = h_0$  and  $t = t_0;$ 
    let  $\langle x, h, t \rangle = gcons(1, nil, h, t);$ 
    let  $\langle y, h, t \rangle = gcons(2, x, h, t);$ 
    let  $\langle h, t \rangle = gupdatehead(x, 3, h, t);$ 
    result  $h(t(y))$ 

```

This reduces eventually to

```

let  $n1 = anyfreenode$  and  $n2 = anyfreenode;$ 
let  $h(n) = n = n1 \rightarrow 3, n = n2 \rightarrow 2, undef$ 
and  $t(n) = n = n2 \rightarrow n1, n = n1 \rightarrow gnil, undef;$ 
result  $h(t(n2))$ 

```

This has value 3 as required (cf. Fig. 1(ii)).

<pre> (iii) begin list $x, y;$ $x := CONS(1, nil);$ </pre>	<pre> print let $x = undef$ and $y = undef$ and $h = h_0$ and $t = t_0;$ let $\langle x, h, t \rangle = gcons(1, gn timer, h, t);$ </pre>
---	--

BURSTALL

```

y:=CONS(x, x);          let <y, h, t>=gcons(x, x, h, t);
HEAD(TAIL(y)):=2;      let <h, t>=gupdatehead(t(y), 2, h, t);
outinteger(1, HEAD(HEAD(y))) result h(h(y))
end

```

This reduces after some substitutions to

```

let t(n)=n=n2→n1, n=n1→gnil, undef
and h(n)=n=n1→2, n=n2→n1, undef;
result h(h(n2))

```

This has value 2 as required.

```

(iv) begin list x;          print let x=undef
                             and h=h0 and t=t0;
                             let <x, h, t>=gcons(1, gnil, h, t);
                             let <h, t>=gupdatehead(x, x, h, t);
                             result (h(t(t(x))))
                             x:=CONS(1, nil);
                             TAIL(x):=x;
                             outinteger(1, HEAD(TAIL
                             (TAIL(x))))
end

```

On reduction

```

let n1=anyfreenode;
let h(n)=n=n1→1, undef;
let t(n)=n=n1→n1, n=n1→gnil, undef;
result h(t(t(n1)))

```

This has value 1 as required since $t(n1)=n1$.

Before leaving the subject of data structures we will say something about arrays in Algol. These can easily be handled by letting each array correspond to a sequence of functions.

Algol

```

begin integer array a[1:10];
a[3]:=1;
a[4]:=2+a[3];

```

ISWIM

```

let a(i)=undef;
let a(i)=i=3→1, a(i);
let a(i)=i=4→2+a(3), a(i);

```

We should really make provision for checking that the subscript is within the bounds. To do this we would need to carry these bounds along with the ISWIM function.

CALL BY REFERENCE

The method of handling data structures described above could be extended to languages in which more elaborate structures can be created. To do this we would introduce not just two functions h and t , but as many as are needed to correspond to the selector operations available in the language in question. What of the case where we introduce only one function, say c for 'contents' instead of h and t ? We can use this to explain call by 'reference' or by 'simple name'.

We will modify Algol again to include variables of type **reference**, calling

the resulting language 'Ref-Algol'. We give an example with a naive ISWIM translation, which will be seen to be inadequate.

<i>Ref-Algol</i>	<i>Naive ISWIM</i>
begin reference integer u, v ;	print let $u = \text{undef}$ and $v = \text{undef}$;
procedure $f(x)$;	let $f(x, u) = [\text{let } x = x + 1;$
reference integer x ;	let $x = x + u$;
begin $x := x + 1$;	result x];
$x := x + u$	
end;	
$u := 0; v := 0$;	let $u = 0$; let $v = 0$;
$f(v)$;	let $v = f(v, u)$;
$f(u)$;	let $u = f(u, u)$;
outinteger(1, $u + v$)	result $u + v$
end	

The Ref-Algol procedure increments the variable fed to it as a parameter and then adds u to it. The ISWIM translation fails if this variable is itself u , adding the unincremented value of u instead of the more recent one. We can get a correct translation using our data structure techniques and the function c . We will need functions $gconsref$ corresponding to $gcons$, and $gupdatecont$ corresponding to $gupdatehead$.

begin reference integer u, v ;	print let $u = \text{anyfreenode}$
	and $v = \text{anyfreenode}$ and $c = c_0$;
procedure $f(x)$;	let $f(x, u, c) =$
reference integer x ;	let $c = gupdatecont(x, c(x) + 1, c)$;
begin $x := x + 1$;	let $c = gupdatecont(x, c(x) + c(u), c)$;
$x := x + u$	result c];
end;	
$u := 0$;	let $c = gupdatecont(u, 0, c)$;
$v := 0$;	let $c = gupdatecont(v, 0, c)$;
$f(v)$;	let $c = f(v, u, c)$;
$f(u)$;	let $c = f(u, u, c)$;
outinteger(1, $u + v$)	result $c(u) + c(v)$
end	

This will reduce to

```

let  $u = \text{anyfreenode}$  and  $v = \text{anyfreenode}$  and  $c = c_0$ ;
let  $f(x, c) =$ 
  [let  $c(n) = n = x \rightarrow c(x) + 1, c(n)$ ; |
  |let  $c(n) = n = x \rightarrow c(x) + c(u), c(n)$ ;; |
  |result  $c$ ];
let  $c(n) = n = u \rightarrow 0, c(n)$ ;
let  $c(n) = n = v \rightarrow 0, c(n)$ ;
let  $c = f(v, u, c)$ ;

```

$$\text{let } f(x, c) = [\text{let } c(n) = n = x \rightarrow c(x) + 1 + [u = x \rightarrow c(x) + 1, c(u)], c(n); \text{result } c]$$
$$\text{let } c(n) = n = u \rightarrow c(u) + 1 + [u = u \rightarrow c(u) + 1, c(u)], c(n)$$

let $c(n) = n = u \rightarrow c(u) + 1 + c(u) + 1, c(n)$

The application of this method of translation to more elaborate cases should be fairly obvious. We have assumed that variables being used as **reference** variables are declared as such. Even if this were not the case it is only necessary to note which variables are ever used as an actual parameter correspond to a formal parameter called by **reference** and note such variables as **reference** variables.

The function c here plays a similar role to Strachey's store. Strachey (1965) defines an entity σ , the store, and two other kinds of entities, 'left-hand values', α (generalised addresses) and 'right-hand values', β (the contents of the addresses). The updating operator, U , and the contents operator, C , act on these.

$\beta = C(\alpha, \sigma)$ (rewriting Strachey's equation on p. 203 in a rather more conventional notation)

$$\sigma' = U(\alpha, \beta', \sigma)$$

For example $y:=2+x$ transforms σ into σ' where $\sigma' = U(\alpha_y, 2 + C(\alpha_x, \sigma), \sigma)$. Now it would appear that the axioms which completely characterise the operators C and U are:

- (i) For all α, β, σ
- $$C(\alpha, U(\alpha, \beta, \sigma)) = \beta$$

ABSTRACT FOUNDATIONS

- (ii) For all $\alpha, \alpha', \beta, \sigma$, if $\alpha \neq \alpha'$ then
 $C(\alpha, U(\alpha', \beta, \sigma)) = C(\alpha, \sigma)$.

These axioms are satisfied by the functions

$$C(n, c) = c(n)$$

$$U(n, i, c) = \lambda n'. n' = n \rightarrow i, c(n')$$

Where the n is equivalent to Strachey's α and the i to his β . Thus the function c , used as I have used it above, gives a particular model (a very simple one) which satisfies the axioms for Strachey's 'contents' and 'update' functions. The functions h and t used above would be a model for the obvious extension of Strachey's system to lists.

ACKNOWLEDGEMENTS

Much of this work was done while I was on a temporary assignment to the CPL project under the direction of Christopher Strachey. I am very grateful for many instructive and stimulating discussions with him and with Peter Landin. They are not responsible for mistakes or misapprehensions in this paper. My thanks are also due to Donald Michie for never-failing support and encouragement.

REFERENCES

- Church, A. (1941). The calculi of lambda-conversion. *Annals of Mathematical Studies*, No. 6. Princeton, N.J.: Princeton University Press.
- Curry, H. B., & Feys, R. (1958). *Combinatory Logic*, vol. 1. Amsterdam: North-Holland.
- Landin, P. J. (1964). The mechanical evaluation of expressions. *Computer J.*, 6, 308-320.
- Landin, P. J. (1965). A correspondence between ALGOL 60 and Church's Lambda-notation. *Communs Assn Comput. Mach.*, 8, 89-101, 158-165.
- Landin, P. J. (1966a). The next 100 programming languages. *Communs Assn Comput. Mach.*, 9, 157-166.
- Landin, P. J. (1966b). A formal description of ALGOL 60. *Formal Description Languages for Computer Programming*. T. B. Steel, Jr. (ed.). Amsterdam: North-Holland.
- McCarthy, J. et al. (1962). *LISP 1.5 Programmer's Manual*. Cambridge, Mass.: MIT.
- McCarthy, J. (1963). A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*. P. Braffort and D. Hirschberg (eds.), pp. 33-70. Amsterdam: North-Holland.
- Rosenbloom, P. (1950). *The Elements of Mathematical Logic*. New York: Dover.
- Strachey, C. (1965). Towards a formal semantics. *Formal Description Languages for Computer Programming*. T. B. Steel, Jr. (ed.). Amsterdam: North-Holland.

2

SOME TRANSFORMATIONS AND STANDARD FORMS OF GRAPHS, WITH APPLICATIONS TO COMPUTER PROGRAMS*

D. C. COOPER

CARNEGIE INSTITUTE OF TECHNOLOGY

INTRODUCTION: PROGRAMS AND DIRECTED GRAPHS

In Cooper (1967) I reported on several attempts at proving theorems about computer programs, with the ultimate goal in mind of providing mathematical proofs that programs are correct rather than just testing them on some particular data sets. In the last section I commented that the proofs I obtained in Cooper (1966a) could be viewed as deep results about small programs but that what was needed were techniques for proving shallow results about large programs. By this latter I meant that the proofs did not depend too much on properties of the basic functions or commands used in the program but rather depend on the synthesis of a large number of trivial properties, the particular way this synthesis is performed being guided by the structure of the large program. Some of the other work reported on in Cooper (1967), particularly the work of Evans (1965), has this flavour but are particular proofs about particular programs. Mention should also be made of the long proof of Balzer (1966) which proves that a particular finite automaton correctly performs its given task.

There are many programs whose control structure can be well shown by a directed graph—that is the flow chart of the program. The first goal of this research is to produce some kind of automatic scheme which will prove results about programs which only use conditional statements and assignment statements and we shall not consider those features of a programming

* This work was supported by the Advanced Research Projects Agency of the U.S. Department of Defense under the contract SD-146 to the Carnegie Institute of Technology.

language which imply some kind of control structure not immediately so representable; for example, the passing of parameters to sub-routines, perhaps by the use of procedures and functions previously defined by the programmer. These are very important features of programming languages, but they will not be further considered in this chapter. Although the techniques of this chapter may not be directly applicable in these circumstances, some modification of the ideas used could well be useful; for example, the graphs could represent the relations between a set of mutual recursive function definitions and the transformations to be defined later could well be meaningful.

Having associated a graph with a program, there may well be transformations which can be made on the graph which 'do not affect' the program, later we shall amplify this remark. The question then arises as to whether the graph may be put into some standard form by a series of such transformations, this standard form being one in which it is easier to prove results about the program by taking advantage of special features of the standard form. The purpose of this chapter is to define several such transformations and two such standard forms, and to give results and conjectures about the possibility of transforming programs to the standard forms.

The particular way in which a directed graph is associated with a given program is not important. All that matters is that the graph transformations defined in the next section should represent meaningful transformations if performed on the programs with which the graphs are associated. However, in order to better illustrate the intention behind the transformations, we give a particular definition of a program and of the graph associated with a program. This definition will use a state vector approach (see McCarthy 1960). In this approach the current state of a computation is represented by the value of a vector, each component of which corresponds to one of the variables, machine locations, etc. (dependent on the particular programming language used). The effect of a basic command can then be described by a function whose argument is a state vector (the state of the machine before the command is obeyed) and whose value is also a state vector (the state of the machine after the command is obeyed). A sequence of commands corresponds to composition of the corresponding functions. A two-way test is represented by a predicate whose argument is a state vector.

Assume then that we have a domain D , a set of functions f_i which map D into D and a set of predicates p_i whose domains are D . A *program* is defined to be a directed graph with every arc of which is associated one of the f functions and one of the p predicates. It is a *legal program* if the graph has just one node (A , say) with no arcs leading to it, if it has just one node (Z , say) with no arcs leading from it, if every node is on a path which starts at A and ends at Z , and if for all nodes (except Z) the set of predicates on all arcs leading from the node is complete and mutually exclusive.

The intended interpretation is that if f_i and p_j are the function and predicate associated with some arc then f_i represents the effect of all the statements obeyed along that arc and p_j represents the condition that the arc is entered.

A program then defines a partial function from D into D in an obvious manner (partial because the program may loop indefinitely).

It should again be emphasised that we are not concerned with this particular definition of a program, only with the ability to associate a graph with a program. As alternatives we could have associated the information in the program with the nodes rather than the arcs, or we could have taken some simple programming language and explicitly stated how to obtain the graph from the program.

The results to be obtained later could well have applications in other areas than computer programs. For example, Seshu and Reed (1961) describe the application of graphs to sequential machines and to systems of linear algebraic equations. All our transformations can be given interpretations in these areas, in fact the delete node transformation (T 5 of the next section) is already well known in these applications. The particular field of interest determines the transformations of interest; for example, later we shall reject a certain transformation (DT 1) as not being a useful one for computer programs. However, this transformation is of use in the areas of sequential machines and of linear equations.

EXAMPLES OF TRANSFORMATIONS

In this section we define several directed graph transformations of interest. In each case we give a description of the general case in English and illustrate the transformation with a particular case. The illustration of course only shows how the affected part of the graph is altered, the graph may have other nodes and arcs except that in all cases we assume that there are no further arcs which lead into or out of the nodes labelled N .

T 1. Stretch a node N by outputs.

Let O be some subset of the output nodes of N , this subset may be all the output nodes but must not be empty. Delete all arcs from N to nodes in O , add a new node N_1 and add arcs from N to N_1 and from N_1 to all nodes in O . (The output nodes of a node N are all those nodes P such that there is an arc NP , including N itself if there is an arc NN .)

In the example O is the set $\{N, D, E\}$.

T 2. Stretch a node N by inputs.

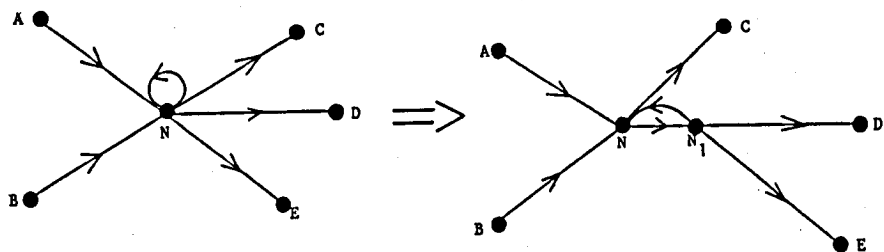
Let I be some subset of the input nodes of N ; this subset may be all the input nodes but must not be empty. Delete all arcs from nodes in I to N , add a new node N_1 and add arcs from N_1 to N and from all nodes in I to N_1 . (The input nodes of a node are defined in an analogous way to the output nodes.)

In the example I is the set $\{A, B\}$.

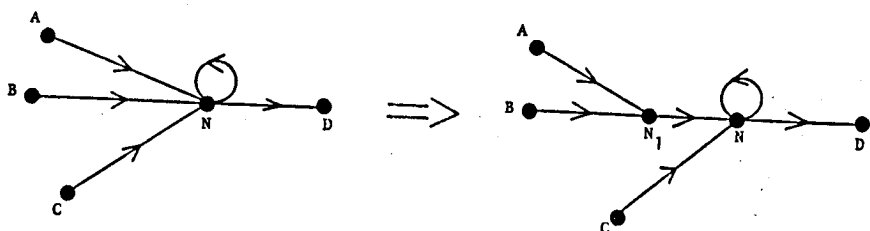
T 3. Duplicate a node N .

Let I be some non-empty, proper subset of the input nodes of N , not including N itself if NN is an arc. Delete all arcs from nodes in I to N , add a new node N_1 , add arcs from N_1 to all the output nodes of N (except N itself if NN is an arc) and add arcs from every node in I to N_1 . If NN is an arc also add an arc $N_1 N_1$.

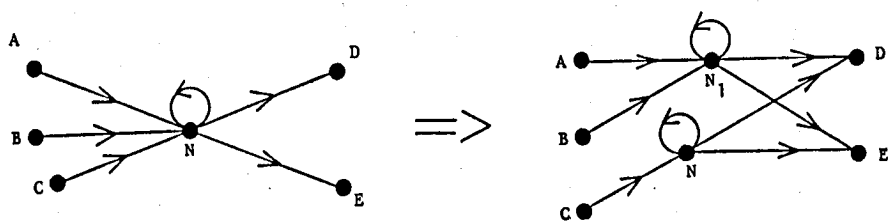
ABSTRACT FOUNDATIONS



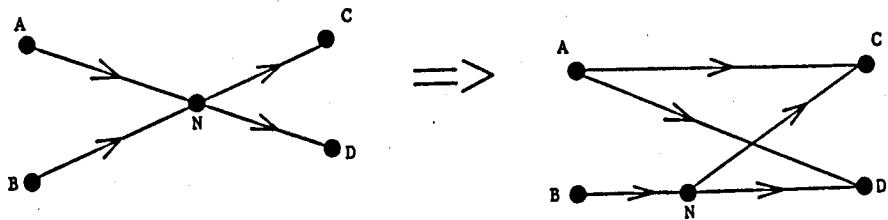
T1



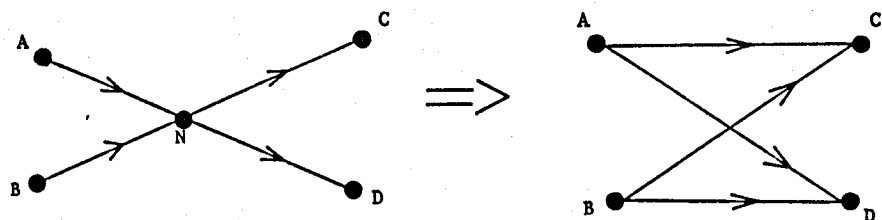
T2



T3



T4



T5

Transformations

In the example I is the set $\{A, B\}$.

T 4. Delete an arc AN.

This transformation may only be made if there is no arc NN. Delete the arc AN and add arcs from A to all the output nodes of N. If A was the only input node of N then also delete node N and all arcs leading from N.

T 5. Delete a node N.

This transformation may only be made if there is no arc NN and is equivalent to applying a sequence of T 4 transformations in any order to all the arcs leading to N. Add all possible arcs from an input node of N to an output node of N and delete node N and all arcs leading to or from N.

In all these cases if the untransformed graph is the graph of some program P it is easy to find an equivalent program with the transformed graph as its associated graph. For example, in the diagram illustrating transformation T 1 if $p_{AN}(s)$ and $f_{AN}(s)$ are the predicate and function associated with arc AN on the untransformed graph (and similarly for the other arcs) then in the transformed graph on arc NN_1 we have the predicate $p_{ND}(s) \vee p_{NE}(s) \vee p_{NN}(s)$ and the identity function, N_1N , N_1D and N_1E have the same predicates and functions as NN, ND and NE respectively, and all other arcs have the same predicates and functions as before the transformation. Transformation T 4 represents the effect of anticipating the test at N at the previous node A so that on the transformed graph, for example, with the arc AC we associate the predicate $p_{AN}(s) \wedge p_{NC}(f_{AN}(s))$ and the function $f_{NC}(f_{AN}(s))$.

As two examples of transformation which we do not wish to allow consider the following:

DT 1. Delete a node N with a loop.

This is defined exactly as T 5 but we remove the restriction that there must be no arc NN.

DT 2. Join two arcs AB and CD.

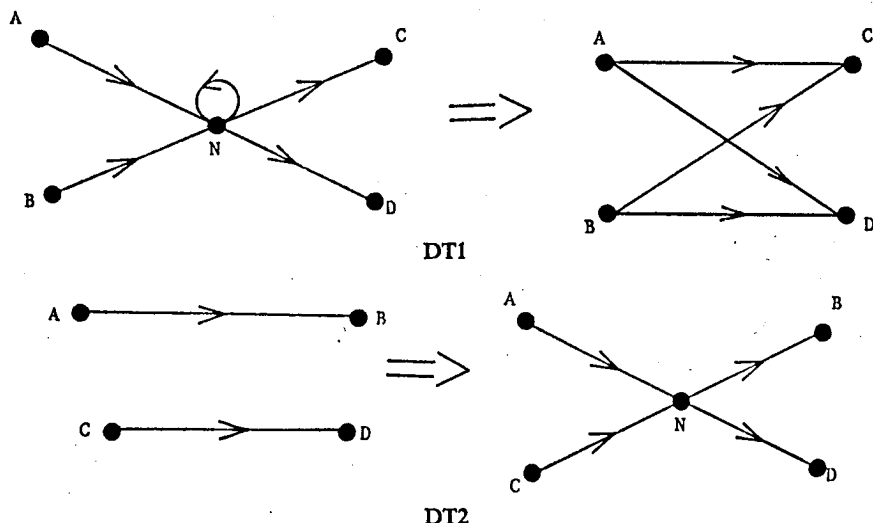
Delete the arcs AB and CD, add a new node N and add arcs AN, NB, CN and ND.

In DT 1 with arc AC we would have to associate a predicate expressing the condition that when control finally comes out of the loop at N control passes to node C. This can be done, but in general requires an infinite disjunction, we assume this is not a useful concept to introduce. The aim in making the transformation is to better reveal the program's structure, we do not wish to produce a transformed program which introduces complex new concepts and makes our task of proving the desired theorem about the program much more difficult. Admittedly we are being vague here, the transformations T 1 to T 5 imply 'we do not mind complicating our programs by forming conjunctions and disjunctions of predicates and compositions of functions'. Note that the situation arising in DT 1 would not be improved by adding loops on the transformed graph at A and B, or at new nodes internal to the arcs AC, AD, BC and BD.

We could allow the transformation DT 2 if an extra Boolean variable is

defined and semantics added to the program to set this variable true along arc AN, false along arc CN. The predicate on arc NB would then be a test whether this variable is true, on arc ND whether it is false. In effect we are coding the path already taken on the graph into a variable. Again we assume that this is not a good thing to do.

In this chapter we define T 1 to T 5 as the only allowable transformations, and then prove or conjecture theorems concerning them. We would like to somehow characterise the allowable transformations and perhaps produce



some more. A possible approach is to define a transformation of one graph into another by a definition of the following form:

$G \rightarrow G'$ if and only if $(p)(\exists p')[G = \text{graph}(p) \supset \{G' = \text{graph}(p') \wedge f(p, p')\}]$.

This definition presupposes that we already have a definition of a program, of a function 'graph' which produces the graph of a program and of a predicate $f(p, p')$. This predicate must express the condition that program p' is equivalent to program p , and moreover can be produced from it with some given rules which allow such things as function composition but does not allow anything corresponding to the infinite disjunction of DT 1 or the new Boolean variable of DT 2.

PROPER CYCLE FREE FORM

Having defined the allowable graph transformations, are there some particularly desirable standard forms into which we can try to transform a given graph? Two desirable forms will be defined, one in this section and one in the following section.

Most, if not all, of the questions we wish to answer about a computer program are easy to answer if there are no cycles in the program. The simpler the interconnections between the cycles the easier it should be to

answer questions about whether a program performs correctly; it should therefore be profitable to make transformations aimed at simplifying the cycle structure of a graph.

The simplest (and also rarest) program is one which has no cycles. Next one might consider programs in which all the cycles are independent, i.e., no section of the program is in more than one cycle. This class of programs is more interesting but still very restrictive. However, experience indicates that a large number of interesting programs may be transformed by a series of transformations into a program with this independent loop structure.

In order to make these vague remarks more exact we define a *proper cycle* of a graph to be a cycle through two or more nodes and a *proper cycle free* (PCF) graph to be a graph with no proper cycles. A graph which is PCF clearly has the independent looping property, and it is also trivial to transform any graph with independent loops into PCF form. We therefore take PCF form as being a standard form into which we try to transform a given graph, and obtain necessary and sufficient conditions for a graph to be transformable into PCF form together with an algorithm for making the transformation if it is possible. For this purpose only transformation T 5, delete a node, turns out to be relevant. If a graph can be transformed to PCF form by a sequence of transformations, each of which is one of T 1 to T 5, then this may be done by a sequence of T 5 transformations alone.

Consider graph 1. By deleting node A we obtain graph 2 which is not PCF (and which in fact cannot be transformed to PCF form at all; see the theorem below). But if instead of deleting A we delete B and then C we obtain graph 3 which is PCF and 'obviously much simpler' than graph 2. (The 2's on graph 3 indicate multiple arcs.) Graph 3 reveals the 'basic structure' of graph 1 in a way which graph 2 does not. These graphs show that the order in which nodes are deleted is critical.

Define a graph to be *loop connected* if it contains two nodes, A and B say, and three cycles α , β and γ such that: (i) there is no one node which is on α , β and γ , (ii) A is on α but not on β , (iii) B is on β but not on α , and (iv) both A and B are on γ . This is a generalisation of part of graph 2 in which arcs have been replaced by paths. Then we have:

Theorem 1. A graph G can be transformed by a series of transformations, each of which is one of T 1 to T 5, to a graph which is PCF if and only if it is not loop connected.

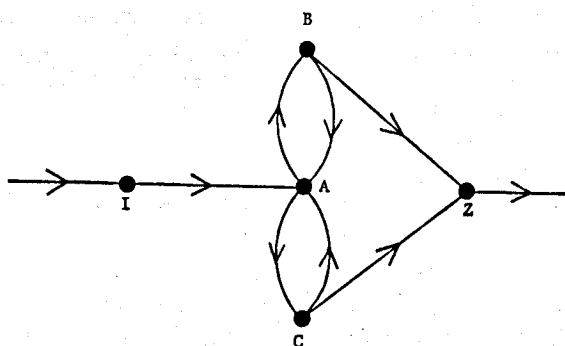
In Cooper (1966b) only transformation T 5 is considered, but it is easy to extend the results of that paper in such a way that Theorem 1 is an immediate deduction from the theorems proved there.

Theorem 1 does not give an algorithm for performing the sequence of transformations. In order to do this define a delete node N transformation (T 5) to be a *safe deletion* if there is some node A (not N) of the untransformed graph with the property that all cycles of the graph which pass

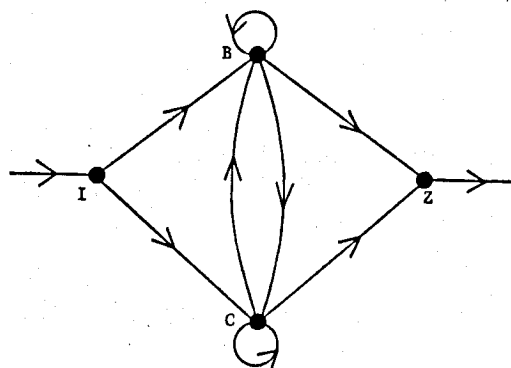
ABSTRACT FOUNDATIONS

through node N also pass through node A. For example, in graph 1 B or C may be safely deleted but not A. Then the following theorem gives the required algorithm; it is an easy extension of one of the theorems in Cooper (1966b):

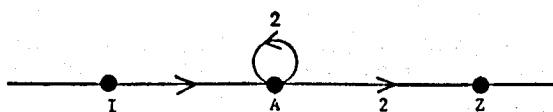
Theorem 2. Start with any graph G and perform safe deletions in any order until a graph G' is obtained on which no safe deletions may be



GRAPH 1



GRAPH 2



GRAPH 3

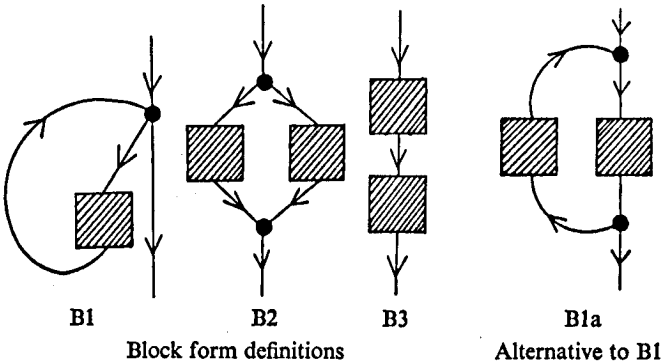
performed. This process must terminate and either G' is PCF or there is no sequence of transformations, each of which is one of T 1 to T 5, which will transform G to a graph in PCF form.

Theorems 1 and 2 completely solve the relevant questions about transformations to PCF form. In the same paper an equivalent condition to loop connectedness is defined; this condition appears more complicated but

simplifies the proofs. In addition a theorem is proved which shows that safe deletions is the widest possible subclass of deletions, i.e., if we make a deletion which is not a safe deletion we obtain a graph which is loop connected and hence not transformable to PCF form.

BLOCK FORM

The second standard form corresponds in a more natural manner than PCF form to the way a large number of programs are written; it roughly corresponds to programs in which all loops are properly nested. A recursive definition of block form is simply given: a graph is defined to be in *block form* if it is of one of the forms B1, B2 or B3 (see the diagrams) where the shaded boxes are either arcs or sub-graphs in block form.



It is not necessary to take exactly these forms; for example, instead of B2 we may allow any number of blocks in parallel and instead of B3 any number in series, obviously a graph of such form can easily be transformed into the standard block form. A more useful equivalent standard form is where instead of B2 and B3 we allow any graph made up of sub-graphs, each in block form, and these sub-graphs joined in any manner that does not create a cycle. Such a graph can be further transformed into the standard block form, but it may well be better not to carry out these further transformations as this usually involves a lot of duplicating of nodes. A third alternative is to replace B1 by B1a.

Very little progress has been made so far with proving results about transformation of graphs to block form, although a large number of small graphs have been investigated. A conjecture has been made as to the necessary and sufficient conditions for a graph to be transformable to block form and we hope soon to have an algorithm to effect the transformations. Any graph in PCF form can be transformed to block form; essentially it already is in block form according to the second alternative definition. However, consider graph 4, which is clearly loop connected and hence cannot be transformed into PCF form. First use T 1 (with N being A and O the set $\{B, Z\}$) and then use T 3 (N being A again and I the set $\{B\}$), this produces graph 5

and it is obvious how to put this into block form. Thus graph 4 is an example of a graph which can be transformed to block form but not to PCF form. As a more complex example, graph 6 may be transformed to block form; this is left as an exercise for the reader; a good start is to delete node A.

What graphs cannot be transformed to block form? Here we have no proven results but it seems almost certain that graph 7 cannot be so transformed (or any graph containing this as a sub-graph). The situation is much more complex than for the PCF case; there a knowledge of just the cycles of a graph was sufficient to determine whether the graph could be transformed to PCF form, but graphs 4 and 7 have the same cycles and one can be transformed to block form but not the other. It seems that a generalisation of graph 7 in which its arcs are replaced by paths (possibly intersecting each other) is the key situation which, if it occurs in a graph, makes the graph untransformable to block form. But examples show that if enough of the paths intersect then we are back to the position of being able to transform to block form.

By induction from examples tried we make the following definition:

A graph is *loop connected with two exits* if it contains three nodes, A, B and Z say, a cycle α through A, a cycle β through B, a path γ connecting A to B, a path δ connecting B to A, a path ϕ connecting A to Z and a path ψ connecting B to Z (see graph 8). Furthermore, there must be no node which is on all three members of any of the following triples:

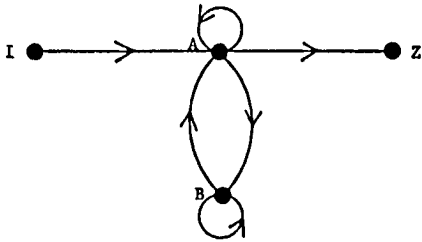
(α, β, γ) , (α, β, δ) , (δ, α, ψ) , (γ, β, ϕ) , (ϕ, ψ, α) and (ϕ, ψ, β) .

We conjecture the following:

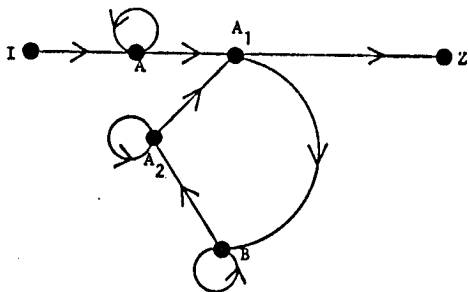
Theorem 3. A graph can be transformed by a series of transformations T 1 to T 5 to a graph which is in block form if and only if it is not loop connected with two exits.*

Böhm and Jacopini (1966) investigate the reduction of flow diagrams to standard forms, they show that corresponding to any flow diagram there is an equivalent one which is 'decomposable into Π , Φ and Δ '. This is precisely block form. Their algorithm for producing this equivalent flow diagram may be looked on as a sequence of graph transformations which, however, allow the transformation DT 2, which we have not allowed. Transformation DT 2 on programs was justified by the introduction of a new variable; rather than introducing a separate new variable every time DT 2 is used, Böhm and Jacopini introduce a single Boolean stack and add semantics to test the top of the stack, to delete the top of the stack and to add a new truth value to the top of the stack. This stack may be thought of as a code for the path through the flow diagram traversed so far. For our purposes we do not wish to allow this transformation, but it is an interesting result that if DT 2 is allowed then any graph may be transformed to block form.

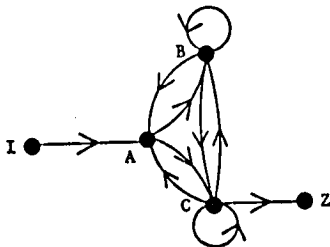
* The theorem is false as stated. It can be corrected by restricting Z to be a node which is not on any cycle. (Added in proof.)



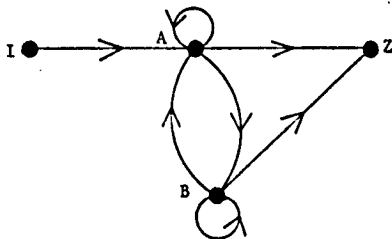
GRAPH 4



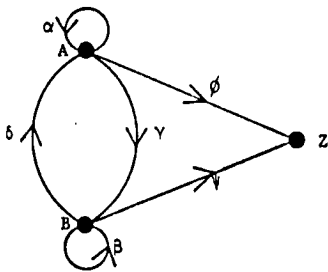
GRAPH 5



GRAPH 6



GRAPH 7



GRAPH 8

REFERENCES

- Balzer, R. M. (1966). *Studies concerning minimal time solutions to the firing squad synchronization problem*, Ph.D. thesis. Pittsburgh: Carnegie Institute of Technology.
- Böhm, C., & Jacopini, G. (1966). Flow diagrams, Turing machines and languages with only two formation rules. *Communs Assn Comput. Mach.*, 9, 366-371.
- Cooper, D. C. (1966a). The equivalence of certain computations. *Computer J.*, 9, 45-52.
- Cooper, D. C. (1966b). Reduction of programs to a standard form by graph transformation. Presented at 'International Seminar on Graph Theory and its Applications', Rome, Italy, July 5-9, 1966.
- Cooper, D. C. (1967). Mathematical proofs about computer programs. *Machine Intelligence 1*, N. L. Collins and D. Michie (eds.), pp. 17-28. Edinburgh: Oliver and Boyd.
- Evans, A. (1965). *Syntax analysis by a production language*, Ph.D. thesis. Pittsburgh: Carnegie Institute of Technology.
- McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine. *Communs Assn Comput. Mach.*, 3, 184-195.
- Seshu, S., & Reed, M. B. (1961). *Linear Graphs and Electrical Networks*. Massachusetts and London: Addison-Wesley Publishing Co., Inc.

3

DATA REPRESENTATION—THE KEY TO CONCEPTUALISATION

D. B. VIGOR

COMPUTING DEPARTMENT
UNIVERSITY OF GLASGOW

Conceptualisation is the formation and storage of concepts (Hunt 1962). A concept is a general rule for action or a general description of a set of attributes of objects. It can be looked at as the representation of a criterion for set membership whether the set be a set of descriptions or a set of actions. The formation of such descriptions or rules requires a notation in which the relations and modes of action can be represented. This notation should be general enough not to limit too strictly the nature of the concepts that can be described using it.

In this paper I would like to propose a model on which to base such a notation. I hope to indicate how this model can be used to set up a method of programming in which the programmer does not have to know a large amount about the system he is using, and can use primitive concepts much more related to conventional mathematics and to his problem, yet still generate 'efficient' programs.

ALGOL and FORTRAN have made large steps in giving a formal notation for numerical programming, but the primitives used are blunt and clumsy in describing actual computations on natural numbers. The programmer has to know concepts like '*real*', '*integer*', and to redundantly declare the nature of his results although these are 'obvious' from context. All these are irrelevant to the nature of the problem he is solving and are consequences of the fact that we use 'real machines' to carry out our computation.

In 'non-numerical' computing with our present notations we are even more bound by the nature of the machines and systems we use, and a large amount of ingenuity and skill is wasted on fitting problems into moulds which do not

suit them. This leads to frustration on the part of the problem solver, and inefficiency inside the machine.

The main items we wish to handle in computing processes are 'representation of objects' to which we can assign various properties or 'attributes' such as value, structure, permanence, and relationship with other 'objects'. Before any object, defined or undefined, can be considered to exist in a computer we must assign to it a 'name'. The 'name' is the primary attribute of 'existence'. This 'name' need not be specifically stated and can be implied by the method of formation of the object's representation. As 'nameless' objects are not members of our universe of discourse we can use the concept of 'name' to represent the 'object'. We will thus use the two words 'name' and 'object' synonymously because there is an isomorphism between them which is unresolvable.

One or more of the properties or attributes of a name inside a real or virtual computer can be looked at as a 'value' of the name. The 'value' of a name will depend upon the nature of the operation being carried out on that name. The method of representing in the computer a 'value' may or may not be the same as that used for representing 'names'.

A mathematical model of the above situation can be found in the theory of coloured graphs (Scoins 1967). A coloured graph is a set of 'named' points (modelling objects) which are connected by links (modelling relations) which are also 'named'. If we extend the definition of relation to allow for more than 2-argument relations, there is a duality between points and relations which allows us to look at the 'graph' as a set of relations which are linked by the entities over which they hold. A monochromatic sub-graph is a set of points having the same relationship. It is possible conceptually and practically to separate out from a coloured graph a monochromatic sub-graph.

There is one completely general way of representing such 'entities' as coloured graphs inside computers which has been exploited in the programming language LISP. There is a theorem of graph theory (Scoins 1967) which states that any general graph can be represented as a binary ordered directed graph. Directedness can be overcome by introducing dummy 'back' pointers; orderedness is an extra property which need not be considered; and any branching can be represented by two-way branching by introducing 'dummy points'. There are however other representations which are more 'efficient' in particular problem situations.

We can represent the operation of a machine in a static form. This can be done in a number of different ways; (a) sequential forms such as programs in ALGOL, FORTRAN, machine code and Ianov schemata (Rutledge 1964); and (b) nonsequential forms such as functional expressions, λ expressions (Landin 1964, 1965, 1966) and conditional forms (McCarthy *et al.* 1962). If one accepts a notation for program texts (R. Burstall, pp. 3-20 of this volume) based on Church's lambda-calculus (Church 1941), then one notes that there are basically three types of functions required in static program specification. These are Constructors, Selectors and Predicates.

VIGOR

1. *Constructors* are operations or functions which allow for the construction of compound or complex objects out of simpler ones, e.g., *cons* in LISP, *concatenate* in string processing languages and in Backus Normal Form, *merging* in commercial languages, $+$ and $\times$ in arithmetic languages.
2. *Selectors* are functions which find sub-objects from more complex ones, e.g., *car* and *cdr* in LISP, any parsing algorithm, any name in a programming language, $A(i, j)$ in ALGOL arithmetic expressions, 'the verb of this sentence' in English, 'if p then . . . else . . . ' in ALGOL, and ' λ ;' in LISP.
3. *Predicates* are functions which map classes of objects on to the truth values **true** or **false**. They can in certain systems, discussed later, be completely handled by special selectors, e.g., *eq*(x, y), *null*(x) in LISP, 'A is a bird' in English (assuming we have a test for birdness).

If we want to talk about evaluation or change resulting from use of functions with specific arguments, it is convenient to distinguish three other types of functions.

4. *Designators* are functions which map 'statements' in one representation onto corresponding 'statements' in another representation. These statements can be either
 - (i) descriptions of the allowable states of an object, or
 - (ii) descriptions of allowable transformations on objects, i.e. functions.

Designators may have 'side effects':

- (i) when mapping functions from one representation to another we may restrict, in some way, the class of arguments which can be used with the function;
- (ii) when mapping an object, we may restrict the class of functions which can accept this object as an argument. In these cases we say that the designators changes the domain of validity of the function or object which it is mapping. Some near approaches to designators in existing languages are:

integer x , real array $A[1:n]$ in ALGOL, *NIL* in LISP, **true** and **false**, *atom*.

5. *Applicators* are functions which specify that a described action, which is the argument of the applicator, has to be carried out within a given designational system, e.g., *EVAL* . . . in LISP. In most languages the applicator is implied by the program form or is specified externally by pressing a button on a hardware computer. In ALGOL an applicator is *comment* . . . ; or in the general purpose macro-generator (Strachey 1965) the macro protection brackets are a type of applicator. In PL1 (I.B.M. Corp. 1965) the operation *compile* is an applicator.
6. *Metafunctions*. These are specific operations which must be loaded on to a computer in order for a 'system' to operate. In LISP these are the constructor *cons*, the selectors *car*, *cdr*, and *cond*, the predicate

$eq(p, q)$, and the designators *true*, *false*, *nil*, *atom* (and their implied applicator *eval*).

I would like to look at the types of designators which we may require in the most general programming situations. In order to do this I need to define a few terms more accurately:

1. A *data structure* is a partially ordered set of items of information, e.g., a character string, a matrix, a tree, an integer, a function in a given representation.
2. A *function* is a mapping of one class of objects onto or into another.
3. The *domain of validity* of a function is the set of ordered pairs of objects for which the function is valid.

$$\text{domain } Z = \{z = (x, y) \mid x \in X \text{ and } y \in Y \text{ and } y = f(x)\}$$

where X is called the input domain, Y is called the range.

4. A *machine feature* is a particular set of hardware or software characteristics of a real or virtual computer, e.g., an adder, a register, a bank of core store, a magnetic tape, a nesting (FILO) store, a buffer (FIFO store).

A machine feature representing a function could be a sequence of changes of state in a particular piece of hardware. The concept of a machine feature involves the idea of predetermined behaviour when given a specific cue or nudge, and corresponds to initiating the evaluation of a function using a particular piece of hardware. The evaluation corresponds to the 'action' of the machine feature.

The orders representing a subroutine when evaluated by a machine initiate changes in state of particular components in this machine. Each one of these components has fixed rules for changing its state, as well as a fixed set of states. The mechanism of initiating a particular sequence of changes, and the components in which these changes occur, are the machine feature corresponding to this particular mechanism.

If we wish to represent an object which is a state or set of states, then the machine feature corresponding to this object is the assemblage of the states, to be set up in specific components of the machine in order to represent this state or set of states. An account of a language for describing objects resembling machine features and their actions has been published recently (Smith 1966).

A designator is a function from data structures on to machine features. I would like to consider the nature of both the function and the domain of validity in certain commonly used designational systems; a designational system is the designator plus a specification of its domain of validity. In non-ambiguous, time-independent computing processes, designators are one-one and onto functions and hence the input domain and the range are of the same cardinality. Thus we can conceive of inverse designators. The idea of non-one-one designators is interesting in dealing with real time

processes, non-deterministic automata and self-adaptive systems, but this is beyond the scope of this paper.

An applicator is a function from the designator of a functional representation onto the 'action' of a machine feature. The 'action' of a machine feature can be formally considered to be the mapping described by applying the 'inverse' of this designator function to the machine feature. It is a selector which selects a data structure corresponding to the particular machine feature. The apparent circularity of this definition is broken by having certain meta-functions which can be looked at either as metadesignators or meta-applicators which carry out the 'actions' required.

In other words, the action of a machine feature is determined by the structure of the machine feature, which can be represented as a formal data structure. We can distinguish some of the following machine features in most machines:

bit, byte, word, vector of words (segments), . . .

and the following data structures are representative of the objects we may wish to describe 'outside' the machine:

booleans, characters, arithmetic quantities (e.g., integer and real), vectors, arrays and trees, general linear graphs, thesauri.

We can relate machine features to one another by describing the constructors which assemble complex ones from simpler ones, and the selectors which allow their subdivision into the basic modules. We can also represent a formally corresponding set of constructors and selectors for the items outside the machine. Now we can see that a designator system can be represented in two formally equivalent ways. One is as a mapping of a set of functions onto another set of functions (this is the compiler approach to the problem), and the other is as a set of data items onto a set of actions of machine features (this is the interpretive approach to the problem).

Let us now look at some of the systems of mapping which have been developed and used till now. In order to simplify this survey, I shall presuppose a result I would really like to demonstrate: that one can satisfactorily classify these systems according to a set of criteria and that my particular choice of hierarchy between these criteria leads to a practically useful model of a system. I distinguish basically three main classes of representation of data which I call the *extended form*, the *mapped form*, and the *coded form* of information. I make the bold statement that the classification of any item into a given representation system is fairly unequivocal.

This classification depends on the assumption that at any level we are considering, data has a structural part (a set of interrelations between items) and a contents part (a group of properties that are purely local to any item). The contents part of a bit is its value 1 or 0, the structure is 'unity'. At this level the contents of an 'integer' could be a 'vector' of bit values in a binary machine, and the structure defined by three parameters: the length of the

integer in bits, the order in which the bits are to be taken, and the class of operators allowed to operate on this 'item'.

In the *extended form*, both the 'structure' and 'contents' are stored within one machine feature. Examples of this are:

1. LISP (McCarthy *et al.* 1962), where the unit is a pure 2-catena or a node in a binary oriented (ordered) graph; the single machine feature is a storage unit (considered to be the top of the stack) which is a function with two arguments, the 'symbol' and the 'link'.
2. IPL5 (Newell *et al.* 1961), which has two 2-argument machine features, one of which accepts a function and a symbol, and the other a link. This can be looked at as a 2-catena with internal structuring.
3. CORAL (Roberts 1965), which is a ring structure language which has a 3-argument machine feature, and a 3-catena as its basic item—one 'symbol' and two 'links' (a forward and a backward link).

In the *mapped form* the 'structure' and the 'contents' are mapped on to separate 'machine features' and could be independently manipulated. Either or both of the structure and the contents can be functional. Examples are:

1. The storage of arrays in 'Kingsgrove ALGOL' (English Electric Leo Marconi 1965), where one has a 'dope vector' which specifies the structure of the array in a form acceptable to one or more selector or 'indexing' functions, and the array contents are stored as a linear, sequential vector in core store (Fig. 1).
2. The 'Multics' (Dennis 1965) file system in which files are stored as segments with heading information on the structure of the file, and segments containing the actual information being stored.
3. A bit map of a 'sparse matrix' in which, associated with every item in the matrix, is a bit which is zero if the element is zero, unity if the element is not, and all the non-zero elements are stored elsewhere as a linear vector of words.

In the *coded form* the structure and the contents are again separated, but either one of these may be rearranged in some way by using some 'artifact' (in the biological sense, i.e., a side-effect) of the representation to handle the information more effectively in a given situation. Examples are:

1. Word condensation in a 'sparse matrix': The matrix contains a large number of sequential zeros when represented in a mapped form like (1) above. We can replace each string of zeros by a single zero followed by the number of such occurrences. The zero can be looked at as a function call using the parameter following it to generate the data concerned.
2. Any syntax-directed programming language has added 'redundancy' which checks the validity of messages transmitted in the language. The constructors for the language must be able to put in this redundancy, a selector or parsing algorithm removes it and checks acceptability of strings.

VIGOR

In any sequence of operations to be carried out using specific machine features there are certain requirements on invariance of domain (i.e., overflow, type clash) which can be built into a system. It would be, however, more natural to allow feasible calculations to be carried out within the specifications given externally without making any assumption about internal representation.

The most common feature of this nature is automatic multilength arithmetic processes where the mathematician *may* not want to be bothered with

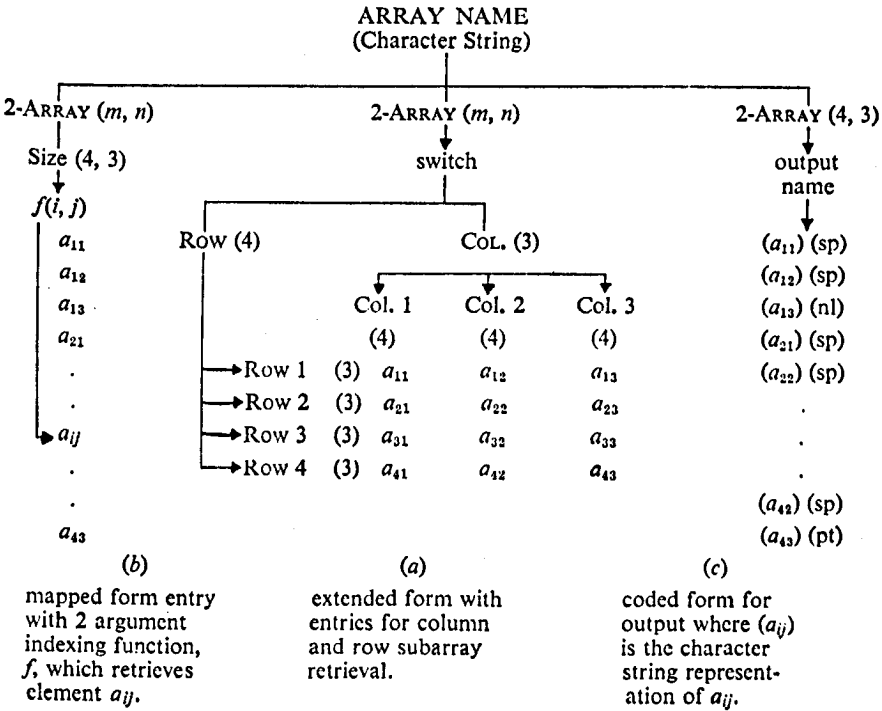


Fig. 1. Example of Array Storage in DDPS ($m=4, n=3$)

Note: (sp)=space, (nl)=new line, (pt)=page throw.

In all cases the 2-array is treated as a function.

absolute size considerations. Operations on mixed items which are externally compatible, e.g., integer and complex double length integer gives complex double length integer and to do this should require no specific declaration.

It is also possible for machine features to have invariant properties, e.g., an indexing register has certain strict requirements on its domain of 'arguments'. The aim of the systems programmer should be to map the invariants of objects outside machines into invariants of the machine features used in implementing them.

The representational system can be changed during the course of operation of the program—that is, we may choose different 'invariant' properties and

relations in order to apply a new class of functions to the same data. The 'decision' to do this can be left to a program which I shall call a 'system preprocessor', which, during the course of preprocessing a program, 'decides' on the most efficient representation of the problem in this particular machine (where a machine is a set of machine features with specific properties). The decision of efficiency can be made by using the following criteria:

1. 'Best fit' between 'machine feature' invariants and 'data structure' invariants.
2. The 'domain' of the 'data structures' maps onto the 'domain' of 'machine features' with the least 'domain overspill', i.e., with the maximum use of machine feature properties.
3. The 'information loss ratio' (I_l) in the system should be maximal.

$$I_l = t_R / t_M$$

where t_R = time taken on 'information reduction' or processing operations;

t_M = time taken on manipulative orders.

Thus the system should be particularised as early as possible in its 'life' inside the computer.

4. Minimal requirement of extra special purpose machine features in implementation, e.g., one can save time by 'syntactically inverting' transfer functions from one data structure representation to another, thus generating the 'inverse transfer function'. Some work on this is at present being carried out at Glasgow University (Patterson & Vigor 1967).

In the *mapped form* one can carry out subclassification of the properties of the constructor and selector functions, and hence of the structural part of data structures, according to the following criteria:

1. number and types of function arguments;
2. number and types of function results;
3. relations existing intra and inter arguments and results;
4. invariant properties of the system under the action of the function;
5. information loss during processing;
6. symmetries in the domain of arguments or results (i.e., mapping of one subdomain onto another in the domain or range of the function).

These properties can also be applied to classification of the 'contents' parts of data structures.

I have been developing over the last year a language called D.D.P.S. (Data Directed Programming System) (Vigor 1967) whose features depend on the concepts briefly described in the first part of this paper.

The internal representation of objects is, in the present version, mainly in the mapped and extended forms, although input, output and backing store messages are at present in a coded form. (This may prove to be a fundamental limitation to the usefulness of the system, as it may obscure the

fundamental simplicity of results requested or obtained from the system.) The objects with which I can operate are the nodes (called switches) in a multifurcating multirelation directed ordered graph, with labelling on both the relations and the nodes. The representation chosen for each node is a map of its structure in the form of:

1. a counter giving its external size in the machine;
2. the address of the indexing operation or 'structure'; and
3. the base address of the contents in its present abode (core store or backing store).

Associated with all nodes which are externally available:

1. there is a name which is the first property and is a character string in coded form;
2. on any node is a 16-valued marker called an item modifier which can be used for tagging;
3. there is a list of addresses of selector functions which allows the retrieval of the next item in any given relation class;
4. there is a node marker, which can be on any switch, which specifies if this switch is to be used in *procedure* or *data* mode. In procedure mode the contents of the active part of a switch are copied, i.e., the selection implied on entry is carried out. In data mode the whole switch word is copied (i.e., the structure and contents are transferred with no operation. By changing this marker from 'data' to 'procedure', we can select, on our next entry into the data structure, the particular sub-structure which is 'relevant' to our particular problem situation. This gives us a method for constructing a program assembler.

The external representation is completely flexible. In it new facilities can be added at will by a 'macro' facility, which allows one to use all the power for dynamic data definition inside each block of the language.

Apart from the general representation discussed above, there are available for specific commonly used types of objects a complete set of constructors and selectors. These carry out updating, compounding, pruning and retrieval of sub-items in the 'natural' representation of this object type. For example (Fig. 1), a 2-array can be represented as

- (a) A tree with the same number of items on each branch.
- (b) For matrix operation it may be in the mapped form with a pair of index registers to index it.
- (c) For output it may be stored in the coded form as a page layout.

Transfer functions between these different representations are available. These are examples of designators and in most cases are one-one and on to. The nature of allowable transformations on programs, and the related data changes to keep the semantics invariant, are being studied in order to gain more understanding of the problem of equivalence of programs, and also to get better criteria for choice of optimal representations.

ABSTRACT FOUNDATIONS

When setting up a problem on the computer in this way, the initial input would be a functional definition of the problem in terms of subfunctions and names of data items (which one need not have previously defined). Each of these is assigned to an empty 'switch' in which relations are gradually built up. A tree of allowable 'machine features' for the particular machine is used to generate questions, to the user on an on-line teletype, concerning the relations between objects in his problem and the domain of validity of the functions he is defining.

The relations holding over the set of constructors and selectors of the data structure or part thereof, are successively mapped on to the relations holding between machine features. When a relation fits there is a test to see if the machine features concerned can map this particular section of the data structure.

The cost C of this 'fit' can be given by an expression of the form

$$C = (t_T + t_O)P_F$$

where t_T = time taken to transfer the data at this stage of the program into the required form and location for the machine feature to act on it;

t_O = time associated with operations expected to be carried out on this section of the data structure;

P_F = machine feature price.

A crude total 'cost' for the job is obtained and a corresponding program is chosen as a specification of the representation. The selector and constructor functions for the chosen machine features are substituted into the 'function' or 'macro' definition table with pointers from the corresponding external names of these constructors and selectors in the problem specification.

When a program to carry out some operation on the structure is assembled, this constructor and selector specification is made available by changing a switch marker from data to procedure node, on that branch which is selecting the next function to be substituted, in the program stream, in place of the external names.

The simplest cases one can consider as relational properties are the properties of (1) associativity and (2) commutativity of function arguments. (1) can be mapped on to the property of sequential indexing in a real computer. (2) can be shown by storing all the characters at the same hierarchical level in store.

A more striking example is the mapping of symmetrically related objects on to the same area of store, in those operations for which symmetry results in equivalent processes being carried out on the symmetrically related 'names'. The 'ordinal' property of the cardinal numbers denoting set size can be used as criteria for ordering sets in an 'optimal manner'.

The above methodology has been hand tested in two problems, Tic Tac Toe (Noughts and Crosses) and Checkers (Draughts). From a very simple specification of the problems, fairly efficient internal representations were

obtained. No playing strategy has been considered and the tests are still at a fairly rudimentary stage. The results are highly encouraging. A complete checkers board situation can be represented in three words of 48 bits with very simple move and capture generation for both Queens and Pawns, and a trivial Queening operation.

The increase in the capability of a programmer, if he is freed from the burden of choosing a detailed representation for his problem, cannot as yet be fully assessed. The vast leaps made in the complexity of feasible programs on the introduction of ALGOL and FORTRAN are indicative of the advances likely to come from such a step.

The objects which we have been storing and manipulating, and the functions which manipulate them, are simple 'concepts' according to the definition at the start of this paper.

The transformations which can be carried out on programs by changing the representation of the data and the accessing functions are capable of generalisation. By systematic substitution of simple subfunctions for others of the same nature (same number and types of arguments) we can generate analogous programs (Poplestone, pp. 185-94 of this volume, for an example in finite field arithmetic). A method of concept formation, closely related to Foster's syntactic generalisations (J. M. Foster, pp. 195-203 of this volume) is that obtained by destroying certain invariances or establishing new ones inside the system, and choosing functions to conform with these. We obtain, by a combination of these methods, a type of semantic generalisation which we might come to think resembled human conceptualisation.

REFERENCES

- Church, A. (1941). The calculi of lambda-conversion. *Annals of Mathematical Studies*, No. 6. Princeton N.J.: Princeton University Press.
- Dennis, J. B. (1965). Segmentation and the design of multiprogrammed computer systems. *J. Assn Comput. Mach.*, 12, 589-602.
- English Electric Leo Marconi (1965). KDF9 ALGOL User's Manual, *E.E.L.M. Document*.
- Hunt, Earl B. (1962). *Concept Learning—An Information Processing Problem*. New York and London: Wiley.
- I.B.M. Corp. (1965). 'PL1' Systems Reference Library File No. 5360-29, Form (28-6571-0).
- Landin, P. J. (1964). The mechanical evaluation of expressions. *Computer J.*, 6, 308-320.
- Landin, P. J. (1965). A correspondence between ALGOL-60 and Church's Lambda notation. *Communs Assn Comput. Mach.*, 8, 89-101, 158-165.
- Landin, P. J. (1966). The next 100 programming languages. *Communs Assn Comput. Mach.*, 9, 157-166.
- McCarthy, J. (Ed.), Abrahams, P. W., Edwards, D. J., Hart, T. P., & Levin, M. I. (1962). *LISP 1.5 Programmer's Manual*. M.I.T. Press.

ABSTRACT FOUNDATIONS

- Newell, A. (Ed.), Baker, C. L., Feigenbaum, E. A., Green, B. F., Jr., Kelly, H. S., Mealy, G. H., Saber, N., Tonge, F. N., Van Wermer, T., & Wolf, A. K. (1961). *'IPL5 Programming Manual'*. Rand Corporation, Santa Monica, California.
- Patterson, J., & Vigor, D. (1967). *On the Inversion and Optimisation of Usercode Programs*, Glasgow Univ. Computing Dept. Report No. 2.
- Roberts, L. G. (1965). 'Graphical Communication and Control Languages.' A Lincoln Lab. M.I.T. Report.
- Rutledge, J. D. (1964). Janov schemata. *J. Assn Comput. Mach.*, **11**, 1.
- Scoins, I. (1967). Linear graphs and trees. *Machine Intelligence 1*. N. L. Collins and D. Michie (eds.), pp. 3-15. Edinburgh: Oliver and Boyd.
- Smith, D. L. (1966). Models and data structures for digital simulation. *Project MAC-TR-31*, M.I.T. Report.
- Strachey, C. (1965). A general purpose macro generator; *Brit. Computer Journal*, **8**, 225-241.
- Vigor, D. (1967). *D.D.P.S. Programming Manual*, Glasgow Univ. Computing Dept. Report No. 3.

MECHANISED MATHEMATICS

4

AN APPROACH TO ANALYTIC INTEGRATION USING ORDERED ALGEBRAIC EXPRESSIONS

L. I. HODGSON

COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF MANCHESTER

INTRODUCTION

This paper describes an attempt to develop a system of mathematical analysis which is suitable for use by a computer. The ideas are applicable to many variables but in order to 'get the feel' the analysis is restricted to one variable. Together with this mathematical approach the techniques which are used to implement these methods on the computer are discussed. These computing techniques, however, are straightforward and require no further explanation until they are discussed in detail (pp. 53-56).

In order to develop a system of analysis with particular reference to computer use it is necessary to examine the existing analysis to determine why this is not fully suitable for computer use. Several operations in conventional analysis can be performed by a computer in a fairly straightforward manner. In order to perform these operations except at a trivial level it is necessary to use the concept of 'function' which is not clearly defined but is taken to be what the average mathematician means by a function, e.g.,

$\cos(x)$
or $\exp(x)$
or $p(x)$

One operation which has been treated by such computer techniques is differentiation. These treatments usually involve storing within the computer the rules for differentiation and the properties of the various functions that may arise. Obviously the derivative of any function, or any expression

involving that function, whose properties are not stored cannot be evaluated, thus making the treatment restrictive. The stored properties of the functions needed for differentiation have been developed by generations of mathematicians using more fundamental properties of the said functions and the rules for differentiation. It would seem to be a reasonable approach to develop an analysis in which a definition of a function enabled its derivative to be evaluated in a mechanical manner. This is the approach used in the analysis developed.

Indefinite integration is more difficult than differentiation and the cleverest mathematician could be set a problem by a novice which the expert could not solve and which he couldn't show that he couldn't solve. It would therefore seem unreasonable to expect a mere computer to deal with the problem of integration as usually posed. So a different approach to the concept of integration is presented which is explained in detail later.

MATHEMATICAL ANALYSIS

This analysis involves constants from the field of numbers, an indeterminate x , positive integers and functions of x . The first three of these can be used to form quantities termed subunits defined as follows

$$\text{const. } x^{(\text{power})}$$

with specific examples of subunits

$$2x^4, -75 \cdot 27x^{29} \text{ and } (2-i)x^{71}.$$

By additions of subunits a string ξ can be formed

$$\text{e.g., } \xi = 2x^4 - 75 \cdot 27x^{29} + (2-i)x^{71}.$$

Differentiation of a string ξ formed of such subunits is the sum of the derivatives of the subunits.

For example, if $\xi = a_1x^{n_1} + a_2x^{n_2} + \dots + a_r x^{n_r}$
then $\xi' = n_1 \cdot a_1 x^{n_1-1} + n_2 a_2 x^{n_2-1} + \dots + n_r \cdot a_r x^{n_r-1}.$

The derivative of a constant is zero.

The functions mentioned earlier can now be defined by using strings and subunits.

For example, $\psi'(x) = \psi(x)$ defines within an arbitrary constant the function $\psi(x)$.

A function ψ can be defined by the following general method. The function ψ has a derivative ψ' and this derivative is set equal to a string which has subunits consisting of some or all of the following elements:

- (i) a variable x ,
- (ii) the function,
- (iii) other functions defined earlier by a similar technique.

Examples of these functions are

$$\psi_1(x), \psi_2(x) \text{ and } \psi_3(x)$$

where

$$\begin{aligned}\psi_1'(x) &= \psi_1(x), \\ \psi_2'(x) &= \psi_1(x^2), \\ \psi_3'(x) &= \psi_1(x) + \psi_3(x).\end{aligned}$$

Strings involving these functions can be generated for example ξ_1 and ξ_2 where

$$\begin{aligned}\xi_1 &= ax^5\psi_1(x) - 7 \cdot 2x^{27}\psi_2(x), \\ \xi_2 &= (a+ib)x^7\psi_2(x) + 7x^{29}\psi_1(x).\end{aligned}$$

A more detailed examination of integration can now be presented. From a naive point of view, integration is just the reverse of differentiation. However, all that is needed from integration is an algebraic string ξ which can be used in exactly the manner as strings ξ_1 or ξ_2 defined earlier. Many such strings can be chosen to satisfy the integral, but it is obviously desirable to choose the simplest of these strings.

Therefore if the integral is not expressible as a simpler combination of simpler functions than itself then choose a new function, whose derivative is equal to the integrand, as the value of the integral.

For example, if $\psi_1'(x) = \psi_1(x)$ and $\int \psi_1(x^2)dx$ is required, then if no simpler combination of simpler functions than $\psi_1(x^2)$ satisfy the integral, then define a new function $\psi_2(x)$ where $\psi_2'(x) = \psi_1(x^2)$.

In order to make any progress with this idea of integration a more precise definition of the meaning of simple is needed. Any definition will do provided it is consistent, e.g.,

$$\text{if } x_1 > x_2 \text{ and } x_2 > x_3, \text{ then } x_1 > x_3.$$

The method used involves the following parameters:

- (i) length of string, i.e., number of subunits;
- (ii) multiplicity of subunits, i.e., number of functions in the subunit;
- (iii) depth of the string, i.e., the maximum depth of the function tree.

To illustrate the ideas consider 2 strings ξ_1 and ξ_2 . Let m_1 be the maximum of the length, multiplicity and depth of ξ_1 and m_2 be the corresponding quantity for ξ_2 . If $m_1 > m_2$ then $\xi_1 > \xi_2$. If $m_1 = m_2$ then $\xi_1 > \xi_2$ if the depth of $\xi_1 >$ depth of ξ_2 . If, however, the depth is the same, then test the multiplicity of the two strings and if there is equality then test the corresponding lengths. If none of these tests is conclusive, then compare the corresponding substrings. These substrings are arranged in order of decreasing complexity.

$\xi_1 = \xi_2$ if none of these tests is conclusive.

If the approach to integration is to be any use then there must be a method to test whether a selected string is equal to the integral. This technique is demonstrated by considering the following problem.

$$\begin{aligned}\text{Is} & \quad ax^n\psi_2 = \int bx^m\psi_1 dx \\ \text{with} & \quad \psi_1'(x) = px^q\psi_1(x) \\ & \quad \psi_2'(x) = rx^s\psi_2(x)\end{aligned} \tag{1}$$

and $n > 0, m > 0, p > 0, q > 0$?

Now if (1) is true, then its derivative is also true, i.e.,

$$nax^{n-1}\psi_2 + arx^{n+s}\psi_2 = bx^m\psi_1$$

or

$$(nax^{n-1} + arx^{n+s})\psi_2 = bx^m\psi_1 \quad (2)$$

Differentiating (2) gives

$$\begin{aligned} & [n(n-1)ax^{n-2} + ar(n+s)x^{n+s-1} + nra x^{n+s-1} + ar^2x^{n+2s}]\psi_2 \\ & = [mb x^{m-1} + bp x^{m+q}]\psi_1. \end{aligned}$$

On eliminating ψ_1 and ψ_2 assuming neither are zero, the result is

$$\begin{aligned} & bx^m[n(n-1)ax^{n-2} + ar(n+s)x^{n+s-1} + nar x^{n+s-1} \\ & + ar^2x^{n+2s}] = [mb x^{m-1} + bp x^{m+q}][na x^{n-1} + ar x^{n+s}]. \end{aligned}$$

Re-arrangement gives

$$\begin{aligned} & abx^{m+n-2}[mn + mr x^{s+1} + pn x^{q+1} + pr x^{q+s+2}] \\ & = abx^{m+n-2}[n(n-1) + r(n+s)x^{s+1} + nr x^{s+1} + r^2x^{2s+2}] \end{aligned} \quad (3)$$

This relation must be true for all values of x for (1) to be true and so the factor $ab x^{m+n-2}$ can be cancelled. None of the powers of x can be zero as this would contradict the assumptions concerning q, s .

Consequently $n(n-1) = m.n$, with the result that $n=0$ or $m=n-1$. $n=0$ contradicts the initial assumptions, therefore $m=n-1$. With this value for m equation (3) reduces to

$$\begin{aligned} & (n-1)rx^{s+1} + pn x^{q+1} + pr x^{q+s+2} \\ & = (2nr + rs)x^{s+1} + r^2 x^{2s+2} \end{aligned} \quad (4)$$

For this relation to be true,

$$\begin{aligned} \text{either} & \quad 2s+2 = q+s+2 \\ \text{or} & \quad 2s+2 = q+1 \\ \text{or} & \quad s+1 = q+s+2 \end{aligned}$$

If $2s+2 = q+1$, then the coefficient of x^{q+s+2} must be zero, which gives $pr=0$ contradicting the assumptions.

If $s+1 = q+s+2$, then $q=-1$, which contradicts the initial assumptions. Therefore $2s+2 = q+s+2$, which gives $s=q$.

Equation (4) reduces to

$$\begin{aligned} & [(n-1)r + pn]x^{s+1} + pr x^{2s+2} \\ & = (2nr + rs)x^{s+1} + r^2 x^{2s+2} \end{aligned} \quad (5)$$

giving $pr=r^2$, which is $p=r$ or $r=0$. The latter contradicts the initial assumptions and so $p=r$ is taken. The other condition from (5) is

$$(n-1)r + rn = 2nr + rs$$

which gives $2n-1 = 2n+s$, i.e.,

$$s = -1$$

which contradicts the initial assumptions. Hence the integral is not equal to the expression (1).

The value $n=0$ although not permissible in the context of the question suggests the relation

$$a\psi_2 = \int bx^m\psi_1 dx$$

with

$$\psi_1'(x) = px^q \psi_1$$

$$\psi_2'(x) = rx^s \psi_2$$

and $m > 0, q > 0, s > 0$.

Following a similar technique to that used earlier gives

$$ar x^s \psi_2 = bx^m \psi_1$$

$$\text{and } bx^m(ars x^{s-1} + ar^2 x^{2s}) = ar x^s(mb x^{m-1} + bp x^{m+2})$$

$$\text{or } abr x^{m+s-1}(s + r x^{s+1}) = abr x^{m+s-1}(m + px^{q+1})$$

$$\text{i.e., } s + r x^{s+1} = m + p x^{q+1}$$

giving $m = s, q = s$ and $p = r$, with a and r undetermined.

These two examples are carried out using expressions linear in the ψ and so represent a special case but give a good overall idea of the methods used. Further progress requires more precise specification of the type of functions and strings that are used. These strings fall into three categories:

- (i) linear in the functions;
- (ii) non-linear in the functions but algebraic;
- (iii) non-algebraic in the functions.

Category (i) is solvable using the techniques mentioned earlier and is to be investigated by means of a computer.

Category (ii) is solvable but the techniques are not general enough to use computer programs.

Category (iii) is not known to be solvable although there is no reason to suspect that it is not.

The structure of category (i) is arranged in ascended order of complexity for illustration.

$$\begin{aligned} \psi_1' &= ax^n \\ \psi_1' &= ax^n \psi_1 \\ \psi_2' &= ax^n + bx^m & m \neq n \\ \psi_2' &= bx^m \psi_1 \\ \psi_2' &= ax^n \psi_2 + bx^m \\ \psi_2' &= ax^n \psi_2 + bx^m \psi_1 \\ \psi_3' &= ax^n + bx^m + cx^p & m \neq n \neq p \\ \psi_3' &= cx^p \psi_2 \\ \psi_3' &= bx^n + cx^p \psi_2 \\ \psi_3' &= ax^n + bx^m \psi_2 + cx^p \psi_1 \dots \\ \psi_4' &= ax^n + hx^m + cx^p + dx^q & n \neq m \neq p \neq q \dots \end{aligned}$$

Integration as defined earlier consists of determining which of these strings which do not have the same ψ on both sides of the expression are related to simpler functions.

Much more investigation needs to be carried out on categories (ii) and (iii), which are of greater interest. Category (ii) corresponds roughly to the functions used in conventional analysis although it is designed to generate any functions required, as indeed are all the categories.

The existence of the function ψ is examined by means of 'connected regions'. The technique is illustrated by considering the class of functions defined by

$$\psi'(x) = \psi(x).$$

Take the point $M (>0)$ in ψ space. A near point $M_1 (=M + M\delta x)$ can be made to approach M as $|\delta x| \rightarrow 0$ but is distinct from M if $|\delta x| \neq 0$. Similar points $M_2, M_3, \dots$ and M_n can be developed where $M_n = M(1 + \delta x)^n$. If $|\delta x| \neq 0$ then $M \neq M_1 \neq M_2 \neq \dots \neq M_n$ but the points will approach equality as $|\delta x| \rightarrow 0$. Therefore starting from M a region of ψ space can be constructed every point of which can be reached from M . However if $M=0$ then whatever the value of $|\delta x|$, provided that it is bounded, will always give $M_1 = M_2 = \dots = M_n = M$. Points with this property are termed 'lonely points'. As the direction in a connected region can be reversed that is if $M_1 \rightarrow M_2$ then $M_2 \rightarrow M_1$, then it follows that as a lonely point cannot be left then it cannot be reached. Hence the name 'lonely point'.

A function ψ may have more than one connected region, but these must be separated by lonely points. A function ψ which consists only of lonely points is said to be non-existent.

Any function ψ defined by the method given on page 48 is determined by an arbitrary constant, i.e., if $\psi(x)$ satisfies $\psi'(x) = \psi(x)$ then so does $\psi(x+c)$. For use in function strings and trees in general the function ψ must be set to a specific value although the particular value chosen is usually immaterial. Choosing the arbitrary constant is equivalent to relating the connected regions of ψ space to x space. Therefore the value of the particular constant must give the value of the function inside a connected region.

Consider for example $\psi_1'(x) = -\psi_1^2(x)$ and let the value $\psi_1 = 1$ be related to $x=0$, i.e., $\psi_1(0) = 1$. One of the properties of the class of functions ψ_1 is that $x\psi_1(x) + c\psi_1(x) = 1$ and substitution of $\psi_1(0) = 1$ in this gives $c=1$. Therefore the particular ψ_1 chosen from the general class $\psi_1'(x) = -\psi_1^2(x)$ satisfies

$$x\psi_1(x) + \psi_1(x) = 1$$

This relation is equivalent to

$$x \cdot \frac{1}{x} = 1$$

of conventional analysis, or more precisely

$$(x+1) \cdot (x+1)^{-1} = 1.$$

To further illustrate these ideas, consider the functions

$$\begin{aligned} \psi_1'(x) &= \psi_1(x), & \psi_1(0) &= 1 \\ \psi_2'(x) &= -\psi_2^2(x), & \psi_2(0) &= 1 \\ \psi_3'(x) &= \psi_2(x), & \psi_3(0) &= 1 \end{aligned}$$

which give rise to the string

$$\psi_1(\psi_3(x)) = x + 1$$

or on re-writing

$$k(x) = x + 1$$

where $k'(x) = k(x) \cdot \psi_2(x)$.

Now, differentiation of $k(x)=x+1$ gives

$$k(x)\psi_2(x)=1$$

and substitution for $k(x)$ yields

$$x\psi_2(x)+\psi_2(x)=1$$

which is the inverting relationship mentioned earlier. The relationship $k(x)=x+1$ is equivalent to the conventional relationship $\exp(\log_e x)=x$ and the analysis above shows this to be equivalent to

$$.x.\frac{1}{x}=1.$$

The analysis above also indicates how to use functions precisely defined to prove relationships. An interesting point is that the function ψ_3 is defined by an integral relationship, i.e.,

$$\psi_3 = \int \psi_2(x) dx$$

but is used in a straightforward manner in developing strings. Other such relationships and functions have been studied, but are omitted in order not to cloud the main principles with detail. One of the objects of the author in carrying out this research was to investigate such relations on a computer.

PROGRAMS WRITTEN TO EXPLORE THE IDEAS

The system of programs developed to investigate the ideas discussed earlier was designed to allow flexibility. This was achieved by writing the program in several phases. The first phase was a language of a similar nature to the compiler compiler of R. A. Brooker *et al.* (Rosen 1964) but very much simpler. The input for this program was: (i) phrase definitions; (ii) formats; (iii) format routines; (iv) machine orders used by format routines; (v) a few alphanumeric lines, e.g., NEWLINE & END.

When these categories had been input then the phase 1 program entered phase 2, which consisted of reading one line of input and then comparing with the format list until recognition was achieved. If recognition had not been achieved when the format list was exhausted, then phase 2 program printed a suitable caption and also set a fault indicator.

The following list of phrases and formats defines the function strings manipulated by the system:

```

PHRASE [1] =0, 1, 2, 3, 4, 5, 6, 7, 8, 9
PHRASE [2] =[1] [2], [1]
PHRASE [3] =[2].[2], .[2], [2]., [2]
PHRASE [4] =A[2], +[3], -[3], -
PHRASE [5] =[0][2]:[19][0], [0][2]:[0], [0]:[19][0]
PHRASE [6] =ψ[5] ( ↓ [2])
PHRASE [7] =[6][7], [6]
PHRASE [8] =[4]X[5][7], [4][7], [4]X5, [4]
PHRASE [9] =[8][9], [8]
```

MECHANISED MATHEMATICS

PHRASE [10] = [0], >, c
 PHRASE [11] = <0[2], ↓ [2], ψ'[2]
 PHRASE [19] = [2], N[2]
 PHRASE [20] = +, -
 PHRASE [21] = A[2][21], A[2]
 PHRASE [22] = [20][21][22], [20][21]
 PHRASE [23] = N[2][23], N[2]
 PHRASE [24] = [20][23][24], [20][23]

The corresponding format line which uses these phrases is

FORMAT [13] = [11] = [9][10]

which defines a general type of function tree satisfying any of the three categories of page 51.

Further explanation of the phrase list is necessary to clarify it. The number between square brackets after the word PHRASE is the phrase number and identifies that phrase. Whenever a number between square brackets occurs on the right-hand side of a phrase definition then that number refers to the phrase of that number. Alternative phrases on the right-hand side of a phrase definition are separated by the symbol ','. Finally the system has a built in definition

PHRASE [0] = [,] , ,

which allows these symbols to be part of a phrase definition.

A more detailed examination of

FORMAT [13] = [11] = [9][10]

shows that [11] defines the type of string as follows:

- <0[2] indicates that the string of number [2] has just been started;
- [2] indicates that the string number [2] is an argument of some function;
- [2] indicates that the string defines a function [2] needed in the overall function tree.

PHRASE [10] defines the end of the string if the category of [10] is [0] then the function tree is not complete but the current string is. If the category is > then both the function tree and the current string are complete.

Finally if the category is c then the string is not complete and will be finished on the next line.

PHRASE [9] defines the right-hand side of the string with the exception of the terminator [10]. As can be seen from the definition of [9] it consists of a series of subunits which are defined by [8].

PHRASE [8] can be of four possible types, but the most general is [4]X[5][7] and is of the form: const. variable power.functions.

PHRASE [5] defines a type of variable or function by two numbers, one of which is the variable or function number and the other is the power of the variable or function.

PHRASE [7] defines the product of functions or a single function.

PHRASE [6] is the single function ψ followed by [5] and finally by an argument $\downarrow [2]$.

A simple example of a function corresponding to FORMAT [13] is

$$\begin{aligned} &\langle \emptyset 1 = +A1x[1:2]\psi[1:2](\downarrow 1) + A2x[1:1], \\ &A2 = +A2, \\ &A1 = +A1, \\ &\downarrow 1 = +79 \cdot 2x[1:N1], \\ &N1 = +N1, \\ &\psi'1 = +A1\psi[1:1](\downarrow 1) \rangle \end{aligned}$$

All these phrase and format definitions define the 'form' of the function string or tree, but provide no facilities for changing the function tree in a manner similar to that in conventional languages which allow variables to be changed dynamically. Nor do they allow decisions to be taken depending on the state of the function tree. The following list of phrases and formats allow such facilities.

PHRASE [25] = [2]: $\emptyset$ [2]:, $\emptyset$ [2]:
 PHRASE [26] = $\emptyset$ [2], $\downarrow [2]$, $\psi^1[2]$
 PHRASE [27] = $N[2]$, $A[2]$
 PHRASE [28] = i, j, k, l, m, n
 PHRASE [29] = [2], $-[2]$, [28]
 PHRASE [30] = $\emptyset$ [29], [29], [29]
 PHRASE [31] = $L\emptyset$ [29][30], $M\emptyset$ [29][30]([39])
 PHRASE [32] = $>, >, =, <, <, \neq$
 PHRASE [33] = [31], [29]
 PHRASE [34] = $+, -$
 PHRASE [35] = [2]: $->$, $->$
 PHRASE [36] = [2]: [28], [28]

FORMAT [23] = $\emptyset$ [2]: = $\emptyset$ [2]
 FORMAT [24] = [2]: $\emptyset$ [2] = $\emptyset$ [2]
 FORMAT [25] = [25][26]: = [9] [10]
 FORMAT [26] = [25][27]: = [27][10]
 FORMAT [27] = [25] [26]: = [26][9][10]
 FORMAT [28] = [35] [2] if [33] [32] [33]
 FORMAT [29] = [35][2]
 FORMAT [30] = [36] = [29] [34] [29]
 FORMAT [31] = [36] = [33]

Formats 23-27 allow function trees to be changed dynamically.

FORMAT [28] allows conditional decisions to be taken. For example

or $-> 3$ if $L\emptyset 3\psi'1 < 4$
 or $-> 3$ if $L\emptyset 3\emptyset 3 < 4$
 or $-> 3$ if $M\emptyset 3\psi'1(4) \neq 7$

MECHANISED MATHEMATICS

The indicator L denotes the length of the string which is defined by the two string names immediately adjacent to the L .

The letter M indicates the multiplicity of the subunit in the string defined by the two string names. The particular subunit is determined by the quantity in the brackets. By the use of these conditional instructions and the dynamic alteration of function trees the programming system becomes much more powerful.

The immediate object of the programming system is to investigate more fully categories (i) and (ii). The use of a computer enables the investigation to be carried out much more rapidly and may help to suggest ideas to solve many remaining problems of which the following are examples:

- (i) how to deal with category (iii);
- (ii) how to determine the true properties of defined functions;
- (iii) to investigate the solution of differential equations;
- (iv) study n -dimensional space.

REFERENCE

Rosen, Saul (1964). A compiler building system developed by Brooker and Morris. *Communs Assn Comput. Mach.*, 7, 403-414.

SOME THEOREM-PROVING STRATEGIES BASED ON THE RESOLUTION PRINCIPLE

JARED L. DARLINGTON

RHEINISCH-WESTFÄLISCHES INSTITUT
FÜR INSTRUMENTELLE MATHEMATIK
BONN, GERMANY

The formulation of the resolution principle by J. A. Robinson (1965a) has provided the impetus for a number of recent efforts in automatic theorem-proving. In particular, the program PG1 (Wos *et al.* 1964, 1965), written by L. Wos, G. A. Robinson and D. F. Carson for the Control Data 3600, utilises the resolution principle in conjunction with a 'unit preference strategy' and a 'set of support strategy' to produce efficient proofs in first-order functional logic and group theory. The present author also has experimented with the resolution principle, and has incorporated it into a pair of COMMIT theorem-proving programs written at the Institut für Instrumentelle Mathematik in Bonn and currently running on the Institute's IBM 7090. These programs have generated proofs of some interesting propositions of number theory, in addition to theorems of first-order functional logic and group theory.

The resolution principle, like many other theorem-proving methods, is a refutation algorithm, in the sense that it seeks to generate a contradiction from an initial set of clauses

$$C_1, C_2, \dots, C_k$$

resulting from the negation of the theorem or formula to be tested, usually in conjunction with some already established axioms or theorems. These initial or input clauses C_i are a set of logical expressions in conjunctive normal form: that is, the C_i are regarded as joined by logical 'and', while an

individual clause is either a single literal L , in which case it is called a 'one-literal clause' or a 'unit clause', or a disjunction of n literals

$$L_1 \vee L_2 \vee \dots \vee L_n$$

which case it is called an ' n -literal clause' or simply an ' n -clause'. A 'literal' is an n -place predicate expression

$$F(x_1, x_2, \dots, x_n)$$

or its negation

$$\bar{F}(x_1, x_2, \dots, x_n)$$

whose arguments are individual variables, individual constants, or functional expressions. Quantifiers do not occur in these formulae, since existentially quantified variables have been replaced by functions of universally quantified ones, and the remaining variables may therefore be taken as universally quantified. For example, the number-theoretic proposition

'For all x and y , if x is a divisor of y then there exists some z such that x times z equals y '

may be symbolised as

$$(i) \quad \bar{D}(x, y) \vee T(x, f(x, y), y)$$

in which ' $D(x, y)$ ' stands for ' x is a divisor of y ' and ' $T(x, y, z)$ ' stands for ' x times y equals z '. The formula (i) is a 2-literal clause, and may be read as follows:

'Either x is not a divisor of y , or there is a function $f(x, y)$ of x and y such that x times $f(x, y)$ equals y '.

Another example is the 'first proposition of Euclid':

'For all x , y , and z , if x is a prime and x is a divisor of $y.z$, then x is a divisor of either y or z '.

which may be symbolised as

$$(ii) \quad \bar{P}(x) \vee \bar{T}(y, z, u) \vee \bar{D}(x, u) \vee D(x, y) \vee D(x, z).$$

Both (i) and (ii) are general formulae, which admit of infinitely many instantiations, or substitution instances, within the positive integers. Exhaustive instantiation, in fact, was the basis of a number of early theorem-proving programs. Starting with a finite number of individual constants, all possible substitutions were made in the input clauses, and the resulting conjunction of substitution instances was tested for truth-functional consistency. If a contradiction was not obtained, more constants were produced by substituting the available constants for the variables in the functional expressions $f(x, y)$, etc., that stood for the existential variables, and the additional constants were used to generate more substitution instances. This process continued until a contradiction was obtained or (the more usual case) the machine ran out of time or storage. The Herbrand theorem guaranteed that exhaustive instantiation would eventually produce a proof if one existed, given enough time and storage, but the large number of irrelevant substitution instances

generated often prevented the machine finding proofs of even relatively simple theorems. It soon became evident, therefore, to researchers in the field that the primary problem in automatic theorem-proving is that of impeding the generation of irrelevant clauses and inferences. A way in which this might be done was suggested by D. Prawitz *et al.* (1960) and elaborated upon by M. Davis (1963). The basic idea was that any substitution instance of a clause is bound to be irrelevant to a proof so long as the literals occurring in it are not 'mated', i.e., negated, by other literals occurring in other clauses. Davis in fact proved that any substitution instance

$$L_1 \vee L_2 \vee \dots \vee L_n$$

that contains at least one unmated literal may be erased without affecting consistency, and that the test for consistency may be confined to 'linked conjuncts', i.e., conjuncts (or clauses) wherein each literal L_i is negated by a mate $\bar{L}_i$ occurring in some other clause. This 'theorem on linked conjuncts' provides a necessary, though not a sufficient, condition of relevance: any substitution instance containing an unmated literal is demonstrably irrelevant to consistency and may be deleted forthwith, but the remaining substitution instances are not necessarily all relevant. Davis showed (1963) that a theorem-proving strategy based on the search for linked conjuncts is capable of producing short proofs of many theorems, and gave as a detailed example the proposition that any left inverse in an associative group is also a right inverse.

The resolution principle takes over the idea of searching for mated literals, but applies it directly to the original clauses rather than to their substitution instances. Instead of searching for mates among the substitution instances, the resolution principle employs an algorithmic procedure to determine in advance whether two literals L_1 and L_2 in two different clauses could yield contradictory substitution instances. For example, if L_1 and L_2 are

$$(iii) \quad T(G(x, y), x, y)$$

and

$$(iv) \quad T(G(u, K(v)), w, K(v)),$$

respectively, it may be determined that these two clauses will yield contradictory substitution instances, and that these (apart from the negation sign in (iv)) are all instances of the formula

$$(v) \quad T(G(x, K(y)), x, K(y)).$$

In the terminology of J. R. Guard (1964), formula (v) may be called a 'general matching formula' ('GMF') for L_1 and L_2 , since it stands for all the common substitution instances of L_1 and $\bar{L}_2$. Whenever a pair of clauses C_1 and C_2 can be found that contain literals L_1 and L_2 such that L_1 matches $\bar{L}_2$, i.e., such that L_1 and $\bar{L}_2$ have common substitution instances, then a 'resolvent' of C_1 and C_2 may be generated by disjunctively joining C_1' and C_2' , where C_1' is formed by making the same substitutions throughout $C_1 - L_1$ that are necessary to make L_1 equal to the GMF, and C_2' is formed by making the same substitutions throughout $C_2 - L_2$ that are necessary to make $\bar{L}_2$ equal

to the GMF. For example, suppose (iv) is replaced by the 2-literal clause
(vi) $\bar{T}(G(u, K(v)), w, K(v)) \vee T(u, v, w)$

in which the first literal is identical to (iv). Then (v) is the GMF of (iii) and the negation of the first literal in (vi), and the latter will equal the GMF (v) so long as the following substitutions are made throughout (vi):

$$u = x, v = y, w = x$$

This leaves

$$(vii) \quad T(x, y, x)$$

as the resolvent of (iii) and (vi). Resolution is thus a functional logic analogue of the operation of 'cut' in propositional logic, whereby a single clause

$$K_2 \vee K_3 \vee \dots \vee K_m \vee L_2 \vee L_3 \vee \dots \vee L_n$$

can be deduced from the two separate clauses

$$K_1 \vee K_2 \vee \dots \vee K_m$$

and

$$\bar{K}_1 \vee L_2 \vee \dots \vee L_n.$$

If C_1 and C_2 are both unit clauses, as in (iii) and (iv), then there is nothing left over after the resolvent is formed: that is, the sole resolvent of two contradictory unit clauses is the null clause. Such a result amounts to a proof that the initial set C of clauses is unsatisfiable. The resolution principle searches for a contradiction between two unit clauses, and if none can be found, it generates more resolvents until a pair of contradictory unit clauses is produced. The algorithm in its original form called for the generation of

$$S_0, S_1, S_2, \dots, S_i, \dots$$

where S_0 is the set of input clauses, and S_i consists of S_{i-1} plus all resolvents and 'factors' (see below) of resolvents of the elements of S_{i-1} . This procedure, though provably complete (as shown by Robinson in 1965a), is extremely wasteful, and is hardly more efficient than the earlier methods based on exhaustive instantiation. The reason for this inefficiency is that resolution normally produces longer and longer clauses (a resolvent of a 3-clause and a 4-clause, for example, is ordinarily a 5-clause, unless one or more of its literals are redundant), while a proof of unsatisfiability requires the generation of shorter, and ultimately unit, clauses. One way of generating shorter clauses is by means of a technique called 'factoring' (by Robinson), whereby a clause C_i may be factored if it contains two matching literals L_1 and L_2 . The factor is formed by making the same substitutions throughout C_i that are necessary to make L_1 equal to L_2 , and then deleting one of these two literals along with any other redundant literals. The following clause

$$(viii) \quad P(x) \vee T(y, y, u) \vee \bar{D}(x, u) \vee D(x, y)$$

is a factor of the earlier example (ii), formed in an obvious way by substituting y for z throughout (ii) and then deleting the redundant literal $D(x, y)$.

A second and more generally applicable technique for generating shorter clauses is to resolve unit clauses against clauses of length $n(n \geq 1)$, thereby

producing clauses of length $n-1$ or less. It turns out in fact that many theorems can be proven by generating only resolvents that have at least one unit clause as parent ('resolvend'), and the resulting simplification of proofs is considerable.

There are several ways of writing a program that gives priority to unit clauses and the resolvents descended from them. In addition to our two COMIT programs, which we may call 'D1' and 'D2', there is the afore-mentioned program PG1 of Wos *et al.* D1 proceeds by generating successively S_0, S_1, S_2 , etc., exactly as the original resolution algorithm, but with the restriction that each resolvent must have at least one unit clause as a parent, or (to allow for the possibility that there may initially be no unit clauses) at least one n -clause ($n \geq 1$) as a parent, where n is the length of the shortest clause. D2 proceeds by resolving the unit clauses (or the n -clauses, as in D1) first against each other, then against the first 2-clause, the second 2-clause, etc., then against the first 3-clause, the second 3-clause, etc. Whenever two clauses C_1 and C_2 produce one or more resolvents R_1, R_2 , etc., then the R_i take priority over the older clauses in the sense that they are stored at the front of their appropriate lists (unit clauses, 2-clauses, etc.) and the last R_i to be generated is the first to be resolved against the unit clauses. PG1 is similar to D2, the main difference being that, while D2 generates all the resolvents R_1, R_2 , etc., of a pair of clauses C_1 and C_2 before proceeding further, PG1 generates only the first resolvent R_1 (or the i th resolvent R_i , if R_{i-1} has already been generated) of C_1 and C_2 and then immediately proceeds to resolve R_1 (or R_i) against the unit clauses (or the n -clauses, where n , as in D1 and D2, is the length of the shortest clause).

The three methods described may be compared in terms of how many resolvents they generate at a time. D1 generates all the resolvents (provided that each resolvent has at least one n -clause as a parent) of a set C ; D2 generates all the resolvents of a given pair C_1 and C_2 (provided that either C_1 or C_2 is an n -clause); and PG1 generates only one resolvent of C_1 and C_2 at a time. The difference between the three programs may be illustrated in terms of the 'Gilmore problem', i.e., the set of clauses

- (1) $F(x, y)$
- (2) $G(x, y) \vee \bar{F}(y, P(x, y)) \vee F(P(x, y), P(x, y))$
- (3) $F(y, P(x, y)) \vee \bar{F}(P(x, y), P(x, y)) \vee \bar{G}(x, P(x, y)) \vee \bar{G}(P(x, y), P(x, y))$

first used as an example by P. C. Gilmore (1960). D1 generated the following proof of this example in 19 seconds (in this and subsequent examples, we shall omit parentheses within literals where no ambiguity can result):

D1 proof of Gilmore example (19 seconds)

- | | |
|--|-----------|
| (1) Fxy | premise |
| (2) $Gxy \vee \bar{F}yPxy \vee \bar{F}PxyPxy$ | premise |
| (3) $\bar{F}yPxy \vee \bar{F}PxyPxy \vee \bar{G}xPxy \vee \bar{G}PxyPxy$ | premise |
| (4) $Gxy \vee \bar{F}PxyPxy$ | (1) & (2) |
| (5) $Gxy \vee \bar{F}yPxy$ | (1) & (2) |

MECHANISED MATHEMATICS

- | | |
|---|------------|
| (6) $\bar{F}PxyPxy \vee \bar{G}xPxy \vee \bar{G}PxyPxy$ | (1) & (3) |
| (7) $\bar{F}yPxy \vee \bar{G}xPxy \vee \bar{G}PxyPxy$ | (1) & (3) |
| (8) Gxy | (1) & (4) |
| (9) Gxy | (1) & (5) |
| (10) $\bar{G}xPxy \vee \bar{G}PxyPxy$ | (1) & (6) |
| (11) $\bar{G}xPxy \vee \bar{G}PxyPxy$ | (1) & (7) |
| (12) $\bar{G}PxyPxy$ | (8) & (10) |

.....

- | | |
|--------------------|------------|
| (22) contradiction | (8) & (12) |
|--------------------|------------|

The gap between (12) and (22) is because nine extra clauses were generated before the contradiction between (8) and (12) was discovered. D1 in fact will usually generate additional clauses after a contradictory pair has been generated, because it does not test each unit clause immediately upon generation, as does PG1, but waits until its 'generation' is complete. Furthermore, clauses (5), (7), (9) and (11) are irrelevant to the proof, and need not be shown in the final printout. Two of these, (9) and (11), are actually redundant, and may be deleted.

D2 generated the following proof, in 10 seconds:

D2 proof of Gilmore example (10 seconds)

- | | |
|--|------------|
| (1) Fxy | premise |
| (2) $Gxy \vee \bar{F}yPxy \vee \bar{F}PxyPxy$ | premise |
| (3) $\bar{F}yPxy \vee \bar{F}PxyPxy \vee \bar{G}xPxy \vee \bar{G}PxyPxy$ | premise |
| (4) $Gxy \vee \bar{F}PxyPxy$ | (1) & (2) |
| (5) $Gxy \vee \bar{F}yPxy$ | (1) & (2) |
| (6) Gxy | (1) & (5) |
| (7) $\bar{F}yPxy \vee \bar{F}PxyPxy \vee \bar{G}PxyPxy$ | (3) & (6) |
| (8) $\bar{F}yPxy \vee \bar{F}PxyPxy \vee \bar{G}xPxy$ | (3) & (6) |
| (9) $\bar{F}yPxy \vee \bar{F}PxyPxy$ | (6) & (8) |
| (10) $\bar{F}PxyPxy$ | (1) & (9) |
| (11) $\bar{F}yPxy$ | (1) & (9) |
| (12) contradiction | (1) & (11) |

Clauses (4), (7) and (10) are superfluous, and may be omitted from the final printout. In addition, an extra Gxy was generated, from (1) and (4), but was immediately deleted.

Finally, we may present the proof that PG1 would produce if it were given the Gilmore example.

PG1 proof of Gilmore example

- | | |
|--|-----------|
| (1) Fxy | premise |
| (2) $Gxy \vee \bar{F}yPxy \vee \bar{F}PxyPxy$ | premise |
| (3) $\bar{F}yPxy \vee \bar{F}PxyPxy \vee \bar{G}xPxy \vee \bar{G}PxyPxy$ | premise |
| (4) $Gxy \vee \bar{F}PxyPxy$ | (1) & (2) |
| (5) Gxy | (1) & (4) |

DARLINGTON

- | | |
|---|-----------|
| (6) $Gxy \vee FyPxy$ | (1) & (2) |
| (7) $FyPxy \vee FPxyPxy \vee \bar{G}PxyPxy$ | (3) & (5) |
| (8) $FyPxy \vee FPxyPxy$ | (5) & (7) |
| (9) $FPxyPxy$ | (1) & (8) |
| (10) contradiction | (1) & (9) |

Only two superfluous clauses are generated: (6), and an extra Gxy from (1) and (6).

On the basis of the Gilmore example, it would appear that PG1 is the best, and D1 the worst, of the three programs described, with D2 falling somewhere in between but rather nearer to PG1. This judgment is confirmed by many examples, of which we may present one more: the proof of the group-theoretical proposition, which we may call 'RI', that any associative system that has left and right solutions contains a right identity element.

First D1 proof of RI (322 seconds)

- | | |
|--|-----------------------------|
| (1) $TGxyxy$ | existence of left solution |
| (2) $TxHxyy$ | existence of right solution |
| (3) $TxyFxy$ | closure |
| (4) $\bar{TK}xxKx$ | no right identity |
| (5) $\bar{Tx}yu \vee \bar{T}yzv \vee \bar{T}xvw \vee \bar{T}uzw$ | associativity |
| (6) $\bar{Tx}yu \vee \bar{T}yzv \vee \bar{T}uzw \vee \bar{T}xvw$ | |
| (8) $\bar{Tx}yu \vee \bar{T}yzy \vee \bar{T}uzu$ | factor of (5) |
| (20) $\bar{Tx}yz \vee \bar{T}zHyyz$ | (2) & (8) |
| (33) $\bar{T}yzKx \vee \bar{T}zxz$ | (4) & (8) |
| (73) $TyHxxy$ | (1) & (20) |
| (77) $\bar{T}xyx$ | (1) & (33) |
| (250) contradiction | (73) & (77) |

Second D1 proof of RI (117 seconds)

- | | |
|--|------------|
| (1) $TGxyxy$ | |
| (2) $TxHxyy$ | |
| (3) $TxyFxy$ | |
| (4) $\bar{TK}xxKx$ | |
| (5) $\bar{Tx}yu \vee \bar{T}yzv \vee \bar{T}xvw \vee \bar{T}uzw$ | |
| (6) $\bar{Tx}yu \vee \bar{T}yzv \vee \bar{T}uzw \vee \bar{T}xvw$ | |
| (7) $\bar{T}yzKx \vee \bar{T}zxu \vee \bar{T}yuKx$ | (4) & (5) |
| (9) $\bar{Tx}yz \vee \bar{T}GxKyzKy$ | (1) & (7) |
| (24) $\bar{T}xyx$ | (1) & (9) |
| (69) contradiction | (2) & (24) |

D2 proof of RI (11 seconds)

- (1) $TGxyxy$
- (2) $TxHxyy$
- (3) $TxyFxy$

- | | | |
|------|--|------------|
| (4) | $\bar{T}KxxKx$ | |
| (5) | $\bar{T}xyu \vee \bar{T}yzv \vee \bar{T}xvw \vee Tuzw$ | |
| (6) | $\bar{T}xyu \vee \bar{T}yzv \vee \bar{T}uzw \vee Txvw$ | |
| (7) | $\bar{T}yzKx \vee \bar{T}zxu \vee \bar{T}yuKx$ | (4) & (5) |
| (10) | $\bar{T}GxKyzKy \vee \bar{T}zyx$ | (1) & (7) |
| (11) | $\bar{T}xyx$ | (1) & (10) |
| (13) | contradiction | (2) & (11) |

PG1 produced a proof of RI essentially the same as that generated by D2 (though PG1 runs much faster than D1 or D2, and obtained its proof of RI in 35 milliseconds).

The latter two proofs depicted above employ an additional heuristic, the 'set of support strategy' of Wos *et al.* (1964, 1965), according to which a subset S of the set C of input clauses is singled out as the 'set of support' of C . This means that every resolvent produced must be a descendant of at least one element of S , i.e., no resolvents are formed entirely from elements of $C - S$. The most natural choice of S is the set of clauses that formulate the special assumptions of the proof and the negation of the conclusion. In the proofs just presented, the set of support consists of the single unit clause (4), the assumption that there exists no right identity element.

Another aspect of the above proofs of RI that should be mentioned is that factoring turns out not to be essential, and is omitted from the latter two proofs, though it is included in the first proof of RI. It is often possible to obtain proofs without factoring, and it is advisable to try to do so, since the generation of factors has the tendency to produce clauses that are too specific to be of any use in a proof. There are, however, some examples in which factoring appears to be essential, such as the following, which we may call 'W5', since it is based on the fifth example of H. Wang (1964) in the appendix (Wang attributes the example to Quine).

D1 proof of W5 (15 seconds)

- | | | |
|------|---|---------------|
| (1) | $\bar{G}ya \vee \bar{G}yw \vee \bar{G}wy$ | premise |
| (2) | $Gya \vee GyPy$ | premise |
| (3) | $Gya \vee GPyy$ | premise |
| (4) | $\bar{G}ya \vee \bar{G}ay$ | factor of (1) |
| (6) | $\bar{G}ya \vee \bar{G}yy$ | factor of (1) |
| (7) | $\bar{G}aa$ | factor of (6) |
| (9) | $GaPa$ | (2) & (7) |
| (10) | $GPaa$ | (3) & (7) |
| (11) | $\bar{G}Paa$ | (4) & (9) |
| (20) | contradiction | (10) & (11) |

In the above proof, factoring plays an essential role in the derivation of the key unit clause (7) in two steps from (1). It is not essential, however, in the derivation of (11), which could have been produced directly by resolution of (1) and (9), instead of first factoring (1) to obtain (4) and then resolving (4) and (9) to obtain (11).

The superiority, implicit in the foregoing examples, of D2 and PG1 over D1 is not always realised in practice, since there are some proofs for which D1's horizontal or across-the-board method of generating clauses is more suitable than the vertical or up-the-ladder procedure of D2 and PG1. One such example is the number-theoretic proof that every integer greater than one has a prime divisor, which was used as an example by S. A. Cook (1965), and which we may call 'C1'.

D1 proof of C1 (51 seconds)

- | | |
|---|---|
| (1) $L1a$ | a is greater than 1 |
| (2) $\bar{P}x \vee \bar{D}xa$ | a has no prime divisor |
| (3) $\bar{L}1x \vee Lxa \vee PKx$ | assumption for induction: every x greater |
| (4) $\bar{L}1x \vee Lxa \vee DKxx$ | than 1 but less than a |
| (5) Dxx | has a prime divisor |
| (6) $Px \vee L1Qx$ | reflexivity of 'divisor' |
| (7) $Px \vee LQxx$ | if x is not a prime, then |
| (8) $Px \vee DQxx$ | x has a divisor greater |
| (9) $\bar{D}xy \vee \bar{D}yz \vee Dxz$ | than 1 but less than x |
| (14) $\bar{P}a$ | transitivity of 'divisor' |
| (15) $L1Qa$ | (2) & (5) |
| (16) $LQaa$ | (6) & (14) |
| (17) $DQaa$ | (7) & (14) |
| (21) $\bar{L}1Qa \vee PKQa$ | (8) & (14) |
| (22) $\bar{L}1Qa \vee DKQaQa$ | (3) & (16) |
| (24) $\bar{D}xQa \vee Dxa$ | (4) & (16) |
| (28) $PKQa$ | (9) & (17) |
| (29) $DKQaQa$ | (15) & (21) |
| (35) $\bar{D}KQaa$ | (15) & (22) |
| (36) $DKQaa$ | (2) & (28) |
| (43) contradiction | (24) & (29) |
| | (35) & (36) |

(Note that clause (1) is not essential to the proof, since it is implicit in clauses (6)-(8).)

Incomplete D2 proof of C1 (116 seconds)

- | | |
|-----------------------------------|------------|
| (1)-(9), as in the D1 proof of C1 | |
| (10) $\bar{P}a$ | (2) & (5) |
| (11) $L1Qa$ | (6) & (10) |
| (12) $LQaa$ | (7) & (10) |
| (13) $DQaa$ | (8) & (10) |
| (14) $\bar{P}Qa$ | (2) & (13) |
| (15) $L1QQa$ | (6) & (14) |
| (16) $LQQaQa$ | (7) & (14) |

MECHANISED MATHEMATICS

(17) $DQQaQa$	(8) & (14)
(18) $\bar{D}Qaz \vee DQQaz$	(9) & (17)
(19) $\bar{D}xQQa \vee DxQa$	(9) & (17)
(20) $DQQaa$	(13) & (18)
(21) $\bar{P}QQa$	(2) & (20)
(22) $L1QQQa$	(6) & (21)
(23) $LQQQaQQa$	(7) & (21)
(24) $DQQQaQQa$	(8) & (21)
(25) $DQQQaQa$	(19) & (24)
(26) $\bar{D}Qaz \vee DQQQaz$	(9) & (25)
(27) $\bar{D}xQQQa \vee DxQa$	(9) & (25)
(28) $DQQQaa$	(13) & (26)
(29) $\bar{P}QQQa$	(2) & (28)
(30) $L1QQQQa$	(6) & (29)
(31) $LQQQQaQQQa$	(7) & (29)
(32) $DQQQQaQQQa$	(8) & (29)
(33) $DQQQQaQa$	(27) & (32)
(34) $\bar{D}Qaz \vee DQQQQaz$	(9) & (33)
(35) $\bar{D}xQQQQa \vee DxQa$	(9) & (33)
(36) $DQQQQaa$	(13) & (34)
(37) $\bar{P}QQQQa$	(2) & (36)

.....
.....

Clauses (1)-(4) in the above two examples are taken as the set of support; together they formulate the assumption that there exists some $a > 1$ that has no prime divisor, but every $1 < x < a$ has a prime divisor. The trouble with the attempted D2 proof of C1 is that it has got itself into a loop, and is unable to form any resolvents from clauses (3) or (4). In order to obtain the proof, it is necessary to resolve (3) and (4) against (11) and (12), thereby producing $PKQa$ and $DKQaQa$, but this requires that unit clauses be resolved against 3-clauses, and there are always 2-clauses that take priority. A simpler example of looping is the tendency of the pair of clauses

$$Px, \bar{P}x \vee PFx$$

to produce

$$PFx, PFFx, PFFFx, \text{ etc.}$$

ad infinitum. This sort of looping was recognised as a problem by the authors of PG1; who programmed around the difficulty by placing a limit, called a 'level bound', on the number of times the unit clauses can be resolved against clauses of length m before going on to the $m+1$ -clauses. This limit is usually set in advance of computation, at around 4 or 5. The 'level' of the input clauses C is 0, the level of a factor of C_i equals the level of C_i , and the level of a resolvent R of C_1 and C_2 is greater by 1 than the maximum of the levels of C_1 and C_2 . If the level bound is set at k , then unit clauses whose level

exceeds k , and non-unit clauses whose level exceeds $k-1$, are simply not generated. In the attempted D2 proof of C1, for example, none of the clauses beyond (17) would be generated if k were set at 5, and the program would be forced to back up and generate the correct resolvents. The level bound strategy depends for its effectiveness upon choosing the correct value of k : if k is too small, the program cannot generate a proof, and if k is too large, the resulting irrelevant unit clauses could seriously delay the generation of a contradiction.

The procedure of D2 and PG1 of exhausting the consequences of the shorter clauses before proceeding to the longer ones creates another kind of difficulty, which becomes evident in problems like the proof of the irrationality of the square root of a prime, which proposition we may call 'SRP'.

D1 proof of SRP (437 seconds)

(1) Pp	p is a prime
(2) $TpKbKa$	$p.b^2=a^2$
(3) $\bar{D}xa \vee \bar{D}xb \vee x=1$	a and b are relatively prime
(4) $\neq p1$	p is not equal to 1
(5) $TxxKx$	$x.x=x^2$
(6) $\bar{T}xyz \vee Dxz$	if $x.y=z$, then x is a divisor of z
(7) $\bar{D}xy \vee TxFxyy$	if x is a divisor of y , then $x.Fxy=y$
(8) $\bar{D}xy \vee \bar{D}yz \vee Dxz$	transitivity of 'divisor'
(9) $\bar{T}xyz \vee \bar{T}zuv \vee \bar{T}xwv \vee Tuyw$	special form of cancellation theorem: if $x.y.u=x.w$, then $u.y=w$ (consequent is commuted form of $y.u=w$)
(10) $\bar{P}x \vee \bar{T}yzu \vee \bar{D}xu \vee Dxy \vee Dxz$	'first proposition of Euclid'
(11) $DpKa$	(2) & (6)
(14) $\bar{T}pyz \vee \bar{T}zuKa \vee TuyKb$	(2) & (9)
(15) $\bar{T}yzu \vee \bar{D}pu \vee Dpy \vee Dpz$	(1) & (10)
(17) $\bar{D}pa \vee \bar{D}pb$	(3) & (4)
(32) $\bar{T}pya \vee TayKb$	(5) & (14)
(35) $\bar{D}pKx \vee Dpx$	(5) & (15)
(48) Dpa	(11) & (35)
(57) $\bar{D}pb$	(17) & (48)
(58) $TpFpaa$	(7) & (48)
(60) $\bar{D}az \vee Dpz$	(8) & (48)
(95) $\bar{D}pKb$	(35) & (57)
(97) $TaFpaKb$	(32) & (58)
(145) $\bar{D}aKb$	(60) & (95)
(156) $DaKb$	(6) & (97)
(250) contradiction	(145) & (156)

In the above proof the first three clauses, which formulate the assumption that there exists a prime whose square root is rational, were taken as the set of support. Without the set of support strategy, D1 was unable to generate a proof of SRP within the storage limitations of the machine. Even with the aid of the set of support strategy, D2 was unable to generate a proof of SRP in 30 minutes, since it never got round to generating the essential clause (35), which is a consequence of (1), (5) and (10), and which states that if p is a divisor of x^2 then p is a divisor of x . An efficient proof requires that (35) be generated fairly early on, but D2 will not form any resolvents with (10), which is the longest clause in the set, until it has first exhausted the consequences of (1)-(9). Furthermore, when D2 starts to work on (10) it resolves it against the most recently generated unit clauses, but the generation of (35) requires that (10) be resolved against the original unit clauses. In other words, D2 (and PG1), which attacks the shortest clauses first, will not produce efficient proofs in cases (such as SRP) where efficiency demands that the longest clauses be attacked first, but D1, which attacks the short and the long clauses more or less equally, will occasionally produce moderately efficient proofs in such cases.

There remains to discuss one further type of example, that which requires the deduction of a contradiction from a set of non-unit clauses. Factoring, as in the case of W5, will sometimes suffice to generate the unit clauses essential to the operation of the programs described here. In other cases, it is necessary to produce unit clauses through resolution of non-unit clauses. This is a matter of finding or generating a pair of 2-clauses

$$C_1 \vee C_2, C_3 \vee C_4$$

that yield a resolvent

$$C_5 \vee C_6$$

that collapses into a unit clause, C_5 , upon deletion of one of the redundant literals as, for example, in the deduction of

$$PFFGy$$

from

$$PFx \vee Px, \bar{P}FGy \vee PFFGy.$$

PG1 makes provision for this sort of inference by the inclusion of a 'non-unit section', which takes over if the 'unit section' fails to generate a proof within the stated level bound. D1 and D2 also implicitly contain non-unit sections, in that they give priority to the n -clauses, which are the shortest non-unit clauses in the event that there are no unit clauses, and their descendants. D1 and D2, however, have no provision for passing to a non-unit section in the event that the unit clauses and their resolvents fail to produce a proof within given limits. An example of how a non-unit section can contribute to a proof is the following, which we may call 'W3', since it is a somewhat simplified version of the third example given by H. Wang (1964) in the appendix (Wang attributes the problem to Church).

D1 proof of W3 (271 seconds)

(1) $FPx \vee Fx$	premise
(2) $\bar{H}Px \vee Hx$	premise
(3) $\bar{H}Px \vee Gx$	premise
(4) $\bar{G}Px \vee Fx$	premise
(5) $\bar{F}Px \vee GPx \vee Hx$	premise
(6) $\bar{F}Qx \vee \bar{G}Qx \vee \bar{H}Qx$	premise
(7) $\bar{F}Px \vee HPx \vee \bar{G}x$	premise
(9) $\bar{H}PPx \vee Fx$	(3) & (4)
(19) $\bar{H}PQx \vee \bar{F}Qx \vee \bar{H}Qx$	(3) & (6)
(20) $Fx \vee \bar{F}Px \vee Hx$	(4) & (5)
(78) $Fx \vee Hx$	(1) & (20)
(89) $\bar{H}PQx \vee \bar{F}Qx$	(2) & (19)
(129) $FPx \vee Hx$	(2) & (78)
(131) $\bar{H}PQx \vee \bar{H}PPQx$	(9) & (89)
(145) $HPx \vee \bar{G}x$	(7) & (78)
(213) $\bar{H}PPQx$	(2) & (131)
(257) $Hx \vee GPx$	(5) & (129)
(269) $GPPPQx$	(213) & (257)
(272) $\bar{H}PPPPQx$	(2) & (213)
(273) $HPPPPQx$	(145) & (269)
(280) $\bar{H}PPPPQx$	(2) & (272)
(281) contradiction	(273) & (280)

Upon the generation of the unit clause (213), the value of n was lowered from 2 to 1, and the 'unit section' took over, producing a series of additional unit clauses that quickly resulted in a contradiction. In order to produce the above proof in a reasonable time, it was necessary to employ a special-purpose deletion heuristic, based on the assumption that a resolvent R of C_1 and C_2 , that is no shorter than the longer of C_1 and C_2 , is of use in a proof if and only if it assists in the generation of shorter clauses. After a reasonable time, therefore, such resolvents R may be deleted. In the above proof of W3, for example, the resolvents R that are no shorter than the longer of their parents are deleted after clauses shorter than R are produced. This entails the deletion of clauses (19) and (20), along with a great mass of other 3-clauses, after the 2-clauses (78) and (89) are generated. This deletion heuristic was tailored specially to fit W3, and there is no guarantee that it would work in very many cases, but some such heuristic is essential if W3 is to be proven in a reasonable time, since the combinatorial possibilities between the various clauses are very great and the amount of garbage produced therefore increases at a rapid rate.

W3 is another example for which the procedure of D1 is somewhat more efficient than that of D2, since D1 resolves the 2-clauses fairly immediately against the 3-clauses, thereby generating the important 3-clauses (19) and (20) at an early stage in the proof, while D2 (and PG1) must exhaust the consequences of the 2-clauses before proceeding to resolve the 2-clauses

against the 3-clauses. D2, moreover, cannot prove W3 at all without some sort of level bound restriction, since otherwise it will get into a loop within the 2-clauses and generate

$$\bar{H}PPx \vee Fx, \bar{H}PPPx \vee Fx, \bar{H}PPPPx \vee Fx, \dots$$

Since the results described in this paper were obtained, several new strategies for limiting the number of resolvents generated have been proposed, most notably J. A. Robinson's ' P_1 -deduction' (1965b) and B. Meltzer's ' P_2 -deduction' (1966) and 'sharpened set of support' (Meltzer & Poggi 1966), which are all based on Robinson's ' P_1 -deduction theorem', i.e., the theorem that the null clause can always be deduced from an unsatisfiable set C via a chain of resolutions R_1, R_2 , etc., such that one parent of each R_i is 'positive' in the sense of containing no negated literals. Meltzer has shown that the predicates of an unsatisfiable set C can be renamed so that the set of positive clauses is in some sense minimal, and that this can usually be done so that all the positive clauses appear as a subset T' of T , where T is the set of clauses that formulate the special hypotheses of the proof and the negation of the conclusion. One may then use T' as a set of support, in conjunction with the restriction that one parent of each resolvent must be positive. This is a 'sharpened' set of support strategy, since T' is usually a proper subset of T , and since the restriction that one parent of each resolvent must be positive rules out some resolvents that the ordinary set of support strategy would permit. Preliminary investigation shows that T' can often be reduced to just one clause, as in our earlier example C1 (minus the superfluous first clause), wherein replacing P by $\bar{P}$, $\bar{P}$ by P' , D by $\bar{D}'$, and $\bar{D}$ by D' leaves (2) as the only positive clause. Examples given by Meltzer show that the 'sharpened' set of support strategy is capable of generating short proofs for many theorems, though few data have so far been obtained on the relative efficiency of this strategy in comparison with the ordinary set of support strategy with unit preference. The former strategy does rule out some inferences involving unit clauses that the latter strategy permits, e.g.,

$$\bar{P}x, Px \vee \bar{Q}x, \therefore \bar{Q}x$$

and the 'non-unit section' of one's program will therefore become relatively more important under the sharpened set of support strategy.

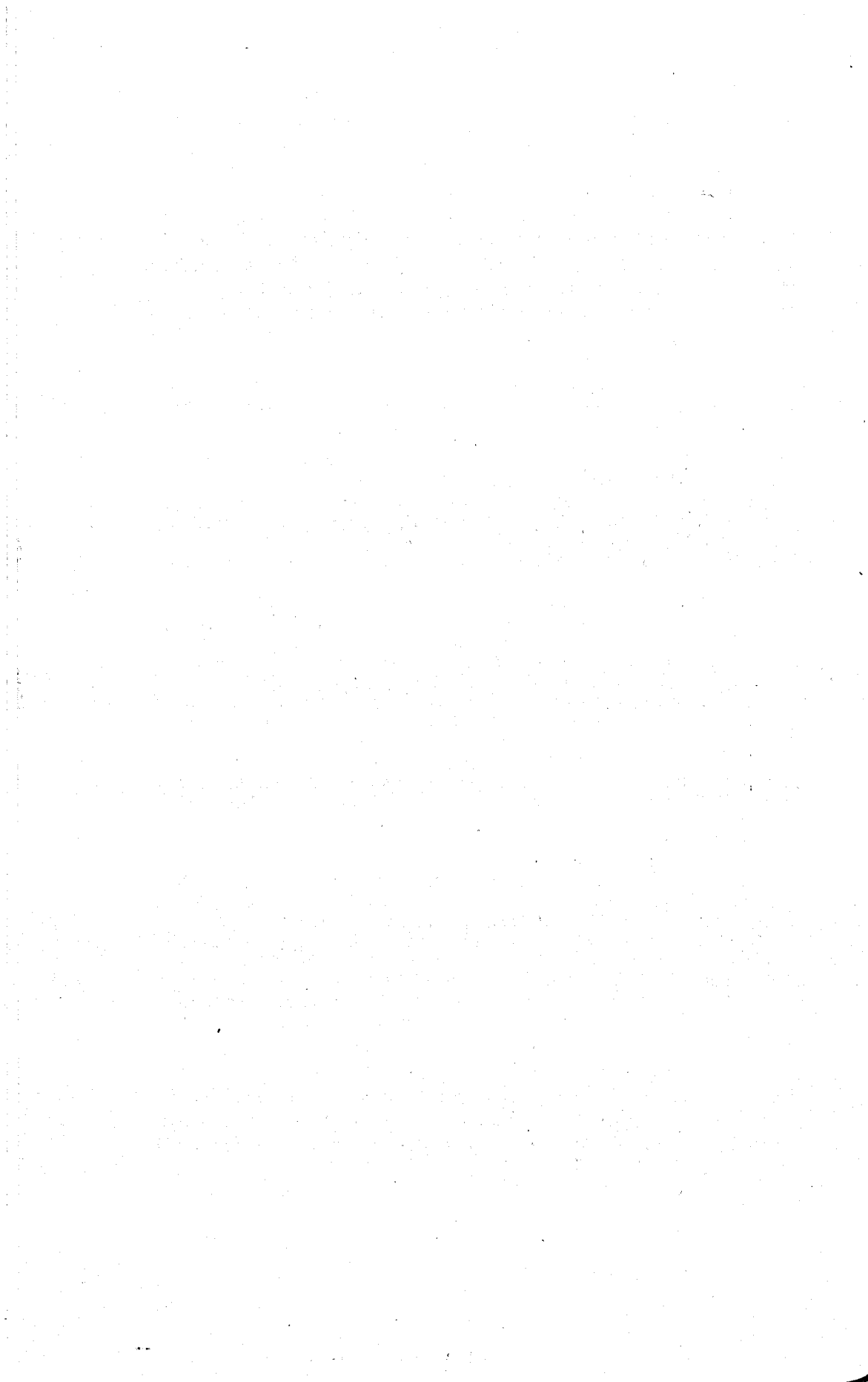
We have so far said nothing about the completeness of the methods discussed. D1 and D2 were not designed to be complete theorem-proving algorithms, but were intended rather as methods of producing efficient proofs for certain types of theorems. In order for them to be complete algorithms, it would be necessary to include an explicit 'non-unit section', as in PG1, general enough to provide for all cases in which the proof depends, in whole or in part, upon the generation of resolvents from pairs of non-unit clauses. The authors of PG1 have shown that their method is complete, so long as the level bound k is equal to or greater than the number of generations

$$S_0, S_1, S_2, \dots, S_i, \dots$$

that the original resolution algorithm would have to produce in order to obtain a proof, and so long as the set of support S is chosen in such a way that $C - S$ is satisfiable (i.e., such that the contradiction depends upon adding S to an otherwise consistent set of clauses). P_1 -deduction and its variants are also provably complete. There is no necessary connection, however, between efficiency and completeness, and if and when really significant mathematical theorems are proven mechanically, the proofs will probably be generated by special-purpose heuristics rather than by theoretically complete general methods.

REFERENCES

- Cook, S. A. (1965). Algebraic techniques and the mechanization of number theory, *RM-4319-PR*, RAND Corporation, Santa Monica, California.
- Davis, M. (1963). Eliminating the irrelevant from mechanical proofs, *Proceedings of Symposia in Applied Mathematics XV*, pp. 15-30. American Mathematical Society, Providence, Rhode Island.
- Gilmore, P. C. (1960). A proof method for quantification theory, *IBM J. Res. Devel.*, 4.
- Guard, J. R. (1964). Automated logic for semi-automated mathematics, *Contract No. AF19(628)-3250, Project No. 5632, Task No. 563205, Scientific Report No. 1*, Applied Logic Corporation, Princeton, New Jersey.
- Meltzer, B. (1966). Theorem-proving for computers: some results on resolution and renaming. *Comput. J.*, 8, 341-343.
- Meltzer, B., & Poggi, P. (1966). An improved complete strategy for theorem-proving by resolution, Internal report, Metamathematics Unit, University of Edinburgh.
- Prawitz, D., Prawitz, H., & Voghera, N. (1960). A mechanical proof procedure and its realization in an electronic computer. *J. Assn Comput. Mach.*, 7, 102-128.
- Robinson, J. A. (1965a). A machine-oriented logic based on the resolution principle, *J. Assn Comput. Mach.*, 12, 23-41.
- Robinson, J. A. (1965b). Automatic deduction with hyper-resolution. *Int. J. Computer Math.*, 1, 227-234.
- Wang, H. (1964). Toward mechanical mathematics, *IBM J. Res. Dev.*, 4. Also appears as Chapter IX in *A Survey of Mathematical Logic*. Peking: Science Press, and Amsterdam: North-Holland Publishing Company, 1964.
- Wos, L., Carson, D., & Robinson, G. (1964). The unit preference strategy in theorem proving, *AFIPS Conference Proceedings*, 26, 615-621. Washington, D.C.: Spartan Books.
- Wos, L., Robinson, G. A., & Carson, D. F. (1965). Efficiency and completeness of the set of support strategy in theorem proving, *J. Assn Comput. Mach.*, 12, 536-541.



MACHINE LEARNING AND HEURISTIC PROGRAMMING

6

AUTOMATIC DESCRIPTION AND RECOGNITION OF BOARD PATTERNS IN GO-MOKU

A. M. MURRAY and E. W. ELCOCK
UNIVERSITY OF ABERDEEN

INTRODUCTION

A series of computer programs have been written to play the board game Go-Moku. Go-Moku is played on a 19×19 square mesh. Player b(w) has a supply of black (white) pieces. The players take it in turns to play a piece on a mesh point. The winner is the first player to complete a *5-pattern*, that is, to make up a (horizontal, vertical or diagonal) line of five and only five adjacent pieces of his colour.

The programs carry out a backtrack analysis on games which have been lost to a human opponent. These analyses automatically generate a list of descriptions of subgoals, each hopefully describing the essential structure of a board pattern from which completion of a 5-pattern is inevitable. The playing capability of such a program depends critically on the power of the abstract descriptions and on the machinery for recognising realisations of described subgoals.

A distinct phase of the work was completed for the Machine Intelligence Workshop held under the auspices of the Experimental Programming Unit in Edinburgh in 1965. This phase included complete and working programs which allowed investigation and evaluation of a number of simple backtrack analysis routines. The learning programs resulting from this first exploratory phase automatically become competent players after a small number of games played against a human opponent. They reach a point, however, still far short of expert play, beyond which they cannot develop further. Their inadequacies did, however, highlight the problems that needed to be solved for more continuing self-improvement to be possible.

The paper by Elcock and Murray (1967) describing this work was published in the proceedings of the Workshop and provides useful background to the present paper.

The last year has been spent on exploring solutions of some of the problems highlighted by the earlier programs. A more sophisticated analysis routine which works in the context of a more powerful and general way of describing board patterns has been designed and implemented. The new programs are capable of analysing winning positions and, from the analysis, synthesising descriptions of subgoals of arbitrary complexity. An interesting method has been developed for automatically generating, from a description of a subgoal obtained by the analysis/synthesis process, a segment of control instructions which, when interpreted, direct the processing of the board required to recognise realisations of the described subgoal.

The present paper reports on the part of this work that is concerned with description and recognition.

DESCRIPTION OF BOARD PATTERNS

This section gives an informal discussion of the aspects of board patterns described and explains why the particular description scheme chosen is appropriate both to play of the game, and particularly to automatic acquisition of descriptions by program analysis of played games.

First, a few general remarks. A description says something about a more or less local pattern of played pieces on the board. It is *not* a representation: it does *not* attempt to say all that can be said about a particular local pattern in the sense that the particular pattern could be reconstructed from the description. The things a description does say express certain constraints (conditions) that a local pattern must satisfy in order to meet the description. The particular kinds of constraint that can be expressed are, hopefully, those that allow any particular board situation which contains a winning move to be described in a way which captures constructively the essential content, and only the essential content, which makes the win inevitable. Such a description, once acquired, should then be useable in game play to recognise any one of a class of board positions which in some local region meet the constraints of the description and which contain a recognisable winning move.

A description makes no reference to colour and can describe either a black object or a white object. In the text and diagrams that follow, white objects will be used throughout: corresponding black objects are obtained simply by replacing black by white and white by black.

Motivation of the descriptive scheme

The next few paragraphs try to expose, by means of examples, the motivation underlying the descriptive scheme implemented. In all diagrams only the relevant region of a board is given.

Example 1. In any of the board situations of Fig. 1 white play at node 1

completes a 5-pattern (and wins immediately). The essential content of a board situation which makes immediate completion of a 5-pattern possible is, trivially:

there exists a node which is a constituent of a possible 5-pattern, of which 4 pieces have already been played, on some one line through the node

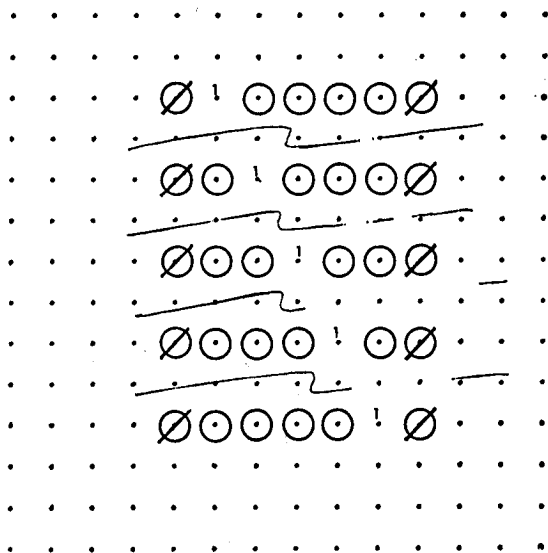


FIG. 1. $\emptyset$ not a white piece.

Example 2. In any of the board positions of Fig. 2 white play at node 1 has the consequence that a 5-pattern can be completed at the next white move whatever the intervening black move: if black plays node 1' (1'') then white plays 1'' (1').

The essential content of all these board situations is:

there exists a node which is a constituent of two possible 5-patterns, in each of which three pieces have already been played, on some one line through the node.

Attention is drawn to the fact that, as well as the node referred to explicitly, the object (as described by the text after 'there exists') contains, as it should, two implicit nodes (the nodes 1' and 1'' of the representations of the object in Fig. 2).

It should be noted that, if it is the opponent to play, he can destroy a realisation of the object by play at any of its explicit or implicit nodes.

Consider the board situation:

$$\emptyset \dots \alpha \emptyset \emptyset \emptyset \emptyset \dots \emptyset$$

The position contains two realisations of the object described above: one with α as the explicit node and one with β as the explicit node. These two

realisations have nodes in common (β is an implicit node for the realisation of which α is the explicit node . . .). With black to play he can destroy both realisations by playing at a common node.

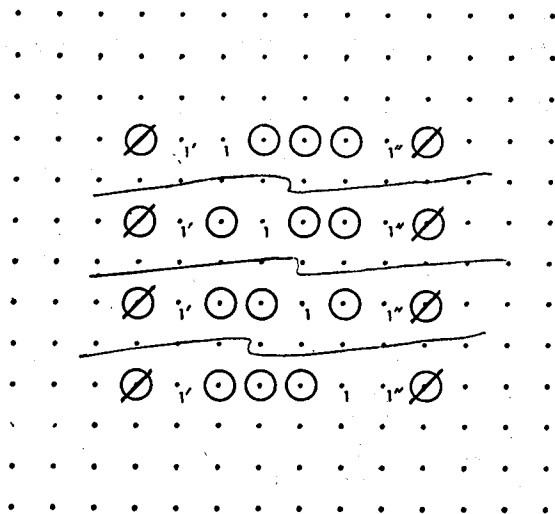


FIG. 2

Example 3. Fig. 3 shows a board position in which, after play at node 1, there are again two realisations of the object described in Example 2. This time, however, they do not share a common node; black cannot destroy both and white has a sequence of moves which inevitably lead to completion of a 5-pattern.

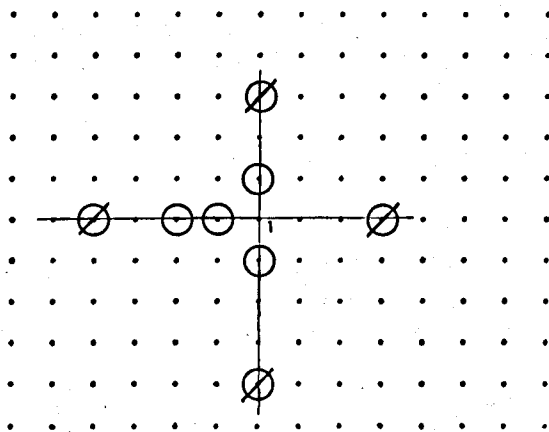


FIG. 3

The essential content of this (and similar) board positions is:
there exists a node which is a constituent of two possible 5-patterns, with two pieces played, on each of two lines through the node.

The description implies six other nodes; three on each line through the explicit node.

Play at the explicit node creates a set of realisations of the object described in Example 2 with no node common to all of them and which consequently cannot all be destroyed by the opponent's next move.

Example 4. Fig. 4 shows a board position in which, after play at node 1, there are two realisations of the object described in Example 1. A 5-pattern can always be completed in two player's moves.

The essential content of this class of board positions is:

there exists a node which is a constituent of a possible 5-pattern, with three pieces played, on each of two lines through the node.

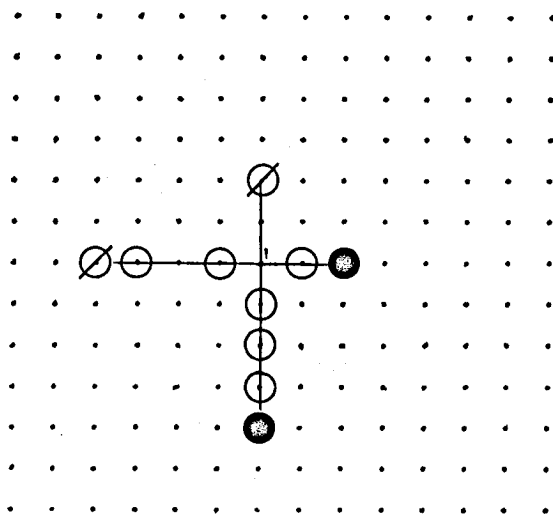


FIG. 4

Examples 5 and 6. So far the descriptions have been very simple: each has had just one explicit node and, in a realisation of the description, play at the explicit node initiates a sequence of moves leading inevitably to the creation of a 5-pattern.

Consider, however, the board positions of Figs. 5 and 6.

In the board position of Fig. 5, after play at node 1 a realisation of the objects described in Examples 2 and 3 is created at nodes 1' and 2 respectively. Similarly, play at node 2 creates realisations of the same objects at nodes 1 and 2' respectively.

The description (in words) of the essential content of this kind of board situation is rather lengthy:

there exists a node (1) which is a constituent of two possible 5-patterns on each of two lines through the node: on one of the lines through the node the patterns have two pieces played: on the other line the patterns

have one piece played and a common implicit node (2) of these patterns is itself a constituent of two possible 5-patterns, with two pieces played, on some other line through it.

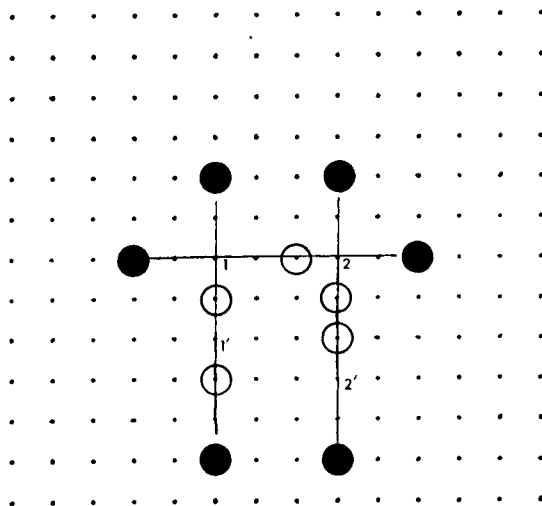


FIG. 5

The description makes explicit reference to two nodes. Nodes such as 1' and 2' of the realisation of diagram 5 are implied by the description.

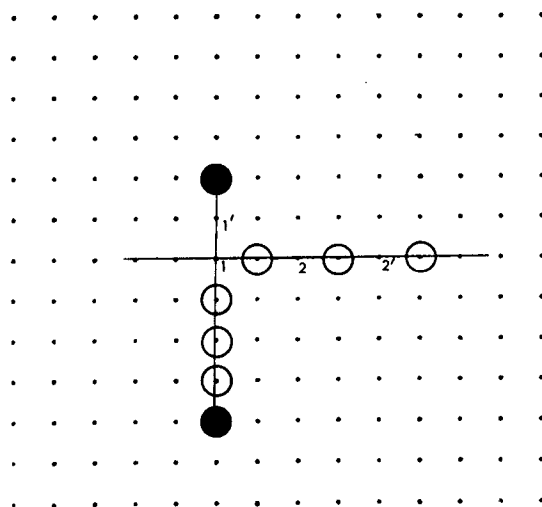


FIG. 6

In a realisation of this description the labelling of the explicit nodes 1 and 2 is arbitrary: the realisation has a certain symmetry. Play at either of the

explicit nodes of a realisation initiates a sequence of moves which leads to creation of a 5-pattern in at most four player's moves.

Contrast this with the board situation in Fig. 6.

After play at node 2, a realisation of the objects described in Examples 1 and 4 is created at nodes 2' and 1 respectively.

The essential content of this kind of board situation is:

there exists a node (1) which is a constituent of a possible 5-pattern on each of two lines through the node: on one of the lines the 5-pattern has three pieces played: on the other line the 5-pattern has two pieces played and an implicit node (2) of this pattern is itself a constituent of a possible 5-pattern on the same line, but with three pieces played.

The description again makes explicit reference to two nodes. This time, however, the labelling of the nodes 1 and 2 in a realisation is not arbitrary and, in fact, play at node 1 does not initiate a sequence of moves leading inevitably to the completion of a 5-pattern.

In the descriptive scheme implemented it is possible to add 'comments' to explicit nodes of a description. The 'comments' take the form of a statement of the expected number of moves required to complete a 5-pattern from play at a realisation of the object. As will be described later, these comments are made by program during the automatic acquisition of the description.

Automatic acquisition of descriptions

All the descriptions given in these examples imply that in a realisation of the description there exists a continuation which leads inevitably to the creation of a 5-pattern by the player. It is just the existence of this continuation that the description is designed to capture.

The described objects have this property because play at an appropriate node of a realisation creates a set of realisations of describable simpler objects, each of which has the property, and which do not share a common (implicit or explicit) node.

Descriptions with this property are called descriptions of 'subgoals'.

Most important is the implication of this property of the descriptive scheme for easy automatic acquisition of descriptions of subgoals by program.

Assume that the program can recognise realisations of described subgoals. Consider a stage in the program's game playing where the subgoals of Examples 2 and 3 have been described. Suppose the current state of the board in a game being played is that shown in Fig. 7. It is the opponent to play. At its last move the program did not recognise any realisations of described subgoals.

The opponent plays at node 1. The program now recognises the following realisations of known subgoal descriptions:

- (i) a realisation of Example 2 at *a* with implicit nodes *a'* and *b*;
- (ii) a realisation of Example 2 at *b* with implicit nodes *a* and *b'*;
- (iii) a realisation of Example 3 at *c* with implicit nodes *c'*, *c''*, *c'''*, *d'*, *d''*, *d'''*.

An analysis of the situation might go as follows:

The realisations (i), (ii) and (iii) do not share a common node and therefore the opponent can inevitably create a 5-pattern.

A minimal set of realisations with this property is (i) and (iii). The piece at 1 must be a common played piece of each member of this set.

From this analysis it is reasonably straightforward to synthesise a description of the subgoal of which there was a realisation one move back in the game.

It is necessary to identify, for each subgoal realisation of the minimal set, the relevant lines for which the piece at 1 is a played piece. It is then necessary to (i) downgrade (by effectively unplaying this piece) the components of the description that refer to these lines, and (ii) make node 1 an explicit node of

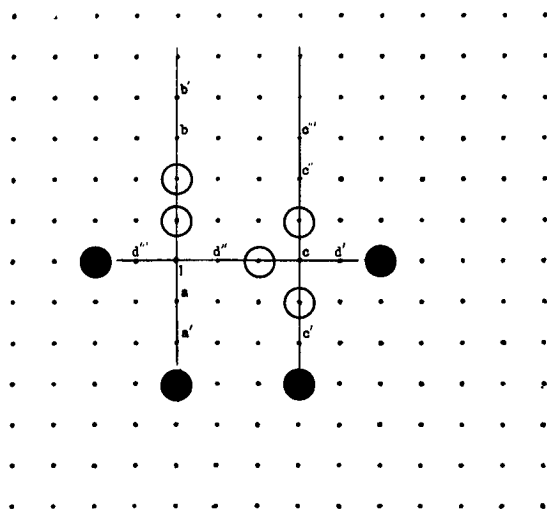


FIG. 7

these downgraded descriptions. A new description can now be synthesised from these downgraded descriptions by taking the union of them through the common node 1.

This description will describe a subgoal for which there is the realisation in the particular board situation from which the description was abstracted. Once the description is acquired, any of the whole set of its realisations can be recognised in future games.

It is not difficult to make this synthesis procedure precise and to automate it so that the program can, from game play such as the above, acquire descriptions of new subgoals. By the very nature of the process, this acquisition of descriptions of subgoals gives the program an increasing capability for directed play in board situations further and further away from a win.

RECOGNITION OF BOARD PATTERNS

The end result of being able to acquire descriptions of subgoals is to be able to recognise realisations of them in game play. This section discusses briefly

how this is done. Since one of the main interests here is the program implementation of the recognition process, it is necessary to introduce a notation for descriptions which is closer to the program level than the segments of English text given in the examples in 'motivation of the descriptive scheme'.

A formal notation for descriptions

As we have seen, a description specifies a number of explicit nodes and certain linear 5-pattern relationships between them. The nodes are labelled 1, 2, 3... A description consists of a 'node list' followed by a 'pattern list'.

The node list has an element for each explicit node (the 'parent' node of the element). The elements are listed in the serial order of the labels of their parent nodes. Thus

(1, 2, 3, 4-1, 3/4-1, 2/4-1, 2/3)

is the node list for a subgoal with 4 nodes, the elements of the node list being separated by '- '.

Each element on the node list is a 'line list' with up to four elements each referring to a different line through the parent node, elements of the line list being separated by ','. An element consists of the labels of those and only those explicit nodes which, in common with the parent node, contribute to a possible 5-pattern or patterns, specified in the pattern list, on the line. The label of the parent node itself is suppressed unless there would otherwise be a null entry.

Thus, the first element of the node list above states that node 1 is a node of specified (in the pattern list) patterns on four lines through it. On one line no other nodes are referenced. Nodes 2, 3 and 4 are referenced on one each of the other three specified lines. The second element states that node 2 is a node of specified patterns on two lines through it. On one line node 1 is referenced and on the other line nodes 3 and 4 are referenced. The third element, etc.

It should be emphasised that the node list specifies the line pattern intersections completely, i.e., none of the specified line patterns intersect in a node of any other specified pattern other than at the stated explicit nodes.

The pattern list has the same structure as the node list. There is an entry for each line element of the node list. The entry specifies the possible 5-pattern or patterns, of which the parent node is a constituent, that must exist on the line. The patterns are specified by an integer n ($0 \leq n \leq 5$) equal to the number of pieces in the pattern after play at the parent node. If the node must be a constituent of two patterns with the given n , then this is indicated by dashing the value of n (e.g. 4').

Consider the complete description:

(1, 2, 3, 4-1, 3/4-1, 2/4-1, 2/3)
3', 2', 2', 2'-2', 1'-2', 1'-2', 1'

Node 1 must be a constituent of two possible 5-patterns, in each of which two pieces have been played, on a line through it which references no other nodes. It must be a constituent of a 2' on each of the three other lines through it

passing through nodes 2, 3 and 4 respectively. Node 2 must be a constituent of a 2' on the line through it which passes through node 1, and a constituent of a 1' on another line through it which passes through nodes 3 and 4, etc. A realisation of the described object is given in Fig. 8.

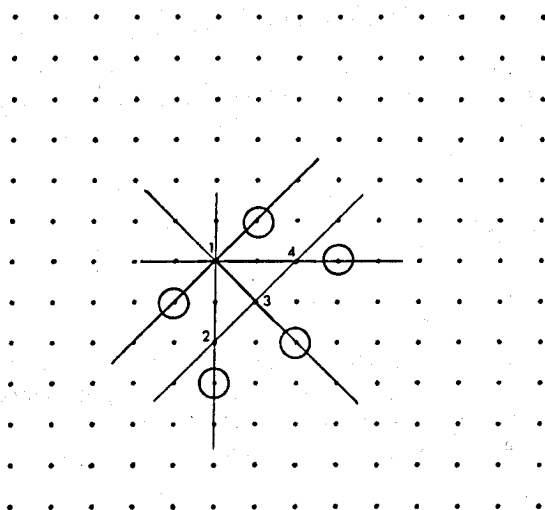


FIG. 8

Finally, comments may be added. As already mentioned (p. 81), the only comment for which provision is made in the present implementation is the prefixing of an element of the pattern list by a number equal to the expected number of player's moves needed to complete a 5-pattern in a realisation of the subgoal. Thus, the comment '6:' in

(1, 2, 3, 4-1, 3/4-1, 2/4-1, 2/3)
6: 3', 2', 2', 2'-2', 1'-2', 1'-2', 1'

for the subgoal of diagram 8.

A complex subgoal can be described in different equivalent ways corresponding to different labelling of nodes, etc. An arbitrary set of rules is used to ensure that descriptions of essentially the same object will always be given in a unique 'canonical' form.

In canonical form, the descriptions of the subgoals of the examples in 'Motivation of the descriptive scheme' are (note that subgoals having the same node list are grouped together):

(1)	
1:5	diagram 1
2:4'	diagram 2
(1, 1)	
2:4, 4	diagram 4

3:3', 3' diagram 3

·
(1, 2-1)
4, 3-3:4 diagram 6

·
(1, 2-1, 2)
4:3', 2'-4:2', 3' diagram 5

The recognition processor

What has been given above is essentially an external (written) notation for descriptions. The program uses an equivalent representation in terms of a list structure. The written notation, or its equivalent list representation, exposes the structure of the subgoal and is particularly appropriate for the manipulations involved in the synthesis of new descriptions. It does not, however, make clear in any operational sense the sequence of actions that must be gone through to recognise a realisation of the described object in any particular board situation.

Recognition is facilitated by the use of a quite different representation of the description called the 'control stream' representation. Each control stream is made up of a sequence of control elements. Each control element in effect specifies a board processing instruction. The complete sequence of elements specifies the total processing that has to be gone through to recognise a realisation of the described object or, more precisely, to determine whether or not a local region of the board contains a realisation of the described object with a particular board node identified with node 1 of the object.

In what follows an example of a control stream is discussed informally and without going into too much detail. The intention is simply to bring out the difference between the synthetic and operational descriptions of subgoals and to present the view that the data structure which is the control stream can be thought of as a set of instructions in a board processing language.

Fig. 9 shows again a realisation of the subgoal of Fig. 8 and gives the written description together with the equivalent control stream. The format of the control stream is a feature of a particular machine implementation and is irrelevant to the present discussion.

In processing a particular board situation for realisations of the subgoal a particular vacant mesh point on the board is assigned the label "1" (i.e., is associated with node 1 of the description), and the control scheme is scanned by an interpreter.

An 'A' element has the structure:

(A code; pattern list element).

The first control stream element of the example of Fig. 9 is interpreted as 'check that mesh point "1" is a constituent of possible patterns meeting the specifications 3', 2', 2' and 2' on different lines through the mesh point: if successful, continue the scan'.

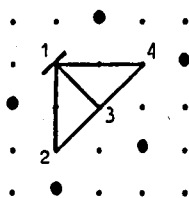
A 'B' element has the structure:

(B code; line pattern specification; node label; line label).

The second element of the example is interpreted as 'look for a line through the mesh point "1" for which the mesh point is a constituent of a pattern which meets the specification 3'; if successful, call this line "line 1 of node 1" and continue the scan'.

The third element of the example is interpreted as 'look for a line through the mesh point "1", not currently called "line 1 of node 1", for which the mesh point "1" is a constituent of a 2': if successful, call this line "line 2 of node 1" and continue the scan'.

(1,2,3,4-1,3/4-1,2/4-1,2/3)
6:3',2',2',2'-2',1'-2',1'-2',1'



A	3'	2'	2'	2'			
B					3'		1,1
B					2'		1,2
C	2'	1'			2'	2	1,2
B					2'		1,3
C	2'	1'			2'	3	1,3
D				1'	1'	3,2	2,2
B					2'		1,4
C	2'	1'			2'	4	1,4
D				1'		4,2	2,2
D						4,1	3,2
E					3'		1,1
E	2				3'		1,2
E	3				3'		1,3
E	4				3'		1,4
E	4	3			3'		2,2

FIG. 9

A 'C' element has the structure:

(C code; pattern list element; line pattern specification; node label; node label; line label).

The fourth element of the example is interpreted as 'look for a vacant mesh point on line 2 of node 1, lying within a possible 5-pattern of which mesh point 1 is a constituent, which is a constituent of possible patterns meeting the specifications 2' and 1' on different lines through the new mesh point and which in particular meets the specification 2' along line 2 of node 1: if successful, assign the label "2" to the mesh point and continue the scan'.

D-words are concerned with checking that already allocated nodes which are required to lie on a line are in fact on a line, and that the pattern specifications in the direction of this line are met.

E-words are concerned with finally checking that, for each line, the allocated nodes are in fact nodes of the same possible 5-pattern (the C-word allocations do not ensure this); checking the specification of this pattern, and listing implicit nodes (defensive moves). If the action of the last E-word is successful then a realisation of the subgoal has been recognised.

If at any stage it is impossible to continue the forward scan either because it is impossible to make the required allocation of a node or line, or after the last E-word to find other realisations with the same mesh point 1, the control

stream is backscanned by the interpreter to the first significant new allocation and the forward scan restarted.

CONTEXT AND SUMMARY

So far nothing has been said about how the control stream representation of subgoals is generated. In fact, there is a component of the total program which can transform the 'written' description, generated by the synthesis procedure, into the equivalent control stream.

With this in mind, Fig. 10 gives a schematic representation of the total program that has been written and of which some aspects of the pattern

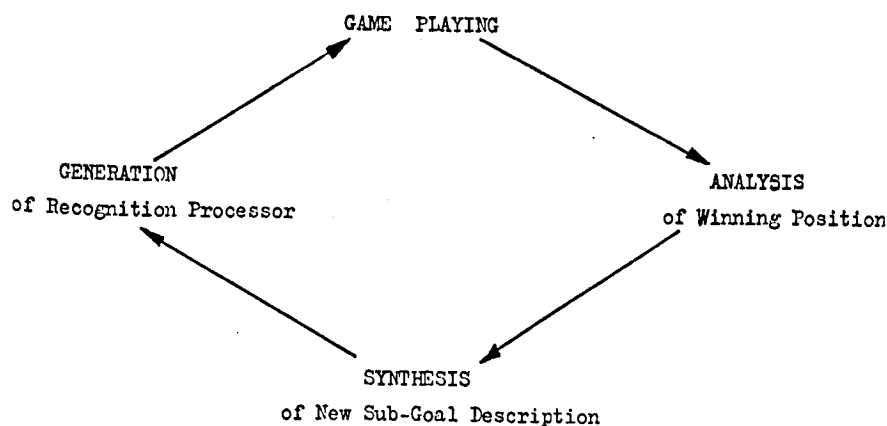


FIG. 10

description and recognition components have been discussed here. We would like to stress the view that the program does not generate data in the conventional sense, but rather segments of subgoal recognition program.

Finally, to counterbalance the simplicity of the examples treated on pages 83-86, Fig. 11 gives subgoal descriptions (together with graphic mnemonics) which present a typical list generated by program over a sequence of played games. It should perhaps be added that, with this list, the program plays at expert level.

REFERENCE

- Elcock, E. W., & Murray, A. M. (1967). Experiments with a learning component in a Go-Moku playing program. *Machine Intelligence 1*, N. L. Collins and D. Michie (eds.), pp. 87-103. Edinburgh: Oliver and Boyd.

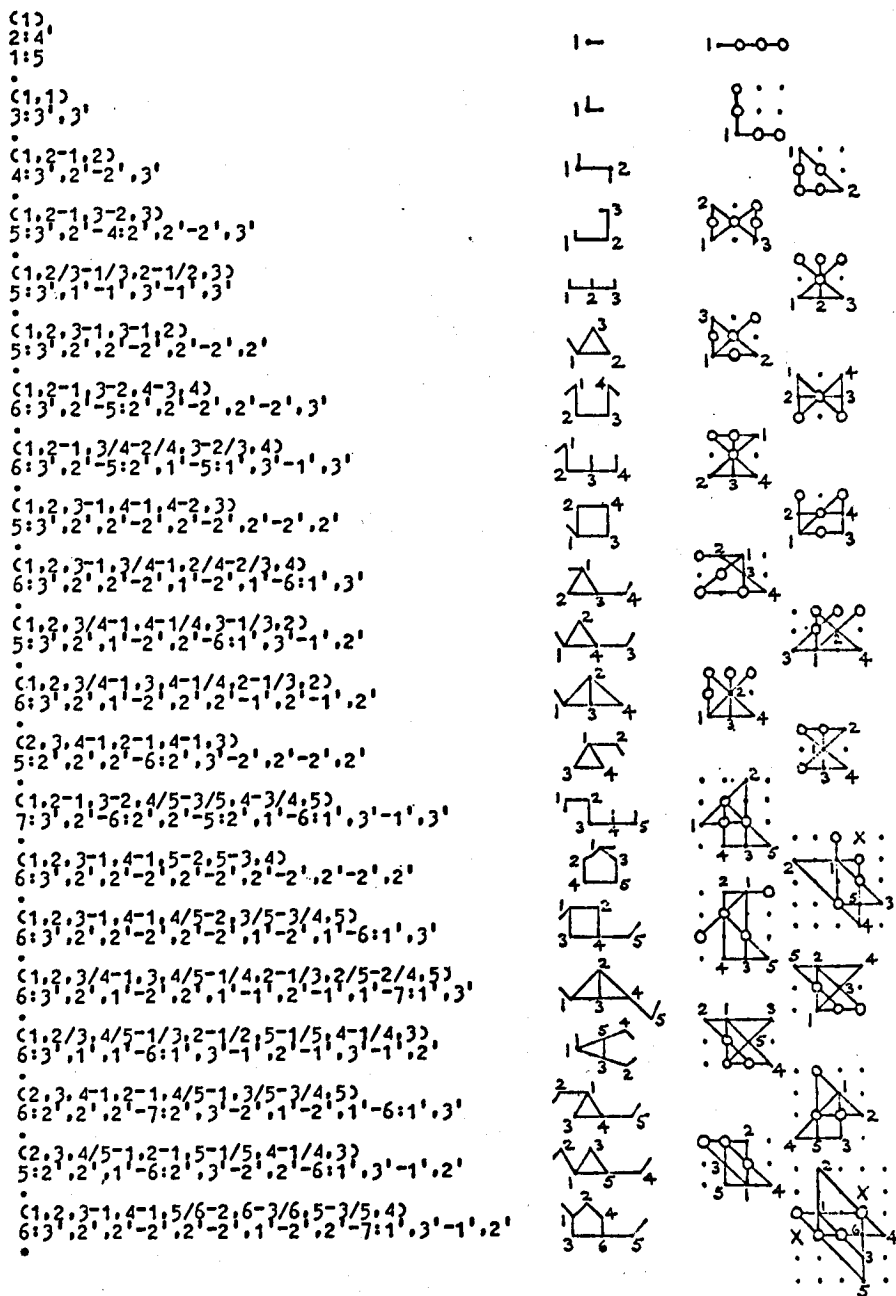


FIG. 11

7

A FIVE-YEAR PLAN FOR AUTOMATIC CHESS

I. J. GOOD*

TRINITY COLLEGE, OXFORD, and
ATLAS COMPUTER LABORATORY, CHILTON

JUSTIFICATION OF CHESS PROGRAMS

Young animals play games in order to prepare themselves for the business of serious living, without getting hurt in the training period. Game-playing on computers serves a similar function. It can teach us something about the structure of thought processes and the theory of struggle and has the advantage over economic modelling that the rules and objectives are clear-cut. If the machine wins tournaments it must be a good player.

The complexity and originality of a master chess player is perhaps greater than that of a professional economist. The chess player continually pits his wits against other players and the precision of the rules makes feasible a depth of thinking comparable to that in mathematics.

No program has yet been written that plays chess of even good amateur standard. A really good chess program would be a breakthrough in work on machine intelligence, and would be a great encouragement to workers in other parts of this field and to those who sponsor such work.

In criticism of the writing of a chess program, Macdonald (1950) quoted a remark to the effect that a machine for smoking tobacco could be built, but would serve no useful purpose. The irony is that smoking machines have since been built in order to help research on the medical effects of smoking. This does not prove that a chess program should be written, but suggests that the arguments against it might be shallow. Many branches of science, and of pure and applied mathematics, have started with a study of apparently frivolous things such as puzzles and games.

* Now at Virginia Polytechnic Institute, Blackburg.

It is pertinent to ask in what way a good chess program would take us beyond the draughts program of A. L. Samuel (see Appendix B). The answer is related to the much greater complication of chess, the much larger number of variations and possible positions. In fact, the number of possible chess positions is about the cube or fourth power of the number of possible draughts positions (see Appendix E). Samuel was able to make considerable use of the storage of thousands of positions that had occurred in the previous experience of the machine, and this led to a very useful increase in the depth of analysis of individual positions. The value of this device depends on the probability that, at any moment in the analysis, we run into a position that has already been analysed and stored. If the expected number of previously analysed and stored positions on an analysis tree exceeds unity, then the expected *effective* number of positions on the tree is infinite! This remark is of course based on an over-simplified model, as is clear since the total number of possible chess or draughts positions is finite. The remark is also somewhat misleading, since an 'infinite' tree might be deep at the wrong places; but it does give a little insight into the effect of storing positions. (See also (a) on p. 92 and Appendix H, p. 114.) Whereas it is useful to store thousands of positions in draughts, in chess it would be necessary to store millions or billions of positions in order to gain a comparable advantage. Presumably the main advantage of storing positions in draughts accrues from the opening and early middle-game; and in chess, by storing millions of positions, the machine could gain an opening advantage. Apart from this, I have believed for a very long time that a good chess program would need to make use of essentially the same methods as those used by men. The difficulty, the interest, and the challenge, is to formalise exactly what it is that men do when they play chess.

The more the program is based on the methods used by humans the more light it will shed on the nature of thought processes. But for the sake of a clear-cut objective I should like to write a program that wins games.

When a man plays chess, he does often recognise situations that have occurred before, but these situations are seldom complete positions, except in the opening and very late end-game; rather they are features of the position, or patterns embedded in it. Thus chess provides an example of pattern recognition, whereas draughts does not to much extent. Nevertheless, several of the techniques used by Samuel should be incorporated.

Chess also provides a better example than draughts of the use of associative memory, since any given position is associated with many situations that have occurred in the chess player's experience. These associated situations suggest strategic or tactical ideas to the player, the strength of the suggestions being dependent on the strength of the associations.

An example of a pattern is when White's king has three unmoved pawns in front of it in the late middle-game or early end-game. This pattern is recognised by every experienced chess player as potentially dangerous, in that there is the possibility of a mate by a rook or queen on the back line,

even in conjunction with a sacrifice. *One* analysis of the position will then be performed with this theme in mind. This applies more generally to the goals and subgoals that occur to the chess player. Thus there should be 'specifically-directed' as well as 'routinely-directed' analysis.

Another important aspect of chess thinking, also required in most other problem-solving, is what de Groot (1946, 1965) calls 'progressive deepening' of an analysis. Typically an analysis of a position by a human player does not simply follow a tree formation, but contains cycles in which a piece of analysis is retraced and improved. (Compare the not purely hierarchical structure of definitions and perception discussed by Good 1962a, p. 124, and 1965a, pp. 42 and 75.) Also a player continually formulates subgoals and subsubgoals and continually modifies them. In these respects again chess provides a better example than draughts.

As every philosopher and programmer knows, it is not easy to formalise human thought. In fact, when we say that a man has made a judgement, we imply that we do not know in detail how he did it, and this is so even if that man is ourselves (Good 1959). This aspect of judgements is not yet mentioned in dictionaries. The work on automatic chess therefore started when men, such as the musician-chess player Philidor (1749), formulated general principles of play. Many of these principles have become embodied in chess slogans or clichés that need to be taken into account by the programmer (see Section 4.2).

Many of the principles of chess are expressed linguistically rather than numerically. This is much more typical of chess, and of many intellectual activities in ordinary life, than in draughts. Thus chess provides a good field, with a limited vocabulary, for those who are working on language handling by computers. (An example that did occur in draughts, or should have occurred, is mentioned in Appendix B, and curiously enough was overlooked by the strong draughts player who was pitted against a machine. He could have drawn the game, and the method can be readily expressed in words, but only artificially in numerical terms.)

Chess programming can also be justified as sport. £500,000,000 are spent on football pools in the United Kingdom in five years, and each pound must correspond to several hours of study. Further, hundreds of millions of man-hours are expended in watching the game. The expenditure on chess programming will be microscopic in comparison.

The nature of complex games

There is one principle of chess that is so old that the early theoreticians probably did not bother to mention it. It is certainly as old as the game of Go, at least 3000 years, and is typical of all complex games, although stochastic games have additional features. It is the habit of thinking 'if I go there, then he might go there or there, and, in the first case, I might go there, etc.'. In other words, analysis often follows the branches of a tree, with evaluations at the endpoints, followed by back-tracking or iterative minimaxing (see

Appendix A). This process corresponds closely to the formulation of games in 'extensive form' (see von Neumann & Morgenstern 1947, or Luce & Raiffa 1957, Chapter 3). The notion of a game tree applies of course much more generally than to chess. With regard to the definition of the endpoints, it is sometimes said that they should be quiescent positions, but, provided that the outcome of the game is clear enough, an endpoint can also rationally be a turbulent position. For this reason, it is convenient to introduce a notion called 'agitation', which is a modification of 'turbulence' taking into account the probability that the result is clear-cut. (See Appendix H for a discussion of the terminology and its relationship to decision making in general.)

Let us suppose that we can measure, for any position π , both its *turbulence* and also the *superficial probabilities* that the player can win, draw, or lose, p_W , p_D , and p_L . (By a 'superficial probability' I mean a probability based on some evaluation function, without any forward analysis. Perhaps 'surface probability' would be a better term.) Then the decision of whether to regard that position, π , as an endpoint, when analysing a position, π_0 , depends on the following considerations:

(a) The *depth* (i.e., the number of moves ahead) of the position π from the current position, π_0 . More precisely, the decision depends on the probability that we shall reach π from π_0 . The smaller this probability, $P(\pi|\pi_0)$, or, less accurately, the greater the depth, the more rational it is to treat π as an endpoint of the analysis tree for π_0 . The 'probabilistic depth' of π from π_0 could reasonably be defined as proportional to $-\log P(\pi|\pi_0)$, and the expected amount of information in an analysis, or its effective depth could be defined as $-\sum_{\pi} P(\pi|\pi_0) \log P(\pi|\pi_0)$ summed over all the endpoints of the tree. This is an incomplete entropy, since, usually, $\sum_{\pi} P(\pi|\pi_0) < 1$. This definition suggests itself, but I think the analysis given in Appendix H will turn out to be more useful. The value of storing previously analysed positions could perhaps be estimated in terms of 'effective depth'.

(b) The lower the turbulence of π , the more prepared we should be to treat it as an endpoint, other things being equal.

(c) The more obviously the outcome of the game is decided at π , the more prepared we should be to treat π as an endpoint, other things being equal.

(d) The larger the analysis tree as a whole, as far as we can judge, the more prepared we should be to treat π as an endpoint, other things being equal.

(e) The less time we have left on our clock, the more we should be prepared to treat π as an endpoint, other things being equal.

(f) The less time our opponent has left on his clock, the more prepared we should be to treat π as an endpoint, other things being equal. (This advice is double-edged!)

In other words, we should define a certain function of six variables, monotonic in each separately, and should treat π as an endpoint if this function exceeds some threshold.

Notice the following philosophical point. The superficial probabilities are not strict logical probabilities, since they are based on a suppression of a

logical argument, namely a forward analysis. (Compare the discussion of the probability of mathematical propositions in Good 1950a, p. 49.) Since we are forced to make use of superficial probabilities, there is bound to be an element of luck in chess. (There has even been a competition on luck in chess: see Golombek 1966.) From the point of view of strict logic, chess is a game of pure skill, but it is a physical impossibility to avoid some degree of chance. Just as in mathematics we must make use of probabilities based on incomplete analysis. When trying to prove a mathematical theorem, it is necessary to formulate subgoals, and to estimate superficial probabilities for them, in order to find an appropriate strategy. Theorem-proving resembles chess playing in that we have an objective and an analysis tree, or graph, but differs in that a superficial expected pay-off replaces the iterated minimax. The minimax idea can come in if we are trying to prove a theorem and we imagine that we have an opponent who wishes to disprove it. The value of our game is 1 if the theorem is true and -1 if it is false. In the proof trees described in the paper by Dr D. C. Cooper the 'and's correspond to moves of the opponent, since we must allow for both branches, whereas the 'or's correspond to our own moves. The minimax (strictly maximin) value of the tree tells us whether the theorem is true, and, if we allow for superficial probabilities at the end-points of the tree, the minimax value is the superficial probability of the theorem.

The programming of complex games will also exemplify an aspect of practical decisions, since here too it is typical that we are forced to make use of superficial probabilities.

Closely allied to the problem of selecting endpoints on an analysis tree, is the selection of which moves to discard without analysis. This problem is discussed in Appendix H.

SOME POSSIBLE KINDS OF CHESS PROGRAMS

2.1. Chess is a game of perfect information, and is a finite game if the 50-move drawing rule is mandatory (see Appendix D). Hence, from the point of view of the Borel-von Neumann theory of games, chess is 'trivial' (see Appendix A). This shows that, for practical chess, the Borel-von Neumann theory has trivial relevance. But it would be a good exercise to program a computer to do an exhaustive analysis of any position given to it. Such a program could be used for solving two-movers, a program for which has already been written at M.I.T., but it would have little applicability to normal chess. It could be largely written independently of the rules of chess, and could invoke these rules as a sub-routine. It would then have applications to other games and decision problems. If this program is written it should have the option of being stopped at any assigned depth.

A program for solving ordinary 'two-movers' would be useful when dealing with a large number of problems, for a tournament or book, or for a writer of chess columns in newspapers. Professor R. C. O. Matthews points out (private communication) that it could be used in the composition of problems.

2.2. If our object is to learn as much as possible about learning, we could try to program a machine to learn the game without even telling it the rules. This is how Capablanca learned chess: by watching his father and uncle playing. Incidentally he won the first game he played, at the age of 6, against his uncle. McCulloch (1965, p. 199) calls a machine of this kind an 'ethical' machine. It would need to be able to formulate a very wide class of hypotheses.

2.3. Next, we could give the rules to the machine and let it learn the principles for itself. Here again the machine would need to be able to formulate a wide class of hypotheses, but not as wide as the 'ethical' machine.

2.4. Next, we could give the machine a limited class of principles and allow it to work out its own evaluation functions in terms of these principles, in the light of its experience. The experience could either be the playing of games or, more efficiently, we could tell the machine which moves in various positions were good or bad, and how good or bad. The simplest form of evaluation function would be linear, with coefficients that were allowed to be positive, negative or zero. The optimisation of these coefficients constitutes a primitive form of learning, but I think most people would be reluctant to call it 'concept formation'. If however non-linear polynomials are permitted, or equivalently if logical combinations of the principles can be formed and incorporated into the linear function, then concept formation becomes possible. For example, a machine might have discovered that two bishops are usually better than bishop and knight, if it had been programmed to examine quadratic evaluation functions (Good 1959). After this had been established, it might have gone on to modify the concept by saying that it was true in open positions, but not in closed positions. It might have discovered the whole of this principle in one step if it had been allowed to use cubic evaluation functions, but this would be a more expensive way of making the discovery. It seems reasonable to say that *the use of quadratic evaluation functions provides the first step in automatic creativity or concept formation*.

Humans usually think this way. First they give relative weights to features, then they look for interactions between pairs of these features. When they find a pair of features that need to be taken together, they define this as a new feature. Thus high-order interactions are often discovered piecemeal, but of course this is not always possible.

The discovery of more intricate concepts will require the generation of a large number of propositions, and the order in which they should be generated provides a most interesting search problem.

2.5. We could try to write the program with man-machine synergy in mind from the start (regarding the terminology, see Appendix K). We would aim to teach the machine how to play chess, and also to learn something about chess from the machine. Our long-term aim would be to cooperate with the machine in games against grand-masters, against other machines, and against other synergistic combines. A simple and effective example of man-machine

synergy is provided by R. C. O. Matthews' suggestion mentioned in section 2.1.

2.6. We could go all out to make the machine a good chess player, by putting as much as we can into the low-level program, and without relying on more machine learning than is implied in the optimisation of linear forms.

2.7. Same as 2.6, but with the machine capable of forming new concepts, that is, allowed at least to use non-linear evaluation functions.

Of these various aims, which are not mutually exclusive, the one under heading 2.2 is the most exciting, but 2.6 would be easier. I think a reasonable approach is to aim at 2.6 at first, in the hope of getting a good chess program, but always holding the other aims in mind.

A BRIEF HISTORY OF CHESS PROGRAMS

As mentioned earlier, the formalising of principles for playing chess began with the early theoreticians. But they wrote exclusively for people, just as grammarians wrote for people and not for machines until the advent of computational linguistics. Emanuel Lasker (1932, p. 339) said 'It is easy to mould the theory of Steinitz into mathematical symbols by inventing a kind of chess, the rules and regulations of which are themselves expressed by mathematical symbols. The Japanese game of Go is very nearly what I mean. In such a game the question, whether thorough analysis would confirm the theory of Steinitz or not, presumably could be quickly solved because the power of modern mathematics is exceedingly great.' What Lasker had in mind was clearly not a mathematical method of playing chess, but a mathematical demonstration of the validity of strategic principles for wider applications. Incidentally, I believe that a good Go-playing program would be much more difficult to write than a good chess program, because Go, not to be confused with Go Moku, depends so much on judgement. Perhaps Lasker was misled by the charming simplicity of the rules of Go (see Good 1965b).

In about 1932, one of my schoolboy chess associates named G. T. Hammond pointed out to me, without claiming originality, that there is a simple mechanical method of playing that is not as bad as one would expect, at any rate in the opening. The method is to play the move that maximises one's *mobility*, where mobility is defined as the number of possible legal moves. I cannot remember whether he allowed for the mobility of the opponent. We realised that this rule should not be used if it involved putting a piece *en prise*, and that the rule was not much good in a combinative position. But this observation about mobility is I think the most surprising single fact, so far, about automatic chess. It might date back much further than 1932, since the theoretical valuation of the pieces, of 1876, was based on the average numbers of squares controlled (see Appendix F). Compare also a remark attributed to Emanuel Lasker by Edward Lasker (1951, p. 56), '... the player who gains control of more than 32 squares has the better winning chance'.

Good (1939) remarked that it might be possible to translate into a mathe-

mathematical technique the knack that masters have of picking out only those lines that matter from the millions of possible ones. This is of course the central problem in writing a good chess program. (When that note was written, what is now called 'programming' would have been called 'mathematics'.) Another point in that little article was that some statistical reasoning could shed light on the nature of chess. For example, White's advantage in moves is $1 - 1 + 1 - \dots$, which is summable ($C, 1$) to $\frac{1}{2}$. In other words, White is, on the average, half a move ahead. Since $2\frac{1}{2}$ moves are worth a pawn, it follows that White's advantage is about a fifth of a pawn. Now, by statistical sampling we could quantify what I called the law of multiplication of advantage, and, from this, we might be able to shed light on the question of whether White has a theoretically forced win.

In 1940, when I met and worked with Turing, we had several discussions on how chess playing could be formalised. We also later had discussions with Shaun Wylie and Donald Michie. Essentially similar ideas were published by Shannon (1950), who had been thinking independently about chess programming. Turing, Michie and Wylie, in 1947-48, designed some one-move analysers, a discussion of which was published by Maynard Smith and Michie (1961) and Michie (1966).

In about 1941, I had discussions with J. Gillis on the possibility of founding a theory of imperfect logic, in which propositions would be exponentially forgotten if not used for some time, and also could be reinforced (the aim was to formalise human logic). A similar idea has been used very effectively by Samuel in his draughts-playing program.

Perhaps the most intensive work ever done on the formalising of chess thinking was that of de Groot. He studied the psychology of chess playing for over twenty years (see de Groot 1946, 1965). I have not yet been able to judge how much of the later literature is explicit in de Groot's writings. His basic experimental method was to record the spoken moment-to-moment thoughts of chess masters.

Shannon (1950) made the following points:

- (a) Different routines should be used for opening, middle-game and ending.
- (b) Forceful variations should be analysed out as far as possible and evaluated only at points of some stability.
- (c) Select the variations to be explored by some process so that the machine does not waste its time on totally pointless variations. (This point was covered by Good 1939.)
- (d) The threshold for deciding whether a position should be regarded as stable should depend on the number of moves advanced from the current position.
- (e) A statistical element can be introduced in the selection of moves so that an opponent who has won once cannot win again by exactly the same sequence of moves.
- (f) A few hundred openings could be stored and played by rote.
- (g) A learning program could be based on the use of a higher-level program

that changes the terms and coefficients in the evaluation function, according to the results of games.

Slater (1950) argued that 'it does seem possible that a chess computer which was programmed, beyond immediate tactical tasks, to maximise the mobility difference between itself and its opponent over a series of moves, might play a strategically tolerable game of chess'. His thesis was based in part on an examination of master games. (I do not know whether G. T. Hammond's remark was so based.) He also gave some statistics relevant to the law of multiplication of advantage.

Good (1950b) pointed out that 'the analysis of forceful variations in any position would be like a stochastic branching process, and the endpoints of the corresponding tree would be quiescent positions needing strategic evaluation (but see Appendix H). The total time taken to cope with such a tree would be roughly proportional to the number of individuals in this tree, and it is known that the probability distribution of this number is extremely skew (see Good 1949). . . . There would therefore be a danger of the machine running into time trouble. It would then modify its definition of the forcefulness threshold of a move.'

Good (1959, based on a lecture of January 1958) pointed out that:

(a) The training of a machine should be move by move, and not merely by the results of games.

(b) In quiescent positions it should be possible to find some rule for estimating the probability of a win, draw or loss, and this is analogous to the estimation of the probability of a mathematical theorem. 'It may be sufficient to assume that the log-odds are linear in all the types of advantage that are worth listing. . . . The probabilities of winning, drawing and losing can perhaps be expressed in the form $(p^2, 2pq, q^2)$, and the above log-odds mean $\log(p^2/q^2)$. Judging by some statistics of E. T. O. Slater, p^2/q^2 is approximately 1.4 for the initial position in master chess. I believe that White's advantage is worth half a move or a fifth of a centre pawn, so a pawn is worth a factor of $1.4^5 = 5$, and a queen $5^9 = 5,000,000$.'

(c) More convincing than the determination of the coefficients of a linear form, as an example of judgement, would be the determination of a functional form. The example of two bishops was mentioned.

(d) Very few human chess players discover new strategic principles for themselves. If we ask this of a machine we are asking for a level of creativity well above the average for humans.

(e) One advantage of an evaluation function expressed in terms of probabilities is that it could be worked in with estimates of the probabilities of the opponent's making various moves in any given position.

In 1956 a program was written at Los Alamos (Kister *et al.* 1957), capable of occasionally beating a beginner. The game was on a 6×6 board, and the analysis was exhaustive to two moves on each side. The evaluation was the simplest possible: a sum of material and mobility measures.

Bernstein *et al.* (1958a, b) wrote a program for the full 8×8 board, also

analysing two moves ahead, but with considerable move selection. As Newell, Shaw and Simon (1958) point out, this selectivity leads to a great increase in the complexity of the program, and so to a great slowing down *per move within* an analysis but that, in a deep analysis, this slowing down is far more than compensated because the number of variations to depth d goes up exponentially with d . If the average of the number of moves to be tried per position is *appreciably* decreased then the size of a deep tree is *enormously* decreased.

Newell, Shaw and Simon (1963, p. 47) mention *en passant* a hand simulation by F. Mosteller, and a Russian program for the BESM, but had little information about these two programs. Their own program avoids the use of a single evaluation on the grounds that humans do not seem to use one. They are especially concerned with the simulation of human thought, and they believe, as I do, that for a good chess program it is necessary to be so concerned. A number of goals are set up, and there is a move generator for each goal. Thus the evaluation function can be thought of as a vector. The goals that are listed are king safety, material balance, centre control, development, king-side attack, and promotion. Centre control and material balance are discussed in detail. Although the evaluation functions are not scalars, it is necessary to decide on an ordering for them in order that iterative minimaxing should be possible. One ordering suggested is lexicographic, the material balance being taken as the first component of the evaluation vector. It seems to me that any strict ordering of evaluation vectors effectively converts them into scalars. In lexicographic ordering, the components of a vector correspond to the digits of a scalar expressed relative to a large radix. The scheme is thus economical for the same reason that variable-length arithmetic sometimes is.

A recent reference is Newell and Simon (1965).

OUTLINE OF A FIVE-YEAR PLAN

The suggested plan can be split up into a number of parts, which I have listed here very roughly in the order in which they would be carried out. But this plan would be subject to revision from time to time. 'Progressive deepening' applies here just as much as to the analysis of a chess position! Note that the plan has not yet reached the software or hardware; it is in the form of so-called 'supporting underwear'.

Five years seems a reasonable estimate for the project for a team of about three people working full-time. In the following breakdown of the work, I have not yet tried to allot these fifteen man-years.

4.1 Program a computer to play legal chess. This has of course been done several times before. It is not as easy as it sounds. A 'position' must be defined to include the information of whose move it is; whether the kings have moved, and, if not, whether the rooks have; whether $P \times P$ *en passant* is legal; how many times the position has occurred before with the same player to move (where here 'position' has its naive interpretation, and presumably no distinction is made between, for example, White's KR and QR); and how

many moves have been played since the last capture or pawn move. A move that gives check must be so described. A distinction must be made between the making of a move and its mere consideration. The legal-chess program should be efficient, since the speed of the *whole* chess program will be proportional to the speed of this one.

The lists that are worth keeping are those that are *cheap to up-date*. These should include a list of all squares occupied by White and by Black. Then, when we are considering White's moves, we do not need to run right through the 64 squares of the board to find the white men. (For the purpose of the whole program, for playing good chess, not merely legal, we should have lists of squares dominated by the two players.) Whether we are in check, and whether it is a double check, should be recorded from the opponent's previous move. If, for example, we are in double check, we know that we must move our king, in virtue of a theorem that happens to be true for chess although it would not necessarily be true if the rules were slightly different. If we are not in double check, then lists of our pinned pieces should be made before looking for legal moves, together with the *rays* along which they are pinned, and also whether each pin is *relative* or *absolute*, i.e., whether the ray is one along which the piece can or cannot move. There should be a *transient* list of our own pieces, from which we cross off each piece after we have listed all the legal moves we have made with it from the given position.

4.2. Collect chess principles from good books on chess, for example, Lasker (1932), Fine (1941), Horowitz and Mott-Smith (1960), Euwe and Kramer (1964-65), Suetin (1965), and de Groot (1965); by introspection and memory; and by dictating your chess thinking to a secretary or dictating machine. Each principle constitutes a potential goal. Here is a preliminary list.

Analyse 'forced' variations either to quiescence or to a point where the result of the game is obvious. A forcing move is one that gives little option to the opponent on his next move, and includes all checks. There are degrees of forcefulness, and the threshold on the forcefulness will vary according to the depth of the analysis from the current position. More precisely it should allow for the discussion of Appendix H concerning the meaning of an 'endpoint' in an analysis tree.

A special case of the above principle is the following deep thought: if you can mate on the move, do so.* Also, never miss a check, it might be mate,—a motto attributed in jest to the Hampstead Chess Club by J. F. O'Donovan. More generally, consider those moves that restrict the opponent's choice on the next move, even if this involves a sacrifice; but don't stalemate him except on purpose.

Material values: one conventional set is $P=1$, $N=B=3$, $R=5$, $Q=9$, but

* This could be extended: if you can mate in two moves, do so. For a complete analysis as far as your second move would be cheap on a large computer. In some positions an exhaustive analysis to n moves deep, for $n > 2$, could be carried out by calling the subprogram mentioned in Section 2.1.

$B+B=7$. In particular, 'sacrifice' your queen if you get value for it. The advancing of a passed pawn can be regarded as the obtaining of extra material. For on the eighth rank a pawn is worth a queen. My own rule is that a passed pawn that cannot clearly be stopped is usually worth about five pawns on the seventh rank, three on the sixth, and one and a half on the fifth. A pair of double pawns = $1\frac{1}{2}$. 'The solution to every end-game is "pawn on"'—Doog. Swap pieces, but not pawns, when ahead in material, but don't let the position get completely blocked. See also Appendix F for values of pieces and squares.

In the opening, pawns can be arranged in decreasing order of value, according as they are Q or K pawns (central pawns), B pawns, N pawns, or R pawns. Lasker (1932, p. 107) gives the ratios 8: 6: 5: 2, but I should like to see a proper statistical survey of games to bear this out. Backward or isolated pawns are usually weak. If your opponent has them, blockade them with pieces. Two pawns, side by side, are strong because they cannot be blockaded. Try to fix a group of your opponent's pawns with a smaller number of your own. Avoid putting pawns on the same coloured squares as your lone bishop, especially if these pawns can be blockaded.

Increase your mobility, as compared with your opponent's. As discussed earlier, this includes more than is immediately apparent. To some degree it covers, for example, the advice to control space, and especially the centre, to get the pieces developed, and to move rooks to open and half-open files.

Attack the bases of your opponent's pawn chains, and defend the bases of your own.

Don't open up the position when you are behind in development.

Prevent your opponent from doing what you would do in his position. Decrease his mobility by advancing pawns. Do as you wouldn't be done by.

Exert pressure on the squares around the opponent's king.

The best strategy against a pawn storm on the wing is usually a thrust in the centre. If the centre is blocked and the kings are castled on opposite sides, then a pawn storm against the opponent's king is indicated. But if the pawns protecting the king are side by side, it is advisable to force one of them to advance before making the pawn storm.

When an attack fails, the side that was attacking is liable to be badly placed for the defence (as in Go), so don't attack without adequate reason.

Strengthen the weakest link in your position when defending, and attack the weakest link in your opponent's position when attacking. Move piece in worst plight. Exchange aggressive pieces when defending. Look for counter-attacks.

Preserve options, for example, develop at least one knight before both bishops in the opening, since the knights usually have fewer squares to go to.

Get pieces to aggressive positions if they can be held there; for example, knights on the fifth and sixth ranks on central or bishops' files, and rooks on the seventh rank for winning pawns and restricting the opponent's king.

Make your pieces co-operate. Co-operation means the achieving of sub-

goals by groups of pieces. For example, have a pawn protecting your advanced knight; connect your rooks by castling; double your rooks on open file or seventh rank. Avoid blocking your pawns with pieces.

Try to make double threats, as in many games.

Make each piece as efficient as possible without over-burdening it. The most important work should be done best if the costs are equal. For example, overprotect the centre, so that the pieces doing this work will have flexibility. (This is again a matter of mobility, but not according to the formal definition in terms of number of squares attacked.) Generally, flexibility is important. (See also the last sentence of this sub-section.)

In quiet positions, aim to accumulate small advantages. This should also be done in turbulent positions, but it is then more difficult to judge. (It needs a master's skill to carry out a complicated combination in order to win half a pawn!) If your small advantages are otherwise transient try to convert them into more permanent ones. Look harder for combinations, that is, lower the threshold of 'forcefulness', when your small transient advantages add up to as much as say $2/3$ of a pawn, and also if your opponent has weaknesses such as cramped pieces, unprotected pieces, exposed king or queen, weak squares in front of king, queen and king in a line. The greater your non-material advantage, the lower the threshold of 'forcefulness' should be.

Standard won classes of end-games should be in store, since they provide endpoints on the analysis trees, and serve a similar function to the stored positions in Samuel's draughts program. A classification of mating positions and motifs serves a similar function (see Appendix G).

The following principles apply especially to the end-game.

An extra pawn is usually enough to win, if there are pawns on both the Q and K side of the board.

If each side has one bishop and these stand on squares of opposite colours, then an advantage of one pawn is usually not enough for a win.

A B is better than a N when there are pawns on both the Q and K side.

K and Q beat K; K and R beat K; K and two B's beat K; K, B and N beat K; K and two N's do not beat K; K and B beat K and 32 B's all on white squares!

If you are two pawns up (in the end-game) you can force a win by advancing them, since your opponent will have to give up a piece for them. This is a special case of the Law of Multiplication of Advantage.

K and P beat K in situations that could be completely formalised without great difficulty. For the present, note that it is useful to get the K in front of the P, and to get the 'opposition' (as in draughts).

K and Q usually beat K and R. The method, which is easy to express in language, but not to formalise, is to bring the K and Q close to the opponent's K and R, force them to the edge of the board, and, by threatening to win the R by pinning it, to force it away from the K. Then it can be forked or mate given.

A rook's pawn is stronger against a knight than other pawns are.

K, B, and RP beat lone K if the corner in front of the pawn is one the B can go to.

Put rooks behind your own passed pawns.

KUFTEG (king up for the end-game). For the king is worth four pawns in the end-game and should be used. If you are not yet sure where to use it, put it in the centre. It will then have less distance to go than it might have had; in other words the king, like the knight, has more flexibility when it is in the centre.*

4.3. Review the literature of chess programs to see if it contains any principles not already in the literature of chess.

4.4. Classify the various principles, in order that later formalisation should be more efficiently performed, and in order to find gaps.

4.5. See if the literature of learning theory, concept formation, pattern recognition, perception, and associative memory suggests new principles. If so, revise step 4.4. Don't devote five years to this step!

4.6. Try to express the various principles unambiguously and, where possible, quantitatively.

4.7. Devise a special-purpose chess language for expressing the principles as unambiguously as possible, but with not too large a vocabulary. (See Newell & Simon 1965, p. 62, for a preliminary list of chess vocabulary.) *Complete lack of ambiguity will not be essential if the application is man-machine communication, since the machine might be able to make syntactic transformations without full grasp of the semantics.* Ambiguous statements can be stimulating and concise, for people.

4.8. See if the terms of this language can be expressed in Algol or similar language.

4.9. Make a study of chess planning, to see whether it requires a language different from the expression of the chess principles. For example, one might say, 'Since the position is closed, I have time to get my knight round to Q5'. This will involve us in the study of the minimal syntax required for the convenient expression of chess principles and planning.

As an example of planning, consider the position in Fig. 1, from Lasker (1932, p. 24):

```

K . k . . . . .
P . . . . .
N . . . . .
. . . . .
. . . . .
. . . . .
. . . . .
. . . . .

```

FIG. 1. White to move. An example in which an efficient analysis is linguistic. Notation: lower case for Black.

* It is a universal principle of rational behaviour that flexibility has to be sacrificed when a decision is made, but a decision to *acquire* flexibility can be made. There is no contradiction here, since 'flexibility' can be achieved at various levels of a hierarchy.

A human might reason in the following manner. Could a machine do this?

'My opponent has only a king and I have a knight and pawn, so I can win only if I can promote the pawn (goal). In order to advance the pawn I must move my king to N7 or N8 (subgoal). This is prevented at present by the black king. He can continue to prevent it (his goal is to do so) by moving his king to B2 and back to B1 and so on (his plan). I must move my knight into such a position as to control his B2 when his king is at B1, or to control his B1 when he is at B2 (subsubgoal). Consult list of theorems about knights. Each time they move, the colour of their square changes. Ah, but each time the black king moves the colour of its square changes (inspiration). Therefore I can never succeed in my subsubgoal, and cannot win, if he continues with his plan.'

If we could put plans in order of merit, then plans, counterplans, counter-counterplans, etc., would form a *tree of plans*, and iterative minimaxing would apply to it.

4.10. Estimate the cost of having built-in facilities for making chess programming easier, on a general-purpose computer. Consider whether some of these facilities could be generalised so as to be useful for other kinds of problem-solving. For a review of more general problem-solving, see Newell and Simon (1961). Remember that the problems of hardware and software are inseparable. General problem-solving programs should not be divorced from general problem-solving machines.

4.11. Plan the effort so that it can be broken up into self-contained pieces, for clearer understanding and for parcelling out.

4.11.5. Work out how to organise a program so that positions that occur on an analysis tree, together with their analyses, are held in store for several moves in the course of a game, until the chance that they will occur in later analyses in the game becomes negligible (see also Section 4.23 below).

4.12. Write separate programs for combinations, for the openings (see Appendix J), for the middle-game, and for the end-game, just as separate books are written on these aspects of the game. What other aspects merit separate treatment?

4.13. Write out routines for winning several special end-games, without analysis trees, that is, by using procedures already known to chess players.

4.14 (continuing 4.8). Write a compiler in Algol, or in some other programming language, or in some modified form of an existing programming language, for the programs written in the chess language.

4.15. Assemble theoretical and experimental arguments for the evaluation of the relative values of the pieces, squares, and other features of the positions (see Appendix F). Can these be related to the changes in the log-odds of winning as against losing, or some such probabilistic measure? (see the discussion of Good 1959, in Section 3 above). Is there a general measure for the turbulence of a position? (see Appendix H). Chess programming and mathematical theory about chess should go hand in hand, but the programming will involve more work. Given the number of times that each player

controls each square, in a quiescent position, what is the player's expected advantage?

4.16. Work out further theory concerning optimal searches in trees. Is it possible to make a tentative evaluation of a chess or draughts *program* by calculating the statistical distribution of size of the analysis tree for given rules concerning the thresholds of the agitation at the endpoints? How should the thresholds for selecting endpoints on the trees vary with depth, with agitation, with the amounts of time left on the two faces of the chess clock? (see Appendix H). How should we estimate the value of storing previously analysed positions?

4.17. How should goals be generated? Each potential goal on a check-list can be considered, and rules worked out for assigning relative importance to them. I think we should aim at estimating the probability that we can achieve each goal, and the utility to us if we achieve it (see Appendix C). For this purpose, I think 'progressive deepening' is necessary (see Section 4.23 below).

4.18. How should the move-generators be organised, i.e., what *moves* should be ignored?

4.19. How should a variety of evaluation functions be handled, each appropriate to a different goal? Can evaluation functions be conveniently adjusted, during an analysis, instead of recomputing them in detail for each move considered?

4.20. Select about fifty quiescent chess positions, get a chess master to put them in order of merit for the player (that is, the side with the move), and then do the same with a variety of evaluation functions. Try to find an evaluation function, partly by hill-climbing, that puts the positions in nearly the same order as the chess master. If there are any bad discrepancies left over, think again and recheck with the master.

4.21. Correlate evaluation functions with those obtained by analysis plus iterative minimaxing, with the object of automatically improving the coefficients (see Samuel 1959, 1963).

4.22. Find evaluation functions that indicate whether a position cries out for startling moves, that is, moves that would be rejected, on a superficial analysis, by most of the goals. Without such evaluation functions, the machine will not be a genius.

4.23. Formalise the process of 'progressive deepening'. This is especially important for problem-solving in general. Example: the threshold at various depths of analysis should depend on how large the analysis tree is going to be. So we should first do a quick pilot analysis, for estimating the size of the tree, and then a more comprehensive analysis. There should be several stages in this process, since no simple method is likely to work. The difficulty is that a single branch of a tree can be greatly enlarged as a consequence of a slight change in the thresholds. The amount of the enlargement cannot be predicted. Hence pilot analysis might have to be applied to subtrees, as well as to the complete tree.*

* The device of holding in store positions that occurred on the analysis trees earlier in the game (see Section 4.11.5) will be relevant to pilot analyses.

Another application of progressive deepening is that ideas that occur at one stage can be used again at a later stage. For example, a move that was considered at one stage and seemed 'interesting' has a better chance of coming up for consideration at a later stage, whether or not it was sound at the earlier stage.

4.24. The relevance of progressive deepening to the formulation of sub-goals should be worked out, or at least worked on.

4.25. Play a few games and make improvements in the program as they suggest themselves.

4.26. 'Conversations' should be held with the computer in the special chess language (man-machine synergy). See what modifications in the chess language are required in order to answer new types of questions about chess. For example, we can ask the machine to count the number of distinct games in which White mates on his third move. (According to an analysis I did in about 1932, the answer is 367.) Another example would be retroactive analysis.

4.27. Try to find a transformation that converts an evaluation function into $p_W - p_L$ (or more generally into $p_W - \lambda p_L$), where p_W and p_L are the probabilities of winning and losing respectively. Try to do this even for turbulent positions (see Appendix H).

4.28. Investigate how the analysis should be modified in order to allow for the strengths, weaknesses, knowledge, habits, and style of a particular opponent.

4.29. Attempt to generalise the entire plan to a wider class of games. Here 4.26 will have been useful practice. Begin with randomised chess (see Appendix J), new and unusual kinds of pieces, and three-dimensional chess (see, for example, Good 1957). Many interesting variations on chess are known, such as Kriegspiel (which is not a game of perfect information), Buying Chess (Landau), Panzerspiel, Shotgun, chess with extra kings, chess without kings, chess on a torus, chess with mined squares, losing chess, and chess with a gradually increasing number of moves of each side. Randomised chess is special, since it is intended to replace ordinary chess, i.e., ultimately to be called 'chess'.

Note that, owing to limitations of human spatial visualisation, the machine might play better in three dimensions, relative to a man, than it would in two dimensions.

APPENDIX A. THE BOREL-VON NEUMANN THEORY OF GAMES, AND ITERATED MINIMAXING

In the 1920s, Borel and von Neumann founded a theory of games, which reached large book size in von Neumann and Morgenstern (1944). For the early history, and kudoological remarks, see Borel, Fréchet and von Neumann (1953). In the zero-sum two-person game the two players independently choose two 'plays', i and j , which can be identified with the choice of a row and a column of a matrix, and the pay-off to the first player is a_{ij} and that to the second player is $-a_{ij}$. Neither player knows the other's play. The matrix (a_{ij}) is called the *pay-off matrix*. Chess is a special case of a zero-sum two-person game if we imagine that, according as the

first player loses, draws or wins, he collects -1 , 0 , or 1 from the second player. The play, i , is then to be interpreted as a set of rules which uniquely determine the player's move in each position, so that a play is not just a chess move, but is a single-valued function of positions taking values that are moves. For the selection of an optimal play, 'all' that is necessary is to work back from the end of the game, and deduce in turn whether each position is a theoretical win, draw, or loss for white. But the number of possible chess positions is about $10^{46\pm 3}$ (see Appendix E) so an exhaustive analysis is virtually a physical impossibility. Perhaps the use of probability ideas in chess has the same kind of justification as its use in classical statistical mechanics!

The Borel-von Neumann theory can be applied to chess in a practical way only if the game is regarded as probabilistic, and then only in an exiguous manner. The theory shows, for example (what is obvious), that, if we can evaluate the position at all move-pairs from the current position, and if these evaluations are regarded as expected pay-offs, then we minimise our expected loss by playing the minimax move. This is the best move for us on the assumption that our opponent plays optimally, otherwise our best move is the one that maximises our expected gain after allowing for a probability distribution over all the moves that our opponent might make. When we do not assume that our opponent will play optimally, we are playing either psychological or trappy chess or both. All this can be extended to deeper analysis of a position, and leads, when we assume our opponent is playing well, to mini-maxi-mini-maxi-mini-maxing, etc., conveniently called *iterated minimaxing*, although the epithet 'iterated' is often omitted in discussions of chess and draughts. For psychological chess, a more appropriate term is (iterated) 'expectimaxing', a term that Michie (1966) applies to games against Nature, or against an opponent who moves randomly. Minimizing is a special case of expectimaxing from the present point of view.

Since we associate $+1$ with a win for us, it would be more correct to use the term 'maximining' (and 'maxi-expecting') but 'minimaxing' is standard.

APPENDIX B. SOME COMMENTS ON SAMUEL'S DRAUGHTS (CHECKERS) PROGRAM

Samuel (1959) described a draughts program which is capable of sometimes beating near masters. His objective was to experiment with a learning program and he used an evaluation function whose coefficients could be varied by the program.* He resisted the temptation to store book variations, but he did store positions that occurred often enough in the experience of the machine. Presumably the effect of this 'rote' learning is that the program learns the openings to considerable depth, especially if it were pitted against a fixed player for a larger number of games in succession. (It can forget positions as well as learn them.) This device would be much less useful in chess since it has far more openings and not many of these extend almost to the end-game as they do in draughts.†

Samuel's paper was reprinted in 1963, together with the beginning of a game played in 1962 against Robert W. Nealey, who had been Connecticut champion, but was not at his top form. Unfortunately the end of the game was omitted in error. The position at the point where the record ceased is shown in Fig. 2 (i). The remainder of the game has been communicated to me by Dr Arthur Samuel, and is: 12-19, 32-27; 19-24, 27-23; 24-27, 22-18; 27-31, 18-9; 31-22, 9-5; 22-26, 23-19;

* It was not clear that these coefficients had reasonably well converged after several games. This makes one wonder how much of the evaluation function is really relevant to the success of the program.

† It would be interesting to have a typical frequency count ($n_0, n_1, \dots$), where n_r = number of stored positions in the draughts program at depth r , from the initial position.

GOOD

26-22, 19-16; 22-18, 21-17; 18-23, 17-13; 2-6, 16-11; 7-16, 20-11; 23-19, White (Nealey) resigns.

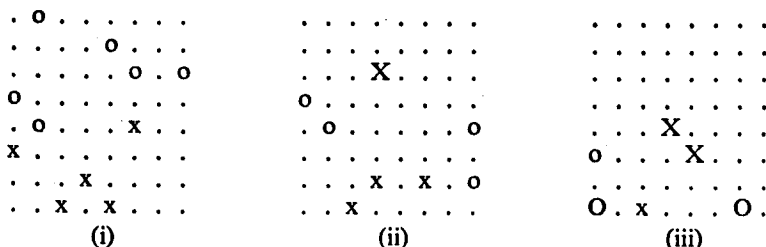


FIG. 2. Positions (i) and (ii) occurred in the game Machine v. Nealey, 1962; and position (iii) would have occurred if Nealey had played differently on his penultimate move. Notation: Black, crosses; White, noughts. Kings are in upper case.

In my opinion, White could have drawn the game if he had played 16-12 near the end in place of 16-11, which is a tactical blunder (see Fig. 2 (ii)). He would then have made a king which he would have been forced to exchange, but he could soon make another one. Black will be able to make a second king, but he will not be able to dig White's king out of the double corner, since White, in Fig. 2 (iii), can move his other king backwards and forwards while Black has his kings in the position shown, or further away. The interesting thing about this analysis, as mentioned in Section 1, is that it is not easy to see how it could be expressed in numerical terms.

In a letter of April 19, 1966, Dr Samuel remarks that, in a return mail match, Nealey won one game and drew five. Also that the machine played four games against each of W. F. Hellman, the world champion, and Derek Oldbury, a British champion, and that the machine lost all eight games. The machine, the IBM 7090, was held to five minutes per move at most, whereas the humans had no time limit. It seems reasonable to describe the machine as a near minor master, especially as the winning players required 70 to 80 moves to achieve winning positions. My guess is that the machine is very sound tactically but lacks judgement.

APPENDIX C. WHAT IS THE 'BEST MOVE'?

'Theoretically' we can do a complete analysis of any given position, π , and arrive at the endpoints of the tree, whose values (win, draw, or loss, for the machine) are known directly from the rules of chess. Then we can backtrack, by iterated minimaxing, to get the values of all the positions that can be reached from π in one 'half-move', i.e., one move by the player whose turn it is. In this way we can in principle determine the moves that give nothing away, and, from an over-logical point of view, we could describe these moves as 'equal best'. Then again, we could instead define the best move (or equal best), if π is theoretically won, as a move that forces checkmate in the minimum number of moves against any play however good. The best move in a lost position would analogously be defined as one that delays mate for as long as possible. If π is a theoretical draw, then in much the same spirit we could define the best move as one that delays for as long as possible an indisputable draw (assuming the 50-move drawing rule to be mandatory). For since, at the moment, we are assuming the machine to be a perfect player, we should like the opponent to have as many opportunities as possible for making a mistake. Or it might be a little more logical to maximise, not the number of moves to an indisputable draw, but the product of the number of losing choices that the opponent would have at each of his moves. For this is the total number of ways he has of

going wrong. It is a little more realistic simply to maximise the probability that he will go wrong, and this brings us close to a practical definition of 'a best move'.

Suppose that, as a result of each of the moves the machine can make in position π , we can estimate the (superficial) probabilities p_w, p_D, p_L , that we (the machine) will win, draw or lose. For a given state of the tournament, prize-list, etc., these possibilities will have various utilities, u_w, u_D, u_L , for us. The expected utility of the move will therefore be of the form $p_w u_w + p_D u_D + p_L u_L$. But since $p_w + p_D + p_L = 1$, this expected utility is of the form $\alpha p_w - \beta p_L + \gamma$. Since we are concerned only in finding the best move, we can drop the constant γ , and also divide out by α . Hence the expected utilities, up to an irrelevant linear transformation, are of the form $p_w - \lambda p_L$. A similar argument can be applied by the opponent, so that, if for the moment we regard the probabilities as objective, we can infer that $\lambda = 1$ if the game is 'zero-sum'. So it is fairly reasonable to take the expected utility as $p_w - p_L$; but $p_w - \lambda p_L$ is more reliable.

If our evaluation functions enable us to infer the value of $p_w - \lambda p_L$, then we could 'in principle' backtrack and find the best moves in the sense of expected utility. If we were dealing with a game, like Hex, in which there are no draws, then the best move in the sense of the principle of rationality would also be the best in the sense of maximising the probability of winning.

There is a slight over-simplification in the above discussion since, in many tournaments, there is more utility in winning quickly than slowly. It preserves your energies for the next game. I have ignored this effect in this paper.

APPENDIX D. THE NUMBER OF POSSIBLE GAMES OF CHESS

If fifty moves are played on each side without any captures or pawn moves, then, in some sets of rules, the game is drawn, whereas, in other sets, the draw must be claimed. If the rule is taken as mandatory, then no game can last more than 6000 moves on each side. Also, in any position, even if all the pawns have been promoted to queenhood, the number of possible moves cannot exceed 321. Therefore the number of possible games is less than $321^{12\ 000} < 10^{30\ 000}$. (This inequality was given by Good 1939. With a little more work, the upper bound can be lowered somewhat.)

If we restrict our attention to reasonable games, or to master games, we get a much lower estimate. The method of estimation is similar to one used in certain Monte Carlo estimation problems having no reference to games (see Good 1954). If a_n is the expected number of moves, at a given standard of play, at the n th ordinal position, counting the initial position as the first, then the total expected number of games of not more than $2m$ moves on each side, for m less than say 200, whose moves are all of the approved standard, is about

$$\sum_{n=1}^{2m} a_1 a_2 \dots a_n (1 - \epsilon_1)(1 - \epsilon_2) \dots (1 - \epsilon_{n-1}) \epsilon_n,$$

where ϵ_n is the probability that the game would terminate immediately after the n th move.

de Groot (1965, p. 25) mentions the analysis of one game in which the average number of good moves per position did not exceed 2, but he does not state what the average was. The analysis of several games would be required in order to obtain reliable estimates of the a_i 's ($i = 1, 2, 3, \dots$). For $n > 15$, an adequate approximation to ϵ_n is $1/40$, as far as the present analysis is concerned.

Another approach is to consider how many opening lines are recorded in Evans (1965). There are about ten thousand lines given to an average depth of about 12 moves, so the average number of moves considered in each position is about

$$10\ 000^{1/24} = 1.48.$$

GOOD

These are almost entirely lines that have been played or analysed by good players. It seems a reasonable guess that

$$1.6 < (1 - \epsilon_n) a_n < 1.9$$

for most values of n , if extrapolation from the openings to the rest of the game is justifiable. (It is interesting to observe that, if 1.75 is the correct average in the opening, then *Modern Chess Openings* ought to run to fifty volumes, that is, only a fiftieth of equally good lines are at present recorded. This might give some measure of the conventionality of 'the book'.)

If this guess is right, then the number of games of not more than m moves on each side is roughly

$$\frac{1}{40} \cdot \frac{\mu^{2m+1} - 1}{\mu - 1}$$

where $1.6 < \mu < 1.9$, that is, the number lies between $1.6^{2m+1}/24$ and $1.9^{2m+1}/70$. These bounds are tabulated approximately in Table 1.

TABLE 1

The number of good games of not more than m moves on each side

m	10	20	30	40	50	100	200
$\log_{10}$ (lower bound)	3	7	11	15	19	39½	80½
$\log_{10}$ (upper bound)	4	9½	15	20½	26	54	110

It is interesting to note that, although some 99 per cent of games terminate before White's hundredth move, nevertheless the number of possible well-played games longer than this is far greater than the number that are shorter.

The number of good 'lines', that is, games not necessarily terminated, up to Black's m th move, is about $(\mu^{2m+1} - 1)/(\mu - 1)$ and is bounded by about 1.6^{2m+1} and 1.9^{2m+1} . These bounds are tabulated in Table 2.

TABLE 2

The number of good 'lines' up to Black's m th move

m	10	20	30	40	50	100	200
$\log_{10}$ (lower bound)	4½	8½	12½	17	21	41	82
$\log_{10}$ (upper bound)	6	11½	17	22½	28	56	112

The above table can be inverted. For example, we can infer from it that a catalogue of 10^{12} good lines would extend to between 21 and 29 moves on each side, or say 25 moves, give or take a few moves. Moreover, if the machine had the white pieces, and was satisfied to adopt a fixed repertoire, 'in principle' it could get to about this depth with a catalogue of only a million lines (this assumes the square root law mentioned in Appendix J). As soon as the opponent played a slightly inferior move, the machine would have to begin exercising intelligence.

APPENDIX E. THE NUMBER OF POSSIBLE CHESS POSITIONS

Shannon (1950, p. 260) states that the number of possible chess positions is of the general order of $64!/(32!8!2!^6)$ or roughly 10^{43} . The formula indicates that he

was assuming that no pawn had been promoted. In Good (1951), I calculated that the number of positions in which no pawn has been promoted, and there are no doubled pawns, is less than 2×10^{39} . The number of positions in which no capture has occurred is about 10^{32} . Allowing for all possibilities the number is less than

$64.63 \sum_{r=0}^{30} (6^2)^r 10^r$ which is about 2×10^{50} , but if the number of moves since the last

capture or pawn move is regarded as part of the position, the upper bound is 10^{52} . I should say that 10^{46} is correct for the total, within a factor of a thousand. But most of these positions would be fabulously improbable. A master chess player would be happy if he knew what the best move was in 99.9 per cent of the positions, with nearly level material, weighted with their probabilities of occurring, that would occur in master chess, without blunders. Judging by the tables in Appendix D, and allowing for the possibility of repetitions of positions in the end-game, it may be that this number would not be more than about 10^{24} . Conceivably a good estimate of the total number could be made if a very comprehensive categorisation of chess positions were first produced, in which positions would fall into clumps of trillions of positions each. Another approach would be to choose a large number of dispositions for the pieces, without pawns, and, for each of them estimate the number of ways of distributing the pawns legally—in effect a method of stratified sampling.

To make a rough comparison of the number of possible chess positions with that of draughts positions, the very crude upper bounds, 13^{64} and 5^{32} or 3^{32} , suggest that the former is about the cube or fourth power of the latter.

APPENDIX F. THE VALUE OF THE PIECES AND SQUARES

A satisfactory theory of the values of positions should include a theory for the values of the pieces as a special case. So it is worth while to look for a theory that gives reasonable values to the pieces. Then we could try to generalise the theory.

The values of the pieces P, N, B, R and Q, have been found by experience to be approximately proportional to 1, 3, 3, 5 and 9, and a king is worth about 4 in the end-game. These values vary with the position and with the number of pieces on the board. For example, two knights are worth less than rook when the only other pieces on the board are two kings, in so far as two knights and a king cannot force mate against a lone king. But the values given are applicable rather widely. For example, two knights are about the equal of six pawns on the second rank even when the kings are removed, as I have found by experiment.

A theoretical attempt to evaluate the pieces was made by H. M. Taylor in 1876, reported by Coxeter (1940, pp. 162-165). The value of a piece is taken as proportional to the average number of squares controlled, averaged over all 64 positions of the piece on the board. This argument leads to the relative values of N, B, R and Q: 3, 5, 8 and 13. Coxeter (or Taylor) goes on to modify the argument by asking instead for the probability of 'safely' giving check, that is, without being *en prise* to the king, if the piece and king are both placed on the board at random. This gives the ratios 12, 14, 22 and 40. These values are good, but this modification of the argument is artificial. For the piece values seem to be valid even for chess without kings, as I have found from some limited experiments, such as the one just mentioned. In fact they are probably more valid when there are no kings. Consider, for example, a piece, which I shall call the Galactic Emperor, that controls all the squares on the board. When the kings are on, it is worth at least seven queens, but, measured by squares dominated, it is worth only about three queens, on an open board. Therefore I believe that a theory that allows for the presence of the kings is too difficult to start with, and that it is more sensible to try first to obtain a theory that arrives at the known values without reference to checking.

GOOD

In about 1936 (unpublished) I tried to make some allowance for the number of pieces on the board:

Let p be the probability that a square of the board is occupied. I assumed that the value of a piece is proportional to the expected number of squares dominated, averaged for all positions of the piece, and obtained the following values for the knight, bishop and rook.

$$\begin{aligned} E_N(p) &= 5.25 \text{ (independent of } p), \\ E_b(p) &= (140 - 196p + 182p^2 - 112p^3 + 44p^4 - 10p^5 + p^6)/16, \\ E_R(p) &= \frac{4}{p} - \frac{1}{2p} \left\{ \frac{1 - (1-p)^8}{p} \right\}. \end{aligned}$$

The expected number of squares dominated by the queen is $E_Q(p) = E_R(p) + E_b(p)$. But we should allow also for the fact that rook and bishop have overlapping fields of influence. For example, a rook on Q5 and a bishop on KB4, on an open board, dominate only 24 squares instead of 27, and so lose about one ninth of their value. This gives a partial explanation of why a queen is worth more than a rook plus a bishop. A similar argument explains why two bishops are worth more than a bishop and knight, or than two knights. It is usually better to control two squares than to control one square twice, not allowing for the different values of squares.

Table 3 gives the numerical values of $E_R(p)$, etc., for a few values of p .

TABLE 3
Values of $E_N(p)$, etc.

p	1	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	0
N	5.25	5.25	5.25	5.25	5.25
B	3.07	4.7	5.72	6.35	8.75
R	3.5	6	7.5	9	14

The case $p=1$ cannot occur in ordinary chess, but it is certainly reasonable that a knight would be worth more than a rook if all 64 squares were occupied. The case $p=\frac{1}{2}$ is not expected to apply for the first several moves of a game, when we know roughly where the pieces stand. The initial position can be examined in its own right, as indeed can any given position. But it would be too short-sighted to say that rooks, bishops and queens have no value in the initial position, merely because they cannot move. It would be more reasonable to make some guess concerning the probability that each piece will occupy the various squares after any given number of moves, and then to use these probabilities as weights in a weighted average of the numbers of squares dominated. Moreover we should discount the future in some way, for example, by giving further weights of μ , μ^2 , μ^3 , ..., to positions 1, 2, 3, ... moves ahead, where μ is some positive number less than unity.

The case $p=1/3$, which is typical in the middle-game, gives values that are generally regarded as good average values of the pieces. It would be interesting to try to verify the remainder of Table 3 experimentally.

To make the theory more accurate we may attach a value to each square, making the central squares more valuable than the others. To do this we can use a similar principle to the one used for pieces, that is, we can find the average number of squares dominated from the square when various pieces are placed there. These can be taken as 1Q, 2Rs, 1B (only one colour!), 2Ns, but not 8Ps since not every pawn has a reasonable chance of getting to the given square. It seems reasonable

to assume $1\frac{1}{2}$ pawns. I found the following values for the squares by this method. The squares are labelled as in Fig. 3.

	D	E	F	I
	B	C	E	H
	A	B	D	G

FIG. 3. Labelling of squares. The labelling is centrosymmetrical, apart from the edges.

$$A: 87 - 84p + 44p^2 - 8p^3.$$

$$B: 83 - 82p + 55p^2 - 20p^3 + 3p^4.$$

$$C: 83 - 92p + 80p^2 - 40p^3 + 8p^4.$$

$$D: 75 - 90p + 95p^2 - 50p^3 + 18p^4 - 3p^5.$$

$$E: 75 - 100p + 110p^2 - 70p^3 + 23p^4 - 15p^5.$$

$$F: 71 - 120p + 160p^2 - 120p^3 + 48p^4 - 8p^5.$$

$$G: 65 - 108p + 130p^2 - 110p^3 + 63p^4 - 21p^5 + 3p^6.$$

$$H: 65 - 118p + 155p^2 - 130p^3 + 86p^4 - 21p^5 + 3p^6.$$

$$I: 63 - 138p + 205p^2 - 180p^3 + 93p^4 - 26p^5 + 3p^6.$$

We can use these values of the squares to improve the estimates for the values of the pieces. We could then use these improved estimates to improve the estimates for the squares and so on. There will always be an error because of the assumption that the density of p is the same over the whole board, and also because we can choose, for example, not to keep knights at the edge of the board, as a general rule. The chance that a knight is on the edge, after the opening, is much less than $28/64$, if the players are any good!

I did this algebra before there were any electronic computers. It will now be easy to do the arithmetic.

If we wish to allow for the existence of kings we should give extra weight to the squares in their vicinities, or in some other way. We might, for example, compute a 'checking value' for each piece as the probability of giving check without being *en prise* to the king, and then take a linear combination of the checking value and the square-domination value.

The values of the pawns require a different theory since (i) they can be promoted, and (ii) they move differently from the way they capture.

It should be considered whether the values of the pieces can be related to the probability of winning. Should the value be taken as the improvement in the log-odds of winning as against losing?

Theories of evaluation of pieces could be tested by inventing new pieces; we are not inevitably tied to the small sample of distinct kinds of piece that occur in ordinary chess. An example of this test, was given above in connection with the Galactic Emperor, although perhaps this piece is too extreme for most purposes.

APPENDIX G. CATEGORISATION OF MATING POSITIONS AND CATALOGUE OF FREQUENT MATING MOTIFS

When attacking the king, human players find it useful to be able to recognise the possibility of mating motifs and mating positions. To have a repertoire of these

GOOD

things is helpful in goal formation and in the pollarding or pruning of analysis trees.

For mating motifs, a useful reference is Vukovic (1965), and there are many other books that deal with this topic. Here I should like to mention a categorisation of mating positions.

(a) *Three-filer*. An example is a mate with two rooks, the opponent's king being on the edge of the board. It is natural to think of the three squares off the board as forming an imaginary file. Another example is mate with a rook and king against a king on the edge, or a mate on the back line, when the opponent's king has three pawns in front of it.

(b) *T and tunnel.* For example, the final position in the game 1 P-K4, P-K4; 2 Q-R5, K-K2; 3 Q×KP mate.

(c) *Cross diagonals and L.* For example,

$$\begin{array}{ccccccc} & & & & & & K \\ \cdot & \cdot & \cdot & N & \cdot & R & \cdot \\ & & & p & k & P & \\ \cdot & \cdot & \cdot & p & \cdot & & \end{array}$$

(d) *Parallel diagonals and two stops.* For example,

$$\begin{array}{ccccccc} & & k & r & & & \\ & p & & & & & \\ NN & & & & & & \end{array}$$

With a little artificiality, most mating positions can be put into one or more of these categories. If a machine could do this it would really have arrived.

APPENDIX H. QUIESCENCE, TURBULENCE, AND AGITATION

Positions for which there is no move forceful or purposeful enough to be worth analysing were independently described as 'quiescent' by Shannon (1950) and by Good (1950b). Turing (1953) called such positions 'dead'. One objection to this term is that a 'dead draw' is a standard expression among chess players. Also it is useful to be able to refer to 'degrees of quiescence', whereas 'degrees of deadness' sounds facetious, in spite of the expression 'dead as a door-nail' and of the notorious variability of deadlines. A good name for the degree of quiescence is simply the 'quiescence' of the position, or inversely one might refer to the 'turbulence'. A quiescent position is one of low turbulence, a combinative position is one of high turbulence. It is possible to define turbulence with respect to goals other than material balance, such as control of the centre (cf. Newell *et al.* 1958). The position illustrated in Fig. 4 is of high turbulence with respect to material balance.

.	.	.	.	.	.	.	.
.	.	k	n	R	.	.	.
.	.	.	r	Q	b	B	.
p	p	p	b	q	n	p	p
P	P	P	R	N	N	P	P
.	.	.	p	p	r	B	.
.	.	.	P	P	P	.	.
.	.	.	.	.	K	.	.

FIG. 4. A position of high turbulence!

Before considering a more precise definition of 'turbulence' we discuss an allied notion called 'agitation'.

If, in the analysis of a position π_0 we reach a position π , the turbulence of π is not quite an adequate criterion for deciding whether to suspend further analysis. If π were obviously won we would suspend its analysis even if it were turbulent. We need some measure, called say 'agitation', that measures turbulence and also takes into account whether the position is obviously won, drawn, or lost. If we can measure the agitation of π , then we would set a threshold on this agitation multiplied by $P(\pi|\pi_0)$, the probability of reaching π from the current position π_0 . (Notice that, when we estimate this probability, we are in part, estimating the probability of our own future actions, not just those of our opponent. In my opinion, the same thing happens in the definition of a 'decision'. See Good 1964.) If this threshold is not reached, we would almost suspend analysis of π . I say 'almost' because there are a few points to attend to before finally classifying π as an endpoint on the analysis tree of π_0 . We should first (i) check whether π is a position known definitely to be won, drawn, or lost; (ii) if π is favourable to us, we should see if we have a move that prevents check; (iii) if π is unfavourable to us, we should see if we have a check; (iv) if π seems to be an obvious draw, we should carry out both the tests (ii) and (iii).

How then should 'agitation' be measured? It will be recalled from Appendix C that the utility of position π , if we reach it, is of the form $P(W|\pi) - \lambda P(L|\pi)$, where $P(W|\pi)$ and $P(L|\pi)$ are, in some sense, the probabilities of a win or loss, given π . If we decide to treat π as an endpoint, then these probabilities are 'superficial' and $P(W|\pi) - \lambda P(L|\pi)$ may reasonably be described as the 'superficial expectation of the utility' or, for short, as the 'superficial utility'. If we decide to suspend the analysis of π , that is, to regard it as an endpoint, it should be because we believe that a deeper estimate of the utility of π , obtained say by spending a dollar on its analysis, is unlikely to make much difference. We must therefore have some method of guessing the expected value of $|U(\pi|\$) - U(\pi)|$, where $U(\pi)$ is the superficial utility of π , and $U(\pi|\$)$ is the expected utility of π in the light of a dollar's worth of analysis, \$, or of course some other unit. The agitation $A(\pi) = A(\pi|\$)$, of π is the guessed expected value of $|U(\pi|\$) - U(\pi)|$, and the decision whether to treat π as an endpoint in the analysis of π_0 is to be made by setting a threshold on $A(\pi)P(\pi|\pi_0)$. Part of the five-year plan is to find an evaluation function that estimates $U(\pi)$, and another one that estimates $A(\pi|\$)$. We need evaluation functions even for turbulent positions.

A well-known issue in the theory of rationality is whether the distribution of a utility is relevant except in regard to its expectation. My own belief, shared by most 'rationalists', is that only the expectation is relevant, but this is on the assumption that all the relevant calculations have been completed. In complicated problems, like mathematics and chess, this assumption is unrealistic, and superficial probabilities and expectations have to be used (cf. Good 1962b). The guessed expected value of $|U(\pi|\$) - U(\pi)|$ is roughly proportional to the standard deviation of the superficial subjective distribution of the true utility of π . The true utility of π is of course either 1 or $-\lambda$, or $\frac{1}{2}(1 - \lambda)$, if both we and our opponent are perfectly rational.

I believe that this notion of the agitation of a position is applicable to almost every decision in practical life, since it is very rare that we can complete our reasoning in a practical problem. Instead of using the axiom of probability that $P(E|H) = P(F|H)$ when E and F are logically equivalent, we are forced to use the weaker one that these probabilities are equal when we have *proved* that E is equivalent to F (Good 1950a, p. 49.) Both humans and machines must often use imperfect logic.

It seems reasonable to define the turbulence, $\tau(\pi|G)$, of π with respect to some goal G , as the superficial expectation, or guessed value, of $|L(G|\pi, \$) - L(G|\pi)|$, which equals $|W(G:\$|\pi)|$, where L denotes log-odds, W denotes 'weight of evidence', and the colon denotes 'provided by'. But it seems that decisions should depend

more on agitation than on turbulence. The utilities should allow for all goals that we are prepared to consider. Agitation with respect to a goal can also be defined by interpreting U appropriately.

If these ideas could be properly worked out, they would be applicable to the question of finding the value of storing previously analysed positions, as in Samuel's draughts program.

At the end of Section 1, we mentioned the problem of deciding what moves to discard 'without analysis'. I think the right approach to this question is to try the move, and estimate $|U(\pi') - U(\pi)|P(\pi|\pi_0)$ where π' is the position reached after that move is played, and the utility $U(\pi')$ is estimated on the (false) assumption that the player retains the move. If this estimate is small enough, then the move can be treated as of no importance. In fact, if this estimate is small, then $P(\pi'|\pi)$ would be small, since a free move is usually quite advantageous. An exception is when the player is *Zugzwang*, and also in some end-game positions when no move is any good.

APPENDIX J. HOW SHOULD THE OPENINGS BE PROGRAMMED? SHOULD CHESS BE REPLACED BY RANDOMISED CHESS?

One obvious suggestion for handling many of the openings is to store nearly the whole of *Modern Chess Openings*. Then the machine will not go badly wrong until the opponent leaves the book. Moreover, some of the published lines would be clear wins for the machine, whereas those that were known to be bad for the machine could be avoided. In correspondence chess, this device would put the machine at a disadvantage, but in 'over the board' chess, the machine would sometimes win without any 'intellectual effort'. This would indeed be a hollow victory.*

Chess was presumably invented as a test of a kind of intelligence, and it is no more reasonable that the machine should be allowed to win by throwing the whole book at its opponent than that humans should be allowed to win tournaments by the learning of innumerable lines. Both for machine programming, and for the health of chess in general, the present rules of the game should undoubtedly be modified, and so-called 'randomised chess' should be adopted by the International Chess Federation, the F.I.D.E. (see, for example, Good 1952-55, 1965, p. 35).

In randomised chess the white pieces on the back line are randomly permuted before the game begins, but if the two bishops turn out to be on squares of the same colour the one on the right is interchanged with the piece on its left. This gives rise to 1440 essentially distinct positions if the black pieces are set up by mirror reflection. Castling can be defined by the rule that the rook is moved up to the king and the king hopped over it. Randomised chess has exactly the same seriousness as ordinary chess, and the basic principles, such as the need to control the centre, are the same. The theory of the openings is genuine theory and not largely practice and fashion as in ordinary chess.

But since it is too much to hope that the F.I.D.E. will be rational enough to organise this change in the face of opposition from vested interests, let us consider what we can do other than swallowing the whole book.

The chess openings form a very large tree: there are about 150 000 moves in *Modern Chess Openings*, consisting of some 10 000 lines. But a player with the white pieces, and a fixed repertoire, need learn only about 100 lines, the square root of the number in the book, and another 100 lines for when he has the black pieces. The argument for this, which is not at all rigorous, is that the number of lines in the book is roughly k^n , where k is the number of choices in each position, and n is the length of the line; whereas a repertoire for White need have only about $k^{n/2}$ lines, since he can make a fixed choice in each book position. The argument

* It would not be hollow if it provoked the F.I.D.E. to go over to Randomised Chess.

is not rigorous because k is a decreasing function of the number of moves from the initial position, since the book is intended to be a book about the *openings*.

At any rate, one way that a computer could be used to help a chess player would be in the selection of an opening repertoire for him. (A repertoire is a Borel-von Neumann pure 'strategy' or 'play', but is incomplete.) If the repertoire were selected randomly, then the player's opponents would not be in a position for some time to take advantage of his published games in order to prepare variations against him. Likewise the machine could select a repertoire for say its next ten games. It would then be able to start a game quickly 'over the board', and perhaps finish by correspondence.

The problem of selecting an opening repertoire or subtree can be framed in several ways. For example, we could try to find a repertoire having minimum storage requirements (which would be especially useful for human players), or one with a preference for open games. It might well turn out that a machine would play a better combinative than strategic game in comparison with humans.

APPENDIX K. SYNERGY, A POINT OF TERMINOLOGY

Close interaction between men and machines is sometimes called 'man-machine symbiosis' and sometimes 'man-machine interaction'. The first term is inappropriate since, in correct usage, it refers to purely biological interaction. The second term does not convey the notion of the closeness of the interaction. Fortunately there is an English word already available, it is 'synergy'. In my opinion the 1970s will be remembered as the age of synergetics.

REFERENCES

- Bernstein, A. *et al.* (1958a). A chess-playing program for the IBM 704 computer. *Proc. Western Jt Computer Conf. (WJCC)*, 157-159.
- Bernstein, A., & Roberts, M. de V. (1958b). Computer vs. chess-player. *Scient. American*, 198, 96-105.
- Borel, E., Fréchet, M., & von Neumann, J. (1953). Discussion of the early history of the theory of games, with special reference to the minimax theorem. *Econometrica*, 21, 97-127.
- Coxeter, H. S. M. (1940). *Mathematical Recreations and Essays*, from the original by W. W. Rouse Ball. London: Macmillan.
- Euwe, M., & Kramer, H. (1964-65). *The Middle Game*, two volumes. London: Bell.
- Evans, Larry (1965). *Modern Chess Openings* (Griffith and White), 10th edn. W. Korn (ed.). London: Pitman.
- Fine, R. (1941). *Basic Chess Endings*. Philadelphia: David McKay; London: Bell.
- Golombek, H. (1966). Luck in chess. *Observer Weekend Review*, January 23, p. 31. London.
- Good, I. J. (1939). Mathematics and chess, *Eureka*, 1, 9-11.
- Good, I. J. (1949). The number of individuals in a cascade process, *Proc. Cambridge Philosophical Soc.*, 45, 360-363.
- Good, I. J. (1950a). *Probability and the Weighing of Evidence*. London: Griffin; New York: Hafner.
- Good, I. J. (1950b). Contribution to the discussion on Slater (1950), p. 199.
- Good, I. J. (1951). Private communication to Mr Leon Jacobson.

- Good, I. J. (1952/55). Notes on Randomised Chess in *Chess*, 17, 161, 1952; 20, 203, 1955; *Brit. Chess Mag.*, 73, 213-214, 1953.
- Good, I. J. (1954). Contribution to the discussion of a paper by J. M. Hammersley and K. W. Morton. *J. Roy. Stat. Soc. B*, 16, 68.
- Good, I. J. (1957). Letter in *Chess*, 22, 293-294, containing misprints, and concerning a form of three-dimensional chess played on a $4 \times 4 \times 6$ 'board'.
- Good, I. J. (1959). Could a machine make probability judgments? *Computers and Automation*, 8, 14-16, 24-26.
- Good, I. J. (1962a). Botryological speculations, *The Scientist Speculates*, I. J. Good, A. J. Mayne, J. Maynard Smith (eds.), 120-132. London: Heinemann; New York: Basic Books.
- Good, I. J. (1962b). How rational should a manager be? *Management Sc.* 8, 383-393; and in *Executive Readings in Management Science*, M. K. Starr (ed.). New York: Macmillan (1965).
- Good, I. J. (1964). Measurement of decisions, *New Perspectives in Organisation Research*, pp. 391-404. W. W. Cooper, J. Leavitt, and M. W. Shelly II (eds.). New York: Wiley.
- Good, I. J. (1965a). Speculations concerning the first ultraintelligent machine, *Advances in Computers*, Vol. 6, pp. 31-88. F. L. Alt and M. Rubinoff (eds.). New York and London: Academic Press.
- Good, I. J. (1965b). 'The mystery of Go.' *New Scientist*, 25, 172-174.
- Groot, Adriaan D. de (1965). *Thought and Choice in Chess*, G. W. Baylor (ed.). The Hague and Paris: Mouton. A translation, with additions, of a Dutch version of 1946.
- Horowitz, I. A., & Mott-Smith, G. (1960). *Point Count Chess*. New York: Simon and Schuster.
- Kister, J., Stein, P., Ulam, S., Walden, W., & Wells, M. (1957). Experiments in chess. *J. Assn Comput. Mach.*, 4, 174-177.
- Lasker, Edward (1951). *Modern Chess Strategy*. London: Bell.
- Lasker, Emanuel (1932). *Lasker's Manual of Chess*, 2nd ed. London: Constable; Dover reprint, 1960.
- Luce, R. D., & Raiffa, H. (1957). *Games and Decisions*. New York: Wiley.
- Macdonald, D. K. C. (1950). Contribution to the discussion on Slater (1950), p. 198.
- Maynard Smith, J., & Michie, D. (1961). Machines that play games. *New Scientist*, 12, 367-369.
- McCulloch, W. (1965). *Embodiments of Mind*. Cambridge, Massachusetts: M.I.T. Press.
- Michie, D. (1966). 'Game-playing and game-learning automata', with an appendix by J. Maynard Smith; in *Advances in Programming and Non-Numerical Computation*, pp. 183-200. Oxford, etc.: Pergamon Press.
- Neumann, J. von, & Morgenstern, O. (1944-47). *The Theory of Games and Economic Behaviour*. Princeton: University Press.
- Newell, A., Shaw, J. C., & Simon, H. A. (1958, 1963). Chess-playing programs and the problem of complexity. *IBM J. Res. Dev.* 2, 320-335, 1958. Reprinted in *Computers and Thought* (see Samuel 1963), 39-70.
- Newell, A., & Simon, H. A. (1961). Computer simulation of human thinking. *Science*, 134, 2011-2017.
- Newell, A., & Simon, H. A. (1965). An example of human chess play in the light of chess playing programs. *Progress in Biocybernetics*, pp. 19-65. Amsterdam, London, New York: Elsevier.
- Philidor, F. A. D. (1749). *L'Analyse des Échecs*. London.
- Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, 3, 210-229.

MACHINE LEARNING AND HEURISTIC PROGRAMMING

- Samuel, A. L. (1963). Reprint of Samuel (1959), with an additional appendix. In *Computers and Thought*, E. A. Feigenbaum and J. Feldman (eds.), pp. 71-105. New York, etc.: McGraw-Hill.
- Shannon, C. E. (1950). Programming a computer for playing chess. *Phil. Mag. (London) 7th Ser.*, **41**, 256-275.
- Slater, Eliot (1950). Statistics for the chess computer and the factor of mobility. *Symposium on Information Theory*, pp. 150-152. London: Ministry of Supply.
- Suetin, A. S. (1965). *Modern Chess Opening Theory* (tr. from the Russian by D. J. Richards, ed. by P. H. Clarke. Oxford, etc.: Pergamon; date of the original Russian suppressed).
- Turing, A. M. (1953). Digital computers applied to games, *Faster than Thought*, B. V. Bowden (ed.), pp. 286-295. London: Pitman.
- Vukovic, V. (1965). *The Art of Attack in Chess* (tr. by A. F. Bottrall, ed. by P. M. Clarke. Oxford, etc.: Pergamon).

8

NEW DEVELOPMENTS OF THE GRAPH TRAVERSER

JAMES DORAN

EXPERIMENTAL PROGRAMMING UNIT
DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION
UNIVERSITY OF EDINBURGH

INTRODUCTION

This paper describes some recent experiments with a computer program which is capable of useful, or at least interesting, application to a number of different problems. The program, the Graph Traverser, has been described in detail in a previous paper (Doran & Michie 1966). However, we shall here need to view the basic algorithm from a rather more general standpoint, corresponding to an actual extension in the flexibility of the program, so that a restatement of what the program can do is desirable.

The Graph Traverser, which is written in Elliott 4100 Algol, is potentially applicable to problem situations which can be idealised in the following way (see for comparison Newell and Ernst 1965). There is given a set of 'states', which are connected by a set of 'transformations', or, as I shall call them, 'operators'. An operator will be applicable to some, but not necessarily all, of the states and two distinct operators applied to either the same or distinct states may each give the same state as end-product. Most of the concepts to be used here which are related to the use of operators were discussed in a paper by Michie (1967).

This type of problem situation is represented in Fig. 1 by a graph (in the mathematical sense) to which have been added various labels. In this representation states correspond to nodes of the graph, and operators to labelled arcs—*a*, *b*, *c* in this quite arbitrary case. Notice that associated with each node (or state) is a triad of integers. These represent the way in which the

program handles the 'structure' that is typically associated with a state, for with each node it holds an integer matrix, the dimensions of which will depend upon the particular problem. It is this structure which embodies the additional information which the program uses to carry out a heuristically directed search for a solution rather than a systematic search.

What 'idealised' problems can be posed in this context? It is convenient to recognise two types (for a fuller classification see Doran & Michie 1966). These we may call informally 'state-search' problems and 'path-search' problems. In a state-search problem we wish to find a state with some particular property. For example, and this will be relevant later on, we might seek the state for which some associated cost was smallest. In Fig. 1 we might seek

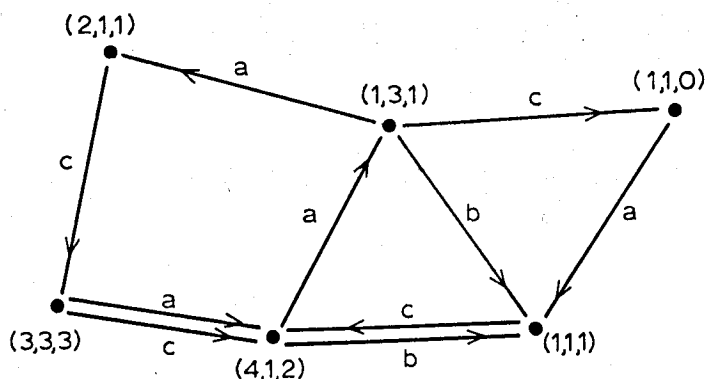


FIG. 1. An example of a problem graph. The nodes correspond to problem states, and the arcs to operators (transformations).

the node for which the sum of the triad of integers was least. In a path-search problem we seek a path between two states, that is, a sequence of operators which take us from one state to another. Referring again to Fig. 1, we might require a path from the node (3, 3, 3) to the node (1, 1, 0). A solution would be (a, a, c). Note that the sequence of states (3, 3, 3), (4, 1, 2), (1, 3, 1), (1, 1, 0) does not immediately give the corresponding operators (though the converse is true) and thus is a solution of a weaker kind. To appreciate this point the reader must realise that although in Fig. 1 he can see the whole of the problem graph laid out before him, the program must work from the definitions of the states and the operators, and the application of the latter to the former.

Before describing the Graph Traverser search method, it must be emphasised that these problems are always to be regarded from a practical rather than a theoretical standpoint. Actual solutions must be found to actual problems within a reasonable search time. A further point is that in a given problem situation the way in which the states and operators are defined is likely to be of great importance. Although this aspect of the matter is outside the present program schema, it must not be overlooked. In particular, for a state-search problem, the definition of the operator set may be an important part of the method of solving a problem, rather than of the problem situation itself.

THE GRAPH TRAVERSER SEARCH METHOD

The program proceeds by selecting an initial state either as directed or at random or, in a path-search problem, from the specification of the problem, and then by applying operators to this state and to the new states it thus generates, until the fragment of the problem graph which it has built up provides the information needed to find the solution. The structure which is constructed by the program is a 'tree', since although the program may generate a state more than once, it will only store it once, and will only remember one path to it.

When growing its search tree the program repeatedly encounters two problems. The first is to decide to which state next to apply operators. The second is, having chosen a state, to decide which operators actually to apply. The program's reaction to the first problem involves the 'evaluation function', and to the second the 'develop procedure'.

Briefly, the evaluation function must assign to each state encountered a value based on its structure. When a state must be selected for development (i.e., application of operators) that with the smallest value (smallest rather than largest since in path-search problems the value may often be identified with estimated distance from the 'goal') is chosen, avoiding states which have already been fully developed.

When a state has been chosen for development the develop procedure must decide two things, (i) which operators of those applicable actually to apply (this is the process of *operator selection*) and then having applied them (ii) whether or not to mark the state as fully developed. If a state is marked as fully developed then it will never be chosen for development again. If it is not so marked then the program leaves itself the option of taking up development of that state again. A state such as this which is neither quite undeveloped nor fully developed we call *partially developed*.

Perhaps the simplest thing to do when developing a state is to apply all possible operators and be done with it. The earliest work with the Graph Traverser used this approach, which does not involve partial development. At the other extreme, at most one operator may be applied in a development, the order of application being fixed but arbitrary, and the state being considered as partially developed until the last operator has been applied. This approach, which has an important application to be discussed later in the paper, is illustrated in Fig. 2, which shows the program searching the problem graph of Fig. 1. The arrows are reversed corresponding to the actual structure of the search tree. The task is to find the node the sum of whose integers is least. The program arbitrarily selects (4, 1, 2) as its starting point. As indicated above it applies just one operator during a development, and will not mark a node fully developed until all the operators which are applicable there have actually been applied. The order of application is always *a, b, c*. The value assigned to a node by the evaluation function is just the sum of its integers. The letters U, P, F indicate the status of a node—undeveloped, partially developed, or fully developed.

MACHINE LEARNING AND HEURISTIC PROGRAMMING

It is a consequence of this algorithm that the program carries out repeated developments of a node until either a "descendant" node of lower value is found, or no more applicable operators are available. In the latter event a "disconnected" development will ensue. Notice in (f) of Fig. 2 that when a

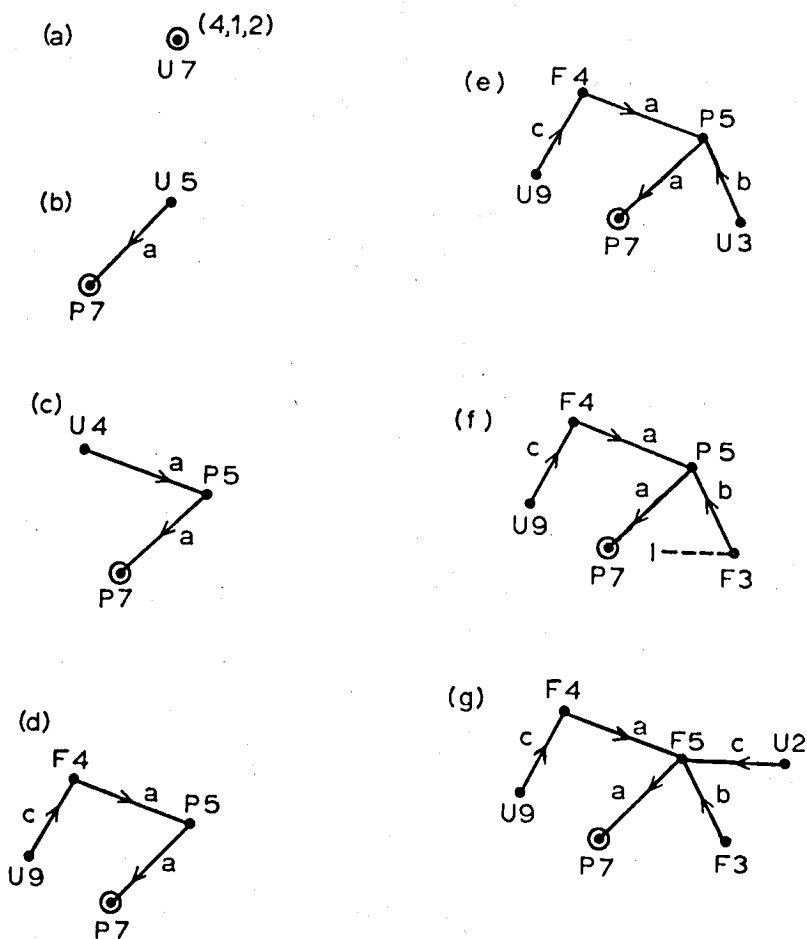


FIG. 2. The Graph Traverser searching the graph of Fig. 1. Successive diagrams show successive developments. The node from which the search starts is ringed for clarity. The letters U, P, F indicate the status of a node—undeveloped, partially developed, or fully developed. The figures give the values calculated for the nodes. The arcs are reversed since in the search tree the program stores a pointer to the ancestor of a node, not pointers to its descendants.

node already on the tree is generated its regeneration is ignored. Had the new encounter provided a shorter path to the node than that already known, the tree would have been modified accordingly. The search ultimately succeeds, but whether the best node would be found in practice depends upon the particular rule used to decide when to terminate the search.

This brings us to the most important point that these various decisions which the program must make depend heavily upon guidance from the user in a particular problem situation, although ways of making the program itself improve its decisions have been suggested elsewhere (Doran 1967, Michie 1967). In practice the develop and evaluate procedures, and any other routines which need to be respecified, are programmed for each application and embedded within the program proper.

THE TRAVELLING SALESMAN PROBLEM

The Travelling Salesman is one of the best known problems of operations research. It is very simply stated. A salesman, starting at a given city must visit $n-1$ other cities, each just once, before returning to base. There is a fixed distance to travel between any particular pair of cities and he wishes the whole tour to be as short as possible. The problem is to specify an efficient way for him to find the shortest tour. The word 'efficient' here is crucial. One can find the shortest tour by calculating the length of all possible tours. But if there are n cities, there are $\frac{(n-1)!}{2}$ possible tours, so that this approach is usually much too laborious.

This version of the problem corresponds to marking a score or so points on a piece of paper, and connecting them up so that the tour goes through each just once, and is as short as possible in length. More generally, we stop thinking in terms of cities or points on a plane surface, and simply say that a cost is assigned to travelling between each pair of 'cities' (independent of direction) in some way which does not concern us.

Linear programming and dynamic programming methods can be applied to the Travelling Salesman problem but only with difficulty, particularly if the number of cities involved is more than 20, say. Yet the problem is one of real importance, and in modified and disguised forms occurs repeatedly in industrial and economic situations. We may, for example, think of ships visiting ports, or less obviously, of an automatic press punching a series of holes in specified locations in pieces of sheet metal.

Shen Lin (1965) and G. M. Roe (1966) have recently explored with considerable success heuristic search methods which provide near-optimal tours, but avoid spending much time trying to turn a near-optimal tour into an optimal tour. By an 'optimal tour' I mean any tour with least possible cost.

Lin's approach, which is the one I shall discuss here (Roe's is broadly similar), defines for the Travelling Salesman problem a certain set of, in our terminology, operators. These are called (using Roe's terminology now) 'cuts'. The set of all n -cuts is the set of all possible ways of breaking any n links of a given tour and putting it together again, as a single loop, in some other way. In particular, a 2-cut involves reversing the direction in which a sequence of the cities is visited, and a 3-cut involves removing a sequence of cities from the tour and then reinserting this sequence, possibly reversed, at the same or some other point. If there are, say, 20 cities in a tour then there are 170 possible

2-cuts, and about 5000 possible 3-cuts. Lin uses 3-cuts (his experiments indicate that 3 is the best value of n for this purpose) to convert the set of all possible tours into a problem graph in the Graph Traversal sense, and then adopts an algorithm now to be described.

Lin's algorithm involves a number of 'searches'. In each search an initial tour is selected at random, and the 3-cuts are systematically checked until one which will generate a new tour with lower cost has been found. This is then applied. The process is repeated with the new tour until an improvement

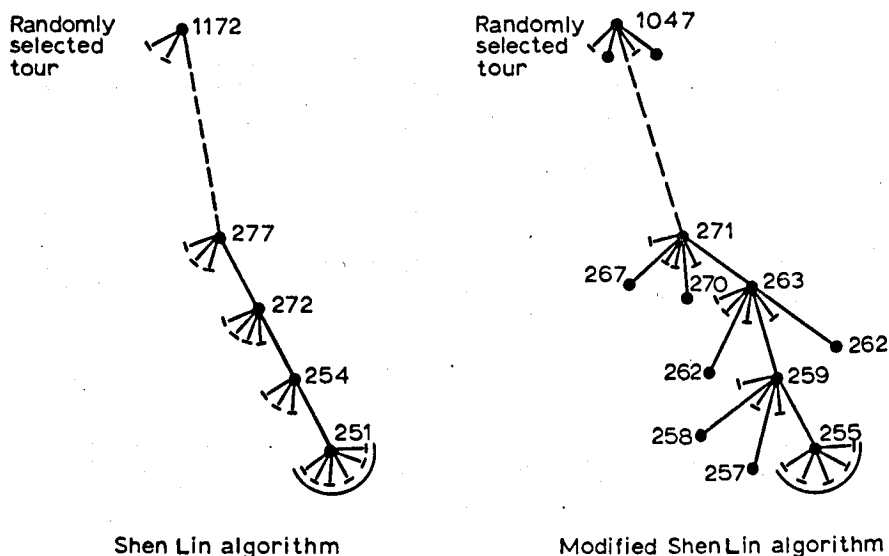


FIG. 3. Shen Lin's algorithm for solving the Travelling Salesman problem, and a modification of this to permit three new tours to be generated at each development. The nodes correspond to possible tours, and the figures give the corresponding costs. The blocked arcs indicate 3-cuts which do not reduce the cost. Both algorithms terminate when a tour is encountered which no 3-cut improves (a 3-opt tour).

is again obtained, and so on until ultimately a tour is reached which no 3-cut improves (Fig. 3). This tour is then the 'locally optimal tour' or '3-opt' tour associated with the initial random starting configuration. In all tens of searches will be made. Experiment shows that for all except the smallest problems, not all, and possibly none, of the 3-opt tours obtained will be optimal, nor in general will they be all distinct.

However, as Lin points out, after some experience with the algorithm it is possible to estimate in advance for an unsolved problem of particular size the probability that a search will give an optimal tour. It is then possible to calculate how many searches are necessary to reduce to a level low enough to be negligible the probability that an optimal tour still has not been discovered. In this situation it is not necessary to push on with an individual search until an optimal tour has definitely been located—a process likely to be both

inconclusive and very time consuming. Lin's program is for an IBM 7094 II, and is very fast, most relevant times being measured in seconds.

Lin also employs various techniques to use information collected in the earlier searches to cut computation in the later. These techniques will not concern us here but are interesting and important.

APPLICATION OF THE GRAPH TRAVERSER TO THE TRAVELLING SALESMAN PROBLEM

The Graph Traverser has been applied to the Travelling Salesman problem with two main objectives:

- (i) to establish that it *is* applicable to this situation, and that all the basic results obtained by Lin can be obtained using the Graph Traverser, though in practice at a slower rate since the Graph Traverser aims at generality rather than speed; and
- (ii) to use the flexibility and facilities of the Graph Traverser to explore the worth of other search methods more complex than that used by Lin.

The actual mapping process which takes Lin's algorithm into that of the Graph Traverser has already been indicated. A state of the problem graph corresponds to a possible complete tour (*not* to a city), and an operator to a 3-cut as already described. All the 3-cuts are applicable to any tour, but the develop procedure when called only actually applies one 3-cut—the first one which it considers which will, it calculates, generate a cheaper tour. The cuts are considered in an arbitrary but fixed order. To implement Lin's strategy, the develop procedure then marks the tour as fully developed. The effect of this is that when a 3-opt tour is encountered, so that no new undeveloped tour is generated before the current tour is declared fully developed, the search must terminate (Fig. 3).

Using the Lin algorithm which is an example of a 'conditional choice strategy' (see Michie 1967) the Graph Traverser was applied to randomly generated problems involving up to 20 cities, and to two published problems, the 20-city problem of Croes (1958), and the 25-city problem of Held and Karp (1962). The results obtained were consistent with those obtained by Lin. For the Croes problem Lin quotes 13 out of 40 for the proportion of successful searches, and for the Held and Karp problem 26 out of 40. The corresponding figures using the straightforward Graph Traverser implementation of Lin's algorithm were 3 out of 10 and 13 out of 16. A typical search on the 20-city problem involved developing about 45 tours, and on the 25-city problem about 70 tours. These results indicated that the first research objective had been achieved. An Elliot 4120 processor was used to obtain these and the other results described in this paper.

MODIFICATION OF SHEN LIN'S ALGORITHM

How can Lin's algorithm be improved? From the standpoint of the Graph Traverser program it is a rather simple strategy, and presents a number of opportunities for modification.

The first modification which suggested itself was to require the develop procedure to produce, where possible, more than one cheaper tour. Since the cost of a tour is being used as its value, the program will always select the cheapest descendant tour for the next development. This trades the extra computation time required for each development against, hopefully, a more successful search. In practice, three descendant tours were generated at each development (see Fig. 3). Table 1, which refers to the Croes problem, shows

TABLE 1

Costs of 3-opt tours obtained with the Graph Traverser for a particular Travelling Salesman problem. The three algorithms used are described in the text. In every case tours with the same cost are identical.

CROES 20-CITY PROBLEM

OPTIMAL TOUR HAS COST 246

Random starting tour	Lin algorithm	Modified Lin algorithm	Graph Traverser algorithm
A	246	246	246
B	251	246	251
C	252	252	246
D	252	252	246
E	252	255	246
F	246	252	246
G	246	252	246
H	252	246	246
I	252	252	246
J	251	252	246
Occurrences of optimal tour	3	3	9
Approximate time for each search in minutes	15-20	20	30

that each search took rather longer with no noticeable improvement in the results. This agrees with Lin's remark that 'Attention should be directed to finding improvements with a minimum amount of computation rather than to making the maximum improvement possible at each step'. What seems to be happening is that there are only a very few 3-opt tours, all quite similar, and which of these is encountered during any particular search depends only upon the final 'direction' of approach to the critical region. What happens earlier is quite irrelevant.

The mutual proximity of the 3-opt tours, in the sense that one can be converted into another by a very few 3-cuts (not, of course, by just one) suggests that terminating a search as soon as a 3-opt tour is encountered may be premature, and that a certain amount of additional investigation might be worthwhile—even though such additional investigation is likely to be very costly in time. Roe suggests something along these lines.

It seems appropriate to invoke the Graph Traverser's ability to partially develop tours, leaving them available for further development at a later time.

This process was illustrated in detail in Fig. 2. Specifically, instead of the develop procedure applying the first 3-cut it encounters which will reduce the tour cost, and then marking the tour as fully developed, this latter step is omitted unless there exists no unused beneficial operator at all. In consequence, when a 3-opt tour is found, the search does *not* terminate, but continues by restarting the development of the tour from which the 3-opt

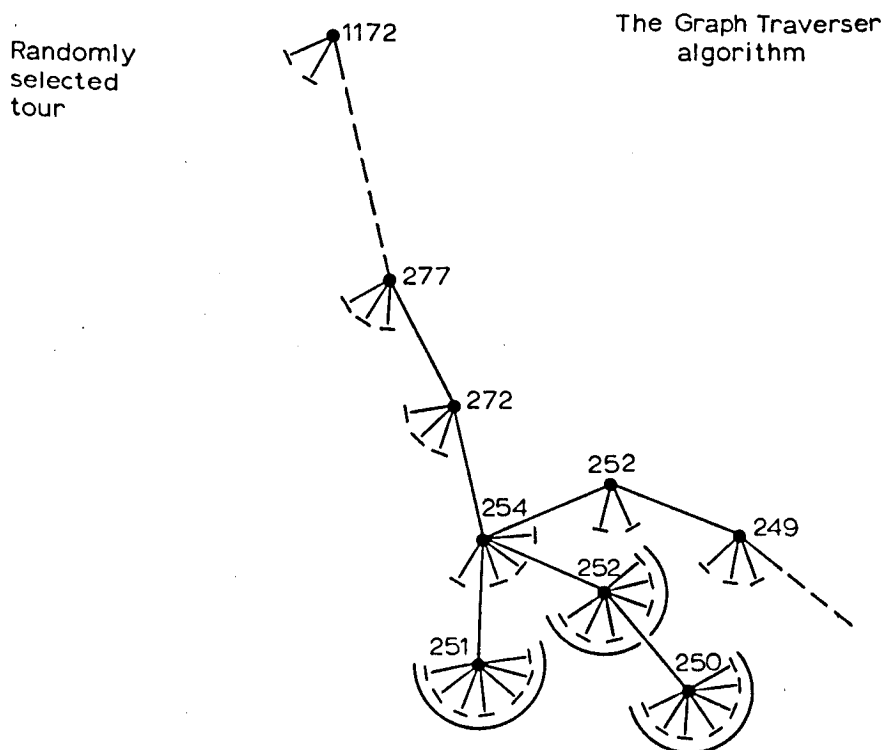


FIG. 4. Shen Lin's algorithm augmented by the use of the Graph Traverser's partial development facility. A search is continued beyond the discovery of a 3-opt tour by continuing the development of the best partially developed tour. The branches are explored from left to right, and the search is terminated when the program has been forced to 'retreat' a specified number of times.

tour was itself generated (Fig. 4). The search is continued in this way until some fixed number of tours has been fully developed. Since each full development corresponds to the program finding itself forced to 'retreat' to a tour of greater cost, this is a convenient and reasonable way to terminate a search. Of course, the 'result' of such a search is the cheapest tour encountered, not merely the last encountered.

EXPERIMENTAL RESULTS

Table 1 shows results obtained by this revised algorithm. For this problem, of which Lin remarks that it 'seems harder to solve than most 20 city prob-

lems', it almost uniformly succeeds in finding the best tour when the Lin algorithm fails. A search was terminated at the fourth retreat, which roughly doubled the time required for it.

Thus, although the average effectiveness of the searches has been increased, the time required for each has also risen. To decide whether there has been an overall improvement in performance we look at the matter in terms of the question 'How long must we run the program on this problem to achieve some selected probability that we have obtained an optimal tour?' Using the Lin algorithm the best estimate of the probability that a particular randomly initiated search will produce an optimal tour is clearly $p_1=0.3$. It follows that the number of searches required to ensure a probability of 0.05 or less, say, that an optimal tour has not been found is found by solving the inequality in N_1

$$(0.7)^{N_1} \leq 0.05$$

This gives $N_1 \geq 9$. Supposing unit time for each search, the time required is 9 units.

Using the Graph Traverser strategy the estimated probability of a search succeeding is $p_2=0.9$. The corresponding calculation gives $N_2 \geq 2$. Since the search time doubles, the time required is 4 units. Thus the time required is reduced by a factor of more than 2. The reader will easily see that this factor is broadly independent of the probability of error initially specified.

Thus the conclusion to be drawn from this admittedly very small quantity of evidence is that the partial development extension of Lin's algorithm is worthwhile.

Further investigation of this conclusion required results gathered from a range of problems. However this led to a snag. Fairly *difficult* problems were required, for otherwise Lin's basic strategy leaves little room for improvement. Suitably difficult problems were too large, however, for the limited machine store size immediately available.

An experiment was therefore carried out using 15 city problems defined by randomly generated cost matrices, and using *2-cuts* as operators rather than *3-cuts*. The *2-cuts* are a smaller, less drastic set of operators which Lin found to be less powerful in practice even allowing for the sharp reduction in the time required for each search when using them. Thus small problems which would otherwise be too easy are made suitably difficult by reducing the effectiveness of the operator set.

Eleyen random 15-city problems were generated, and ten searches were carried out on each, each search being continued until the 30th retreat occurred. Table 2 summarises the results obtained. The following points are important:

1. Column 2 gives the average number of cuts applied or considered for application by the program before reaching the n th retreat in the search, where n is indicated by the figure in column 1. Just over 300 operations are needed to reach the first retreat—the point at which Lin's search would

terminate—and thereafter just under 100 are on average needed to reach each successive retreat. These figures are effectively a measure of *time*.

2. Column 3 gives the mean cost of the best tour obtained up to the n th retreat and column 4 the observed probability that an optimum cost tour has already been found. What was the cheapest cost was decided by inspection of all the results obtained for the given problem. In no case was the matter in doubt. We see that the probability that a Lin search would be successful is about 0.09.

Note that there appears to be a sharp improvement if the search is carried only a little beyond the point where Lin terminates.

TABLE 2

Results obtained by applying the Graph Traverser algorithm to eleven randomly generated 15-city problems. The means are calculated over 110 searches in all. The final column refers to a 'typical' problem of this type and envisages several searches each terminating after a fixed number of retreats. The basic Lin algorithm would halt at the first retreat.

Retreat at which search ends	Mean number of operations per search	Mean cost of cheapest tour found in search	Observed probability that optimal-cost tour has been found in search	Total number of operations needed to give 0.99 probability of optimal tour
1	317	201	0.09	15316
2	416	193	0.22	7783
3	517	190	0.31	6439
4	605	187	0.34	6795
5	704	186	0.36	7172
6	798	185	0.36	8130
7	902	184	0.38	8636
8	995	183	0.43	8221
9	1090	182	0.47	7842
10	1176	181	0.49	8021
15	1651	179	0.55	9643
20	2105	178	0.60	10579
25	2570	177	0.65	11413
30	3025	177	0.68	12165

3. The last column attempts to combine columns 2 and 4 to discover if the loss in time is balanced by the gain in power. The figures give the number of operations, that is the number of inspections or applications of 2-cuts, needed to give a probability of 0.99 that an optimum tour has been located, assuming that we are faced with a new 15-city problem generated in the same way. The total number of operations required with termination on the r th retreat is $-2i_r/\log(1-p_r)$ where p_r is the estimated probability that an optimum tour will be found during a search thus terminated (column 4), and where i_r is the observed average number of inspections up to the r th retreat (column 2). These figures are artificial in two ways. First they assume that a non-integral number of searches is meaningful, in effect some smoothing has

been carried out, and secondly, and more important, it is assumed that all the problems generated by the random process are of equal difficulty. This latter assumption is certainly false. However, there is no reason to think that this will cause the figures to be misleading for our purposes.

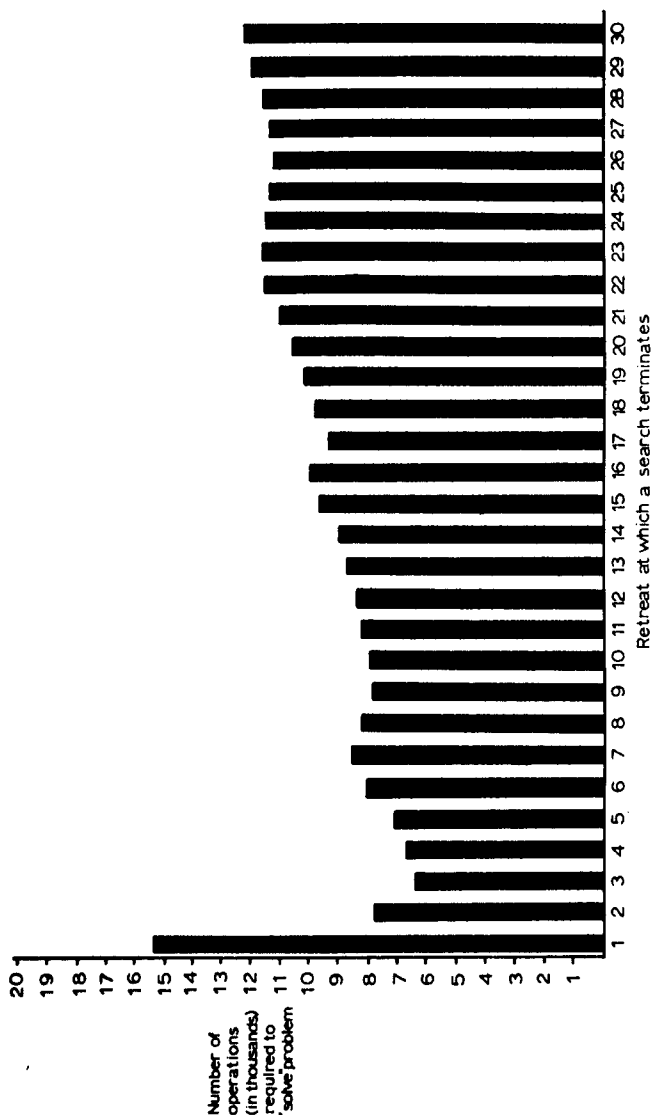


FIG. 5. Diagram showing the variation in the number of basic operations needed to 'solve' a typical randomly generated 15-city problem with the number of 'retreats' permitted in a search. A problem is said to be 'solved' when the probability that an optimum cost tour has been found exceeds 0.99. If the search is terminated at the first retreat then this gives Lin's algorithm.

Fig. 5 shows the required number of operations plotted against the length of search permitted. Again there is striking evidence in favour of continuing searches beyond the point at which Lin would terminate, and we see that the best results are obtained when a search is terminated at the third retreat. The apparently rhythmic fluctuation has no significance.

To sum up, although work is required on larger and more varied problems, it does appear that the Graph Traverser search method is capable of improving significantly Lin's already very successful algorithm. The improvement is obtained by continuing a search beyond the moment at which the first retreat occurs, that is, spending time searching the neighbourhood of locally optimal tours and proportionately reducing the total number of searches carried out.

Put another way, which may be more illuminating, the modification to Lin's algorithm is to start a proportion of the searches (as he defines them) from tours already known to be good, rather than always to start searches from randomly generated tours. This follows from the fact that when a retreat occurs, and development of an already partially developed tour is restarted, this is virtually equivalent to beginning a new search from that tour.

ON-LINE COLLABORATION WITH THE GRAPH TRAVERSER

It seems probable that in the foreseeable future problem-solving programs will be designed to interact with the user, rather than operate in isolation, when they are really intended to be useful. Such an arrangement permits the user to support the program in those aspects of the problem-solving procedure where it cannot cope adequately by itself. I shall therefore end this paper with a discussion of a version of the Graph Traverser adapted for such collaboration in the context of two simple sliding block puzzles, the Eight-puzzle and the Fifteen-puzzle, and with some more general remarks.

The Eight- and Fifteen-puzzles have been described in detail elsewhere (e.g., Doran & Michie 1966, Schofield 1967). Briefly, the Eight-puzzle consists of eight unit square pieces in a 3×3 frame, the pieces being numbered from 1 to 8, and there being one empty cell. The problem is to convert one configuration of the pieces into another by sliding the pieces around. Thus we might seek a sequence of *moves*, where a move means sliding one piece into the adjacent space, which converts

7	5	4		1	2	3
1	0	8	into	8	0	4
2	6	3		7	6	5

Provided that the two configurations regarded as permutations of the digits 0 to 8 have the same parity, at most 30 moves will be required.

The Fifteen-puzzle is the same puzzle scaled up to a 4×4 frame, and with the pieces numbered from 1 to 15. For this puzzle, although the same parity rule holds, the maximum number of moves needed to convert one configuration into another is not known.

Extensive results have already been published from the application of the Graph Traverser to these puzzles (Doran & Michie 1966, Doran 1967, Michie 1967). The problem is of the 'path-search' variety, unlike the Travelling Salesman application, and the evaluation function plays a much more important role, for it has the difficult task of estimating how close con-

figurations generated in the search are to the target configuration. As a result of this earlier work, fairly good evaluation functions are available for the puzzles.

A very simple way to bring a human solver into the scheme is to arrange that he takes over the function of the develop procedure, and this has been implemented. Every time the program wishes to develop a configuration the

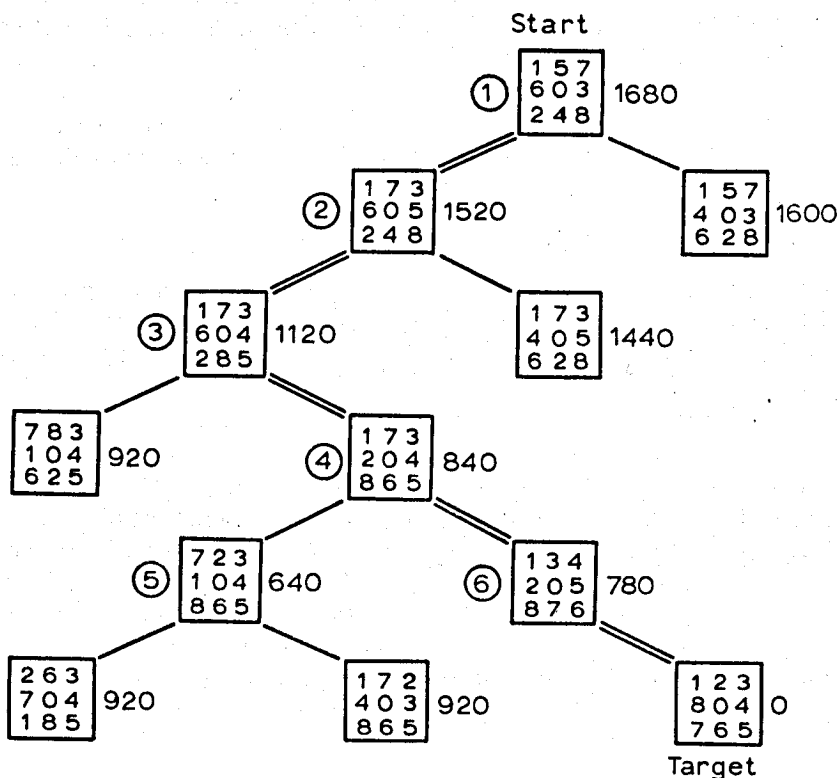


FIG. 6. The search tree constructed collaboratively by the Graph Traverser and the author to solve a particular Eight-puzzle configuration. The ringed figures give the order of development, and the unringed figures the values assigned by the evaluation function. Notice that after the fifth development the program rejects both the macro-moves suggested by the author in favour of a continuation previously itself rejected.

human solver is presented with it, and asked to specify sequences of moves (operators) which he suggests be applied. Once he has suggested all the moves or move sequences he thinks reasonable, he indicates this, the moves are implemented, and the configuration is henceforth regarded as fully developed. Thus there is no partial development.

From the user's point of view the sequence of events is as follows. The program asks whether it is the Eight-puzzle or the Fifteen-puzzle which is to be solved. Given the answer that it is the Eight-puzzle, say, the program then types out the standard goal configuration and asks the user to type the

starting configuration. This done, the program asks the solver what moves he suggests for this configuration. The solver might suggest three possible sequences of moves, and then type ALL. The program implements the moves, decides which of the three resulting configurations it favours, and presents this to the user for more moves to be suggested. This continues until either the goal is located, when the program will output the solution path with relevant statistics, or the solver types RESIGNS. The program will ignore impossible moves or typing errors.

The configuration selected by the program for 'human' development will not necessarily be the outcome of one of the moves or move sequences last suggested by the solver. As always with the Graph Traverser, disconnected developments are possible. That is, if none of the current suggestions of the solver is sufficiently attractive, the program will return to an earlier possible continuation not thought the best at the time. This point is illustrated in Fig. 6, which shows the search tree built up by the author and the 'on-line' Graph Traverser to solve a particular Eight-puzzle configuration. The nodes correspond to configurations and are labelled with the value assigned to them by the program, and with an integer indicating the order in which the developments occurred. The final solution path involved 26 basic moves, compared with a minimum possible of 22. In a parallel attempt on this configuration *not* making use of the program the author required 36 moves to solve it. Indeed, although no serious experimentation has been attempted, the author usually seems to benefit from the program's assistance, although whether the converse is true is less clear!

DISCUSSION

This example of a collaborative use of the Graph Traverser is of no great importance in itself, but it does serve to introduce some further ideas. Dr Donald Michie has pointed out to me that Travelling Salesman problems could be solved by an analogous collaborative approach. The user would be presented with a tour to which he would suggest modifications, and the program would next present either a tour generated by one of the suggested modifications, or a tour generated earlier in the search, but not then judged sufficiently attractive to be followed up.

More generally, a version of the program might be prepared in which the function of both the develop procedure *and* the evaluate procedure were taken over by an on-line user. Since all other aspects of the program's behaviour can be controlled by setting parameters, which even in the current version of the program can be done on-line, such a program would be capable of being applied, *without additional programming*, to a wide range of problems certainly including both of the puzzles mentioned and the Travelling Salesman. However, although this would certainly achieve generality, so much of the work load would have been transferred to the user and progress would be so slow, that the arrangement is unattractive.

What facilities would an on-line user of the Graph Traverser ask for?

Certainly he will not wish to do any preparatory programming. He will wish to enter the program, to specify to it the task, to give it such guidance as he can, and then either to withdraw entirely or to remain on hand to answer questions which the program may ask. He will often wish to monitor the program's progress and to change the problem specification, or to give additional guidance where appropriate. In Graph Traverser terms this means that the user must at least be able initially to specify the operator set, that is the develop procedure, and the evaluation function, and then to change his specification at will. Some kind of on-line problem-oriented programming language, however primitive, seems essential. This could be provided within Algol, but neither easily nor efficiently. It seems obvious that such a version of the Graph Traverser should be wholly written in a language designed for on-line usage. POP-1, created by R. J. Popplestone and described elsewhere in this volume, is attractive given the ease with which it permits the construction of a library of sub-routines.

Other papers already mentioned have suggested ways in which the Graph Traverser might itself improve its search apparatus, usually by some form of parameter optimisation, and thus be a more 'intelligent' problem-solver. D. Vigor, in his stimulating paper included in this volume, emphasises another aspect of intelligence by stressing the importance not of the way in which a program *attacks* a problem (or general game) situation, but of the way in which the program *represents* the situation. He suggests that very often this is the most important step in the solution process, and outlines a method for seeking the most efficient problem representation. I mention this because it indicates an aspect of problem-solving which, although unrecognised by the present Graph Traverser, must surely be important in future work.

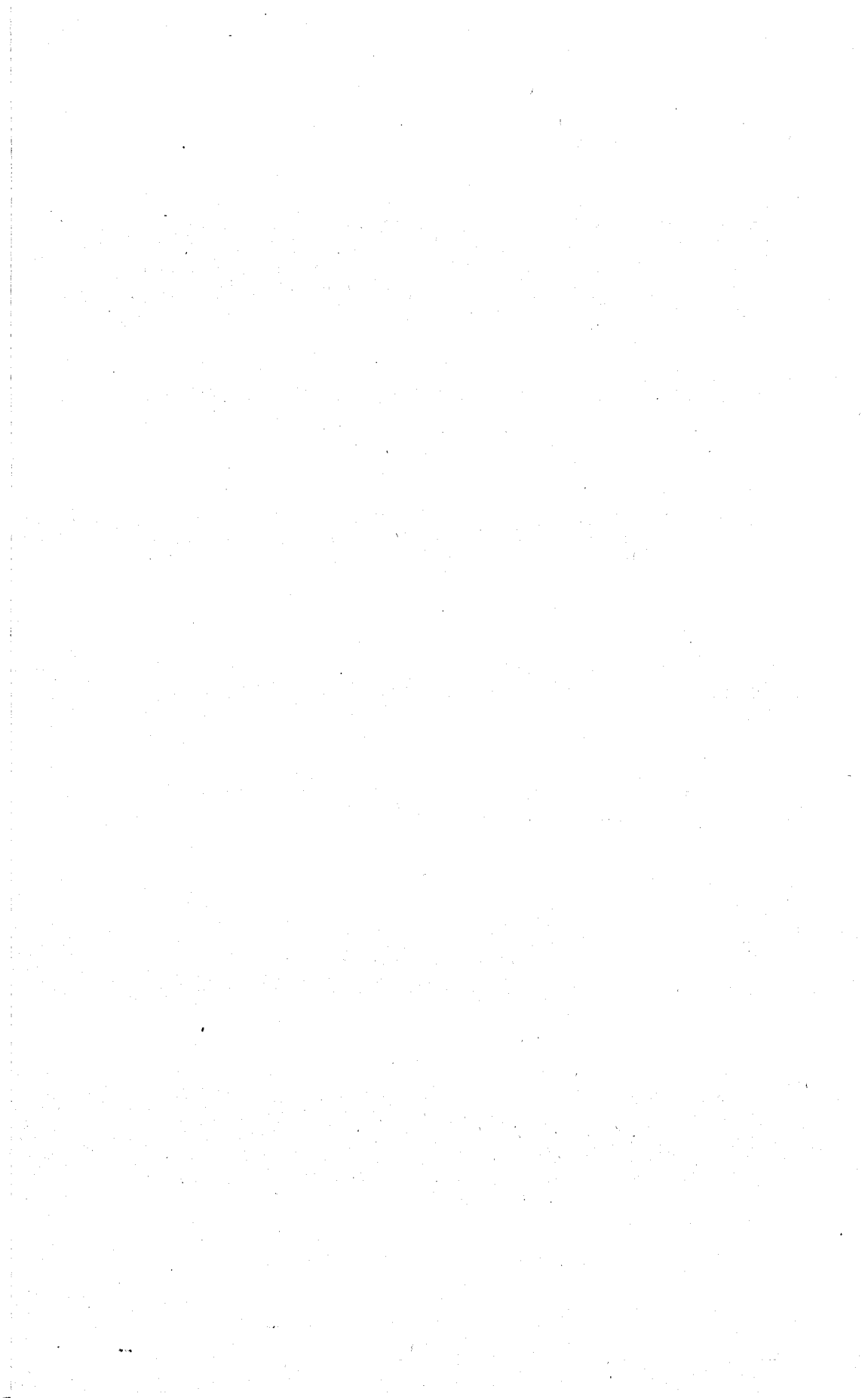
ACKNOWLEDGEMENTS

The research described in this paper is sponsored by the Science Research Council. I am greatly indebted to Dr Donald Michie for many discussions on the topics covered by this paper and, in particular, for his original suggestion that the Graph Traverser might be applicable to the Travelling Salesman in the manner described.

REFERENCES

- Croes, G. A. (1958). A method for solving Traveling Salesman problems. *Operations Research*, 6, 791-812.
- Doran, J. E. (1967). An approach to automatic problem-solving. *Machine Intelligence* 1, pp. 105-123. Collins, N. L., and Michie, D. (eds.). Edinburgh: Oliver & Boyd.
- Doran, J. E., & Michie, D. (1966). Experiments with the Graph Traverser program. *Proc. R. Soc. (A)* 294, 235-259.

- Held, M., & Karp, R. M. (1962). A dynamic programming approach to sequencing problems. *J. Soc. Ind. Appl. Math.* **10**, No. 1, 196-210.
- Lin, S. (1965). Computer solutions of the Travelling Salesman problem. *Bell System Tech. J.*, **44**, 2245-2269.
- Michie, D. (1967). Strategy-building with the Graph Traverser. *Machine Intelligence* 1, pp. 135-152. Collins, N. L., and Michie, D. (eds.). Edinburgh: Oliver and Boyd.
- Newell, A., & Ernst, G. (1965). The search for generality. In *Information Processing 1965: Proceedings of IFIP Congress 1965*. Vol. 1, pp. 17-24. Wayne A. Kalenich (ed.). London: Macmillan.
- Roe, G. M. (1966). Three fast, suboptimal procedures for the Traveling Salesman problem. *General Electric* report No. 66-C-051.
- Schofield, P. (1967). Complete solution of the Eight-puzzle. *Machine Intelligence* 1, pp. 125-133. Collins, N. L., and Michie, D. (eds.). Edinburgh: Oliver and Boyd.



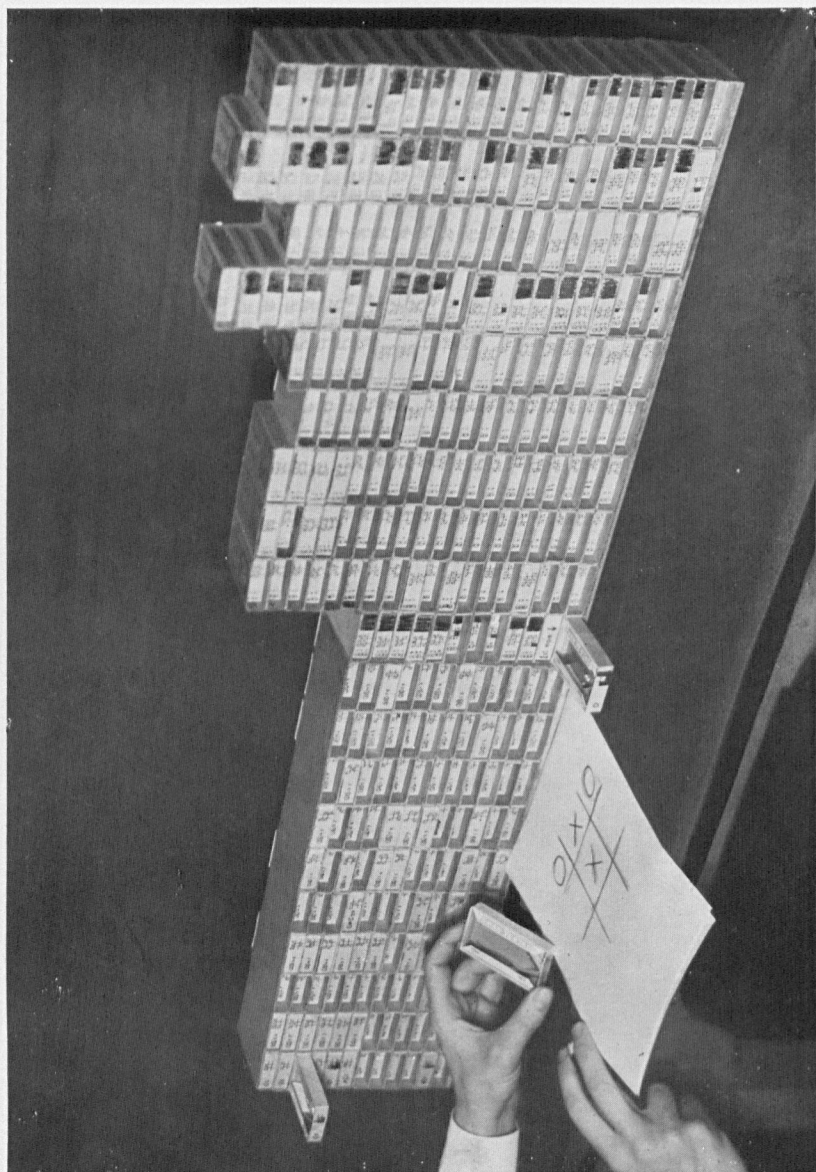


FIG. 1. The original matchbox version of MENACE

9

BOXES: AN EXPERIMENT IN ADAPTIVE CONTROL

D. MICHIE and R. A. CHAMBERS

DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION
UNIVERSITY OF EDINBURGH

BOXES is the name of a computer program. We shall also use the word boxes to refer to a particular approach to decision-taking under uncertainty which has been used as the basis of a number of computer programs.

Fig. 1 shows a photograph of an assemblage of actual boxes—matchboxes to be exact. Although the construction of this Matchbox Educable Noughts and Crosses Engine (Michie 1961, 1963) was undertaken as a 'fun project', there was present a more serious intention to demonstrate the principle that it may be easier to learn to play many easy games than one difficult one. Consequently it may be advantageous to decompose a game into *a number of mutually independent sub-games* even if much relevant information is put out of reach in the process. The principle is related to the method of subgoals in problem-solving (see Newell *et al.* 1960) but differs in one fundamental respect: subgoals are linked in series, while sub-games are played in parallel, in a sense which will become apparent.

DECOMPOSITION INTO SUB-GAMES

The motivation for developing algorithms for small games (by a 'small' game we mean one with so few board positions that a boxes approach is feasible) needs explanation, since small games are generally too trivial to be of intellectual interest in themselves. The task of *learning* a small game by pure trial and error is, on the other hand, not trivial, and we propose that a good policy for doing this can be made useful as a component of a machine strategy for a large game. The point is that the board states of a large game may be mapped

on to those of a small game, in a many-to-one mapping, by incomplete specification. This is what the chess player does when he lumps together large numbers of positions as being 'similar' to each other, by neglecting the strategically irrelevant features in which they differ. The resultant small game can be said to be a 'model' of the large game. He may then, in effect, use his past experience of the model to select broad lines of play, and in this way guide and supplement his detailed analysis of variations in the large game. To give a brutally extreme example, consider a specification of chess positions so incomplete as to map from the viewpoint of White the approximately 10^{50} positions of the large game on to the seven shown in Fig. 2. Even this simple classification may have a role in the learning of chess. A player comes to

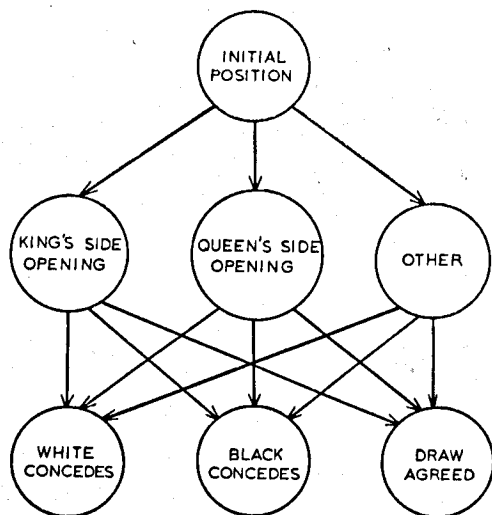


FIG. 2. A 'model' of chess.

realise from the fruits of experience that for him at least it pays better to choose King's side (or Queen's side, or other, as the case may be) openings. The pattern of master play, it may be added, has also changed, over the past hundred years, in ways which would show up even in this model, for example through decrease in the frequency of King's side openings and increase in the frequency of draws. Before leaving the diagram we shall anticipate a later topic by observing that it has the structure of a three-armed bandit with three-valued pay-offs.

In brief, we believe that programs for learning large games will need to have at their disposal good rules for learning small games. In the later part of this paper we apply the idea to an automatic control problem viewed as a 'large game'. As will appear, the approach is crude but the results to be reported show that decomposition into sub-games, each handled in mutual isolation by the same simple decision rule, is in itself sufficient to give useful performance in a difficult task.

As an example of the 'boxes' approach, consider again the matchbox machine of Fig. 1. This consists of 288 boxes, embodying a decomposition of the game of tic-tac-toe (noughts and crosses) into 288 sub-games, this being the number of essentially distinct board positions with which the opening player may at one time or another be confronted. Each separate box functions as a separate learning machine: it is only brought into play when the corresponding board position arises, and its sole task is to arrive at a good choice of move for that specific position. This was implemented in MENACE by placing coloured beads in each box, the colour coding for moves to corresponding squares of the board. Selection of a move was made by random choice of a bead from the appropriate box. After each play of the game the value of the outcome ('win', 'draw' or 'lose') was fed back in the form of 'reinforcements', i.e., increase, or decrease, in the probability of repetition in the future of the moves which in the past had led to the good, or bad, outcome. Echoing the terminology of Oliver Selfridge (1959), whose earlier 'Pandemonium' machine has something of the 'boxes' concept about it, one may say that the decision demon inhabiting a given box must learn to act for the best in a changing environment. His environment in fact consists of the states of *other boxes*. He cannot, however, observe these directly, but only the outcomes of plays of the game which have involved his box.

How should each demon behave?

PROBABILISTIC DECISION BOXES

In MENACE, and its computer simulation, choices among alternative moves were made initially at random, and after each play the probabilities of those moves which the machine had made were modified by a reinforcement function which increments move-probabilities following a win, leaves them unchanged following a draw and diminishes them following a defeat. In an alternative mode the value of the outcome is measured not from a standard outcome as baseline but relative to the past average outcome. With this sliding origin feature, a draw is reckoned a good result when defeat has been the rule, but a bad result when the machine is in winning vein. Before leaving the topic we shall make two remarks about adaptive devices based on the reinforcement of move-probabilities:

- (i) such devices cannot be optimal;
- (ii) it may nevertheless be possible in some practical contexts, to improve the best deterministic devices that we know how to design by incorporating a random variable in the decision function.

The basis of the second remark is connected with the fact that a move has an information-collecting role, and that a trade-off relation exists between expected gain of information and expected immediate payoff. The existence of this relation becomes apparent as soon as we try to devise optimal deterministic decision rules to replace *ad hoc* reinforcement of move-probabilities.

DETERMINISTIC DECISION BOXES

The task then is to draw up optimal specifications for the demons in the boxes.

In the present state of knowledge we cannot do this, as can be seen by considering the simplest learning task with which any demon could possibly be faced. Suppose that the demon's board position is such that his choice is between only two alternative moves, say move 1 and move 2. Suppose that his opponent's behaviour is such that move 1 leads to an immediate win in a proportion p_1 of the occasions on which it is used and move 2 wins in a proportion p_2 of plays. p_1 and p_2 are unknown parameters, which for simplicity we shall assume are constant over time (i.e., opponent does not change his strategy). The demon's task is to make his choices in successive plays in such a way as to maximise his expected number of wins over some specified period.

Under these ultra-simplified conditions, the problem is equivalent to the 'two-armed bandit' problem, a famous unsolved problem of mathematics. The difficulty can be expressed informally by saying that it can pay to make a move which is, on the evidence, inferior, in order to collect *more evidence* as to whether it really is inferior. Hence the problem can be formulated as that of costing information in the currency of immediate gain or loss: how much is the choice of a given move to be determined by its cash value for the current play of the game (we assume that all games are played for money) and how much by its evidence-collecting value, which may be convertible into future cash? We propose now to illustrate this formulation by exhibiting the behaviour of a game learning automaton designed by D. Michie to be optimal in all respects except that the evidence-collecting role of moves is ignored. R. A. Chambers' program which simulates the automaton is known as GLEE (Game Learning Expectimaxing Engine).

The learning automaton starts with a full knowledge of the moves allowed it and the nature of the terminal states of the game, but it is initially ignorant of the moves available to the opponent and only discovers them as they are encountered in play. As they are encountered, new moves are recorded and thereafter frequency counts are kept to record their use by the opponent.

Each terminal node of the game-tree represents a result of the game; in the application of GLEE to the game of Noughts and Crosses, a terminal node is given a utility value of +1 if the result is a win for the automaton, -1 for a loss and 0 for a draw. Non-terminal nodes are assigned scores which estimate the expected outcome value of the game. The automaton assigns these scores in the light of its previous experience and revises them as its experience is increased. Initially the scores are zero for all non-terminal nodes.

The graph is considered by levels, all nodes on the same level being produced by the same number of moves. Thus nodes at level one represent the states of the game produced by the opening move. Positions which can be equated by symmetry are represented by the same node (as is also the case with the MENACE automaton).

The revision of scores, i.e., the update, is done by backward analysis starting from the terminal node where the last game ended. The process moves back through the graph one level at a time. For levels where the automaton is on play each node is given a score equal to the maximum of the scores of nodes that can be reached by one legal move. Where the opponent is on play, his previous behaviour from the given node is considered and the node is assigned a score equal to the *expected value* of the outcome. We call this updating process 'expectimaxing'. The expected value calculation is derived as follows:

Consider a node, N , at which the opponent is to move and let him have used k different moves, each n_i ($i=1, \dots, k$) times in the past. The total number of alternative plays from N is unknown so we assume that c additional moves exist and reach nodes of score zero (the value 2 was taken for c in the experimental runs referred to later). The other moves end in nodes with scores s_i ($i=1, \dots, k$).

By a development of Laplace's Law of Succession we can determine the probability, p_i , that the next opponent move from N will use the i th alternative, i.e.,

$$p_i = \frac{n_i + 1}{\text{Estimated number of alternatives} + \text{total number of past plays from } N.}$$

$$= \frac{n_i + 1}{(k + c) + \sum_{i=1}^k n_i}$$

Knowing those values of p_i ($i=1, \dots, k$) and the scores reached by each alternative we can calculate the quantity.

$$E = \sum_{i=1}^k p_i s_i. \quad \text{This defines the score associated with the node } N.$$

To make a move the automaton examines all the legal alternatives and chooses the move leading to the position having the highest associated score, ties being decided by a random choice. It thus seeks to optimise the expected outcome of the *current* play of the game only. Fig. 3 shows the results of three trials each of 1000 games in which the opening player was a random move generator and the second player was the learning automaton. The automaton's score is shown as the number of wins minus the number of losses over each 100 consecutive games. By analysis of the optimal move trees associated with each type of opening play, the score level for an optimal strategy was calculated. It can be seen from Fig. 3 that the performance of the learning automaton levels out below this optimal level. This is due to an inherent weakness in the automaton's aim of short-term optimisation. The weakness is that as soon as the automaton finds, in a given position, a move with a positive expected outcome, then since other tried moves have had negative outcomes and unknown moves have zero expected outcomes it will continue to use that move as long as its expected outcome stays just greater than

zero. In this way it may be missing a better move through its lack of 'research'. Thus we see that neglect of evidence-collecting can lead to premature decision taking. This can be demonstrated in detail by reference to the very simple game of two-by-two Nim.

In this trivial case of the game there are two heaps of two objects and each of the two players in turn removes any number of objects from any one heap. The winner, according to the version considered here, is the player to remove

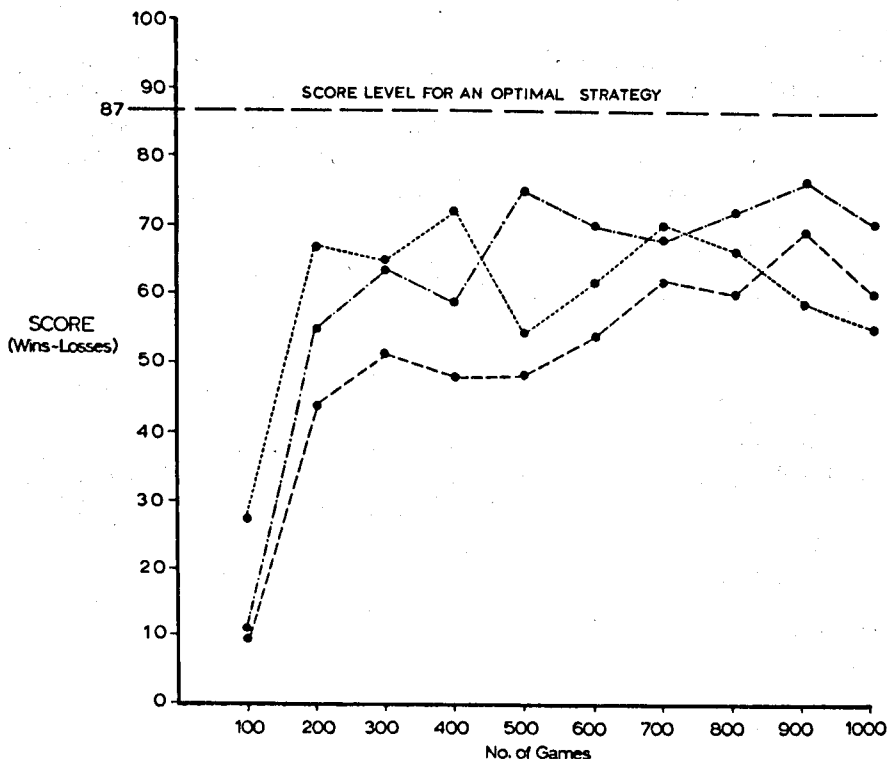


FIG. 3. Some representative results of GLEE playing Noughts and Crosses as second player, the opening player being a random move generator. The vertical scale gives hundred game totals.

the last object. Let the automaton be the opening player, then symmetry reduces the choice of opening moves to two. Against a good opponent the automaton will always lose the game, and both openings will be equally bad. But consider the case of a random opponent. Fig. 4 shows a graph representation of the game. For his first reply the opponent has three alternative moves from node (A) and two from node (B). As the opponent plays randomly, games proceeding from (A) will on average give the automaton two wins for every loss while those games proceeding from (B) will yield equal numbers of wins and losses. Thus in the long term the automaton's best opening move is to select position (A). Consider now a sequence of four games as follows:

the automaton chooses (A) and loses, (B) and wins, then (B) twice more losing each time. Fig. 5 shows the state of the automaton's knowledge with the assumed and as yet unknown opponent moves. Applying expectimax, as

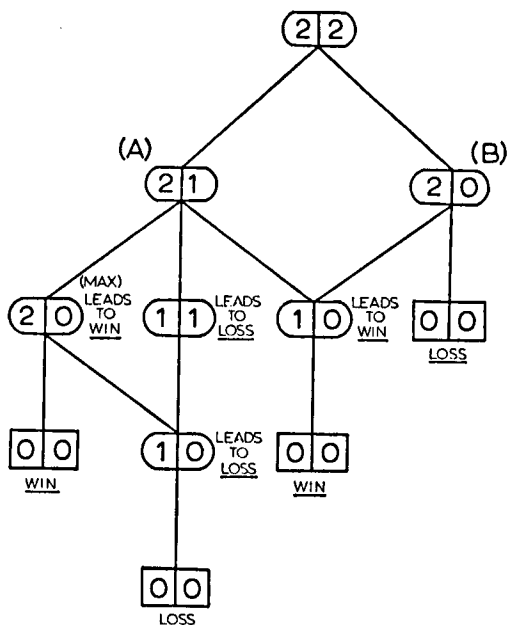


FIG. 4. 2×2 NIM showing winning and losing moves. The integer pairs at nodes indicate the state of play by showing the number of items in each of the two heaps.

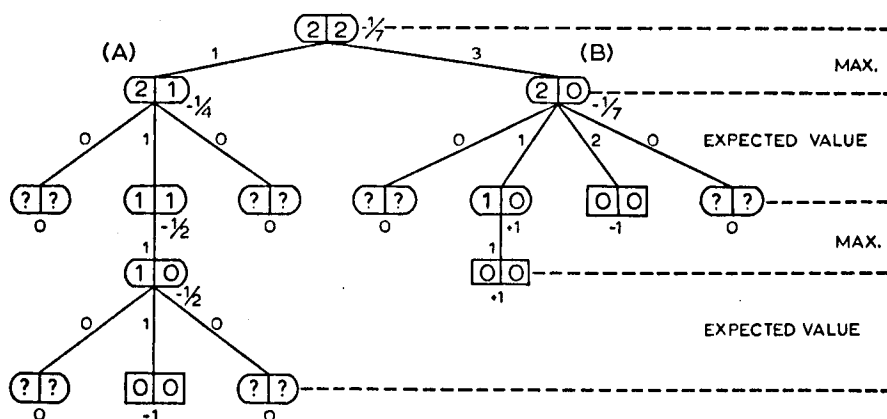


FIG. 5. 2×2 NIM showing expectimax scores after a hypothetical run of four games. The numbers against nodes are scores and the integers against graph connections show move frequencies.

explained, shows that due to the premature decision taking, brought on by losing at (A) and winning first time at (B), the automaton will still choose (B) despite the last two losses.

This feature of premature decision taking was also very evident in the trials of the automaton at noughts and crosses against a random opponent. Table 1 illustrates a series of games in which the automaton was second player and shows all the games in the first 100 of the series in which the random opponent opened in the centre. Analysis of this opening shows that by symmetry there

TABLE 1

Some selected results from the GLEE game-learning program using the game of Noughts and Crosses (tic-tac-toe). The automaton is the second player against a random opponent. Only those plays are shown in which the opponent opened to the centre of the board.

For further explanation see text.

No. of each game in which random opponent opened in centre	Automaton's first reply:		Result for automaton
	M=middle of side	C=corner	
7	C		Lost
12	C		Lost
23	C		Lost
26	M		Won
27	M		Won
28	M		Won
31	M		Drew
37	M		Won
39	M		Lost
40	M		Won
44	M		Won
55	M		Lost
56	M		Won
61	M		Won
62	M		Lost
65	M		Lost
66	M		Lost
67	M		Lost
69	M		Won
74	M		Lost
77	M		Won
80	M		Lost
83	M		Lost
87	M		Won
89	M		Drew
90	M		Lost
92	M		Lost
95	M		Won
98	M		Drew
100	M		Lost

are only two distinct replies, a corner, C, or the middle of a side, M, the first being superior to the second. Indeed against an optimal strategy the second player always loses after playing M. However against a random opponent the automaton chanced to lose three games using the first reply, C, and then to win three using the second, M. This was sufficient to cause it to play the second, poorer, move throughout a trial of 2000 games. In the first 1000 games

the random player opened 336 games with the centre move with the automaton replying by moving into the middle of a side, M. In a similar series of 1000 there were 320 games with centre opening but this time the automaton always played in the corner, C. The results as percentages were as follows:

Automaton replying M 62% wins 18% losses 20% draws

Automaton replying C 75% wins 16% losses 9% draws

A remedy for this fault, which illustrates again the information-versus-payoff dilemma of trial-and-error learning, could be developed by use of a method which we have incorporated in our application of the boxes idea to a problem in adaptive control. The program is called BOXES and the method is called the 'target' method.

ADAPTIVE CONTROL AS A STATISTICAL GAME

BOXES is based on formulating the adaptive control problem in terms of the 'game against nature'. A valuable and detailed survey of formulations of this type has recently been published by Swarder (1966), who restricts himself in the main to theoretical questions of optimality. The boxes algorithm was devised by D. Michie for tasks for which optimal policies cannot be specified, and implemented in 1961 by Dean Wooldridge, junior, as a FORTRAN II program. It is incomplete in a number of ways. Yet it illustrates clearly the idea, expressed above, of reducing a large game to a small, model, game and then handling each separate board state of the model as a separate sub-game (box). In the adaptive control situation, where the state variables are real numbers, the large game is infinitely large, so that the sacrifice of information entailed in the boxes approach is correspondingly extreme. In spite of this, the algorithm actually does master the simulated task, which is one which severely taxes conventional methods.

BALANCING A POLE

The model task chosen for experimentation was that of balancing a pole on end. For the purpose in hand, this system has a number of merits:

- (1) it has been thoroughly studied by Donaldson (1960) in an illuminating exercise on a related, but different, theme—namely the design of an automaton to learn a task by sensing the control movements made by a second automaton already able to perform this task (e.g., a human being);
- (2) Widrow and Smith's (1964) subsequent and independent approach to Donaldson's problem used a design reminiscent of the boxes principle;
- (3) a recent study in automatic control (Schaefer and Cannon 1966) has shown that the pole-balancer problem generalises to an infinite sequence of problems of graded difficulty, with 1, 2, 3, . . . , etc., poles balanced each on top of the next. There is thus ample scope, when needed, for complicating the problem;

- (4) even the 1-pole problem is extremely difficult in the form for which adaptive methods are appropriate, i.e., physical parameters of the system specified in an incomplete and approximate way, and subject to drift with the passage of time. No optimal policy is known for such problems in general.

Fig. 6 shows the task. A rigid pole is mounted on a motor-driven cart. The cart runs on a straight track of fixed length, and the pole is mounted in such a way that it is only free to fall in the vertical plane bounded by the track.

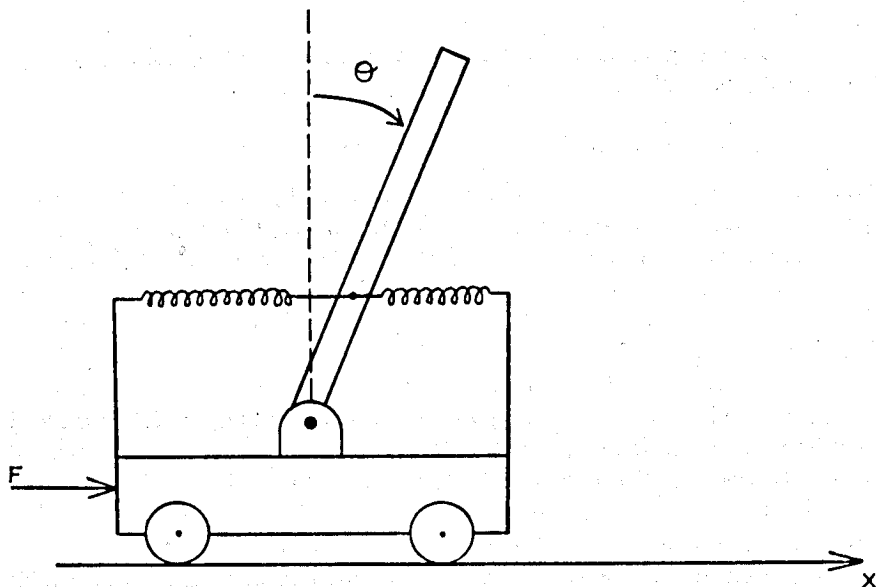


FIG. 6. A simple representation of the system being controlled. The spring is used to reduce the effective force of gravity. This feature was incorporated in the original laboratory apparatus so as to bring it within the range of human powers of adaptive control.

The cart's motor applies a force which is constant except for the sign. The sign is controlled by a switch with only two settings, + and -, or 'left' and 'right'. The problem is thus one of 'bang-bang' control. For the experimental runs reported here the cart and pole system was simulated by a separate part of the program, not built in hardware. The interval from sense to control action was set to zero and the interval from control action to sense to 0.05 sec. Various other parameters of the simulation program corresponding to such quantities as frictional resistance, length of pole, the mass of the pole and of the cart, and the spring clamping of the pole, were adjusted until the behaviour of the simulated system approximated to that of a real physical system.

BLACK BOX

The form in which we pose the problem is this. The adaptive controller must operate without prior knowledge of the system to be controlled. All it knows

is that the system will emit signals at regular time intervals, each signal being *either* a vector describing the system's state at the given instant *or* a failure signal, indicating that the system has gone out of control. After a failure signal has been received, the system is set up afresh and a new attempt is made. On the basis of the stream of signals, and this alone, the controller must construct its own control strategy. No prior assumptions can safely be made about the nature of the system which emits the signals, nor of the time-invariance of its parameters. In this sense, the task is never finished, since parameters of the system might drift into a new configuration requiring further modification of the controller. Most 'black box' problems permit some time-invariance assumptions. Our program must be ready for an even blacker box.

STATE SIGNALS

The state of the system at any instant can be represented by a point in an n -dimensional space of which the axes correspond to the state variables, n in number. In our case it is convenient to recognise four such variables,

- (i) x , the position of the cart on the track,
- (ii) θ , the angle of the pole with the vertical,
- (iii) $\dot{x}$, the velocity of the cart, and
- (iv) $\dot{\theta}$, the rate of change of the angle.

(iii) and (iv) could be estimated by differencing (i) and (ii) with respect to time interval, but the program has these sensed separately. The state signal thus consists of a 4-element vector.

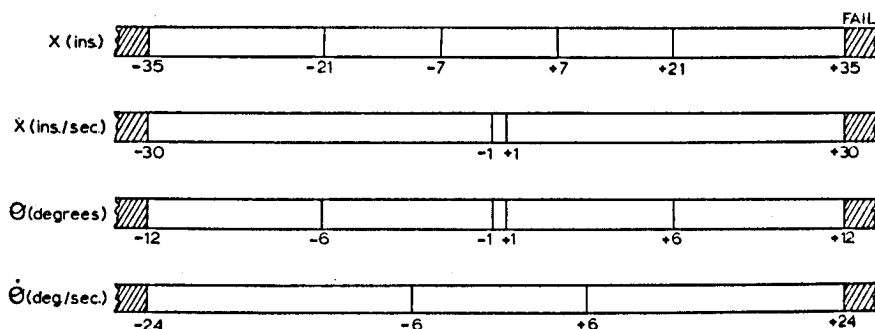


FIG. 7. Thresholds used in quantizing the state variables.

The 'model' was constructed by quantising the state variables by setting thresholds on the four measurement scales as shown in Fig. 7: thus, only 5 grades of position, x , were distinguished, 5 of angle, 3 of velocity and 3 of angle-change. It can be seen that this quantisation cuts the total space into a small number of separate compartments, or boxes; in our implementation the number was $5 \times 5 \times 3 \times 3 = 225$.

DECISION RULES

In order to envisage how the rest of the algorithm works it is easiest to imagine each one of the 225 boxes as being occupied by a local demon, with a global demon acting as a supervisor over all the local demons, according to the

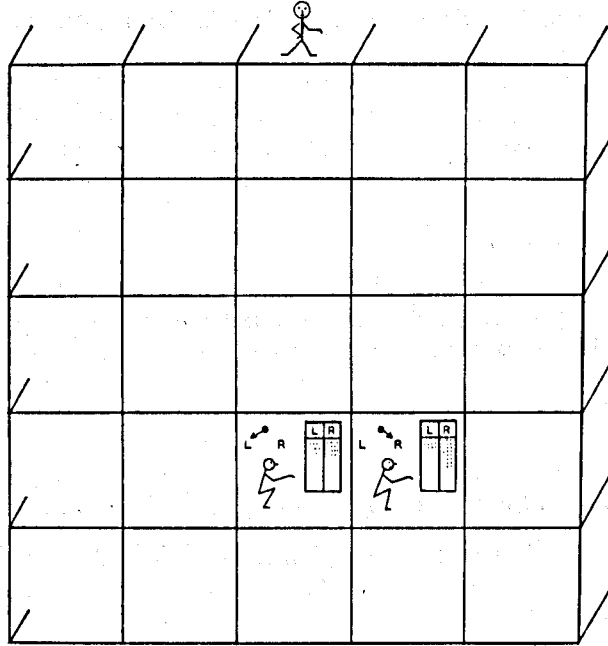


FIG. 8. A simple representation of the independent decision mechanisms controlling each box in the state space.

general scheme of Fig. 8. Each local demon is armed with a left-right switch and a scoreboard. His only job is to set his switch from time to time in the light of data which he accumulates on his scoreboard. His only experience of the world consists of:

- (i) *LL* the 'left life' of his box, being a weighted sum of the 'lives' of left decisions taken on entry to his box during previous runs. (The 'life' of a decision is the number of further decisions taken before the run fails.)
- RL* the 'right life' of his box.
- LU* the 'left usage' of his box, a weighted sum of the number of left decisions taken on entry to his box during previous runs.
- RU* the 'right usage' of his box.
- (ii) *target* a figure supplied to every box by the supervising demon, to indicate a desired level of attainment, for example, a constant multiple of the current mean life of the system.

- (iii) $T_1, T_2, \dots, T_i, \dots, T_N$ times at which his box has been entered during the current run. Time is measured by the number of decisions taken in the interval being measured, in this case between the start of the run and a given entry to the box.

The demon's decision rule gives S , the setting of his switch, as a function of (i) and (ii). We can express this informally by saying that when the system state enters his box for the first time, the demon sets S by applying his decision rule, and notes the time of entry, T_1 ; if his box is re-entered during the run, the demon notes the new entry time, T_i , but does not change his switch, so that the same decision, S , is taken at every entry to his box during the course of that run. When the run fails, the global demon sends a failure message to all the local demons, indicating the time of failure, T_F , and also giving a new value of *target*. Each local demon then updates the 'life' and 'usage' totals for his box so that with these new values and the new value of *target* he can make a new decision if his box is entered during the next run.

The rules for updating and resetting adopted in the present case were as follows, using the convention that, say, X' means 'the previous value of X ':

Consider a box in which the decision setting, S , was *left*, during the run which has just terminated.

Let N = the number of entries to the box during the run;

DK = constant multiplier less than unity which performs the function of weighting recent experience relative to earlier experience;

K = another multiplier weighting global relative to local experience;

then the local demon updates his totals, defined under (1) above, using:

$$LL = LL' \times DK + \sum_{i=1}^N (T_F - T_i);$$

$$LU = LU' \times DK + N;$$

$$RL = RL' \times DK;$$

$$RU = RU' \times DK.$$

The global demon has similar totals, 'global life', GL , and 'global usage', GU , and updates them using:

$$GL = GL' \times DK + T_F;$$

$$GU = GU' \times DK + 1;$$

From this the demon calculates *merit* as $\frac{GL}{GU}$,

and then computes *target* as $C0 + C1 \times \text{merit}$,
where $C0 \geq 0$ and $C1 \geq 1$.

From *target* and his local totals, the local demon calculates the 'left value' of his box from

$$\text{value}_L = \frac{LL + K \times \text{target}}{LU + K}$$

and similarly the 'right value', value_R . The control action, S , is selected as *left* or *right* according as value_L is greater or less than value_R .

PERFORMANCE TESTS

After some preliminary trials the adjustable parameters DK and K were set at 0.99 and 20.0 respectively for the tests. The values of $C0$ and $C1$ were 0 and 1 respectively, throughout all the trials. Time has not allowed

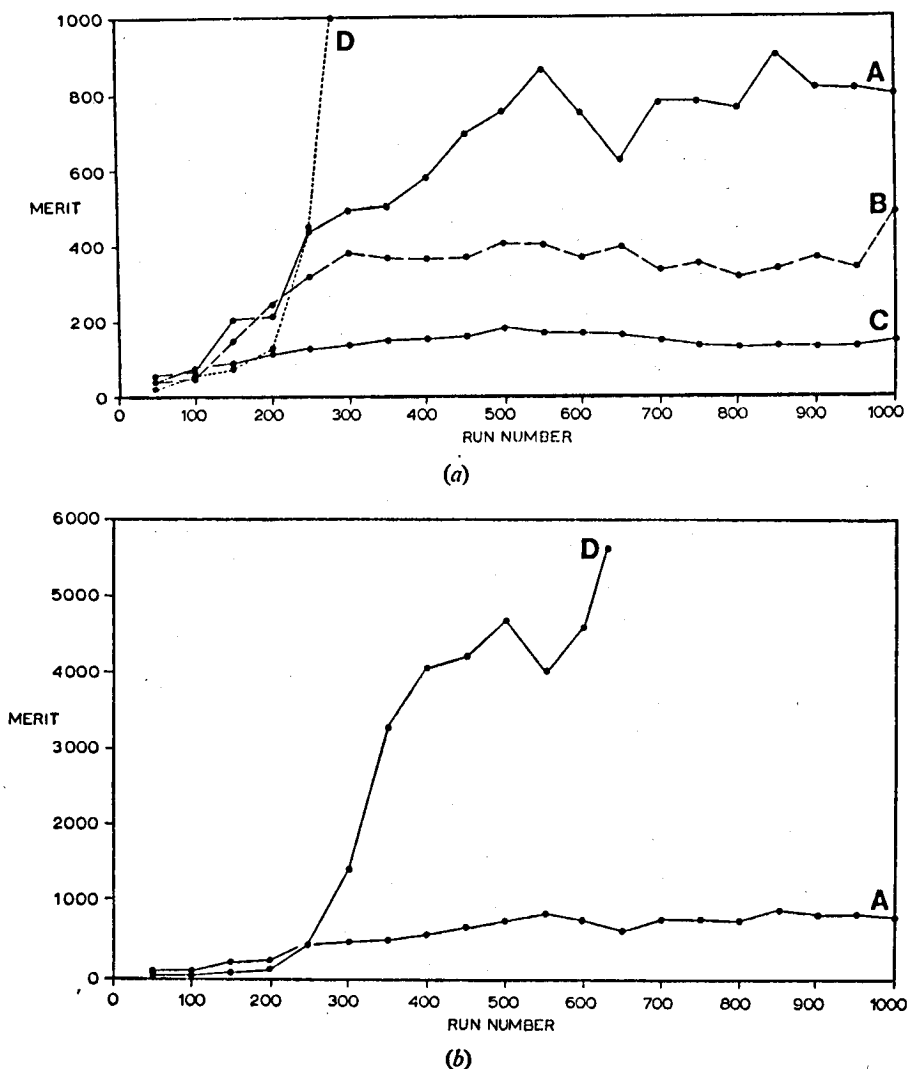


FIG. 9. Duplicate runs of BOXES. Run A is shown on both graphs to facilitate comparisons.

any systematic attempt at optimisation since the decision algorithm has only recently been developed to the stage reported here. A test run consisted in a succession of attempts by the program, each attempt being terminated by receipt of the failure signal. For each new attempt, or 'control run', the

system state was set up by choosing a random point from a uniform distribution over the central region of the space defined by $x = \pm 21.0$ in, $\dot{x} = \pm 1.0$ in/sec, $\theta = \pm 6.0$ deg and $\dot{\theta} = \pm 6.0$ deg/sec.

Figs. 9 (a) and (b) show duplicate test runs A, B, C and D. In these test runs the value of *merit* (explained in the previous section) was recorded after the update which followed the failure of each control run, and the graphs show these values for every fiftieth control run. For each test run the same values were kept for all the adjustable parameters, but different starting values were given to the pseudo-random number generators which were used for resolving control decisions with equal left and right values and for setting up the new starting conditions of the system for each control run.

The wide limits within which starting conditions could be generated caused considerable variation in the duration of control runs. This is not obvious from the individual graphs because of the pronounced smoothing effect of the weighted-average calculation by which *merit* is computed, but in fact the length of extreme runs was a factor of 10, or more, greater or less than *merit*. Thus test run C produced a run of 1368 decisions at a point where *merit* was only 146, and test D, at a point with a *merit* value of 4865, produced a run of over 72 000 decisions which is equivalent to an hour of real-time control.

The use of different random number sequences in test runs, combined with this variation in the duration of control runs, means that the sequences of boxes entered by the system and the early decisions taken in those boxes differed greatly for each test run. Since the 'success' of any box depends on the decisions taken in neighbouring boxes, it is easy to see how the early decision settings for the whole collection of boxes were different for different test runs, whereas the low level of experience caused the control runs to be short and thus produce similar values of *merit* for all the tests. However, because of premature decision-taking fixing some of these early decision settings, the later behaviour of the tests varied considerably as can be seen in the graphs of Fig. 9, but the influence of the *target* term in the decision rule is noticeable even without optimisation. Graphs B and C are comparable, respectively, to the best and average performances of earlier, non-*target*, versions of BOXES. Graph A is better than any previous performance and D indicates what we hope to achieve when the target method has been improved.

'LEVEL OF ASPIRATION'

Earlier in this paper we referred to the fact that a move which on the evidence is disadvantageous may none the less be worth making if the expected information-gain associated with it is sufficiently high, and we emphasised that this fact should be reflected in the design of a trial-and-error automaton.

The problem of optimal information trade-off, as stated earlier, still awaits solution even in the simplest case—the 'two-armed bandit' problem. We have, in the meantime, evolved an approximate solution to this family of problems, and have found that it shows promise of being able to keep the 'premature

fixation' hazard at bay. It can moreover be made to yield, in some contexts at least, a rather close approach to optimality as we have verified by comparison with a limited range of optimal values computed for the two-armed bandit problem. The essential idea is to use a 'target' level of performance to calculate for the available alternative moves *not* their expected payoffs but 'optimistic' values of these expectations. Choice between moves is then made by comparing the optimistic values. These are calculated by taking a weighted mean between the expected value of the move and the target value. The weight of the expected value is taken as the amount of information associated with it, while the weight of the target value is an adjustable parameter of the program. The higher the target value and the greater its weight, the more 'research minded' is the behaviour of the automaton, tending to try new possibilities rather than to consolidate partial successes. Its 'level of aspiration', in anthropomorphic terms, is higher.

The idea is readily generalised, and can be immediately applied, for example to the GLEE automaton, or to the two-armed bandit itself. We have only recently started to explore this line: even the results of Fig. 9 were obtained before any optimisation of the parameters of the target method was attempted. We can, however, say in summary that the target method, when grafted on to the 'boxes' approach, has given good performance for adaptive control problems inaccessible to standard procedures.

REFERENCES

- Donaldson, P. E. K. (1960). Error decorrelation: a technique for matching a class of functions. *Proc. III International Conf. on Medical Electronics*. pp. 173-178.
- Michie, D. (1961). Trial and error, in *Science Survey, 1961* (eds. S. A. Barnett and A. McLaren) part 2, pp. 129-145. Harmondsworth: Penguin.
- Michie, D. (1963). Experiments on the mechanisation of game-learning. 1. Characterisation of the model and its parameters. *Computer Journal*, 6, 232-236.
- Newell, A., Shaw, J. C., & Simon, H. A. (1960). A variety of intelligent learning, in a general problem solver in *Self-organizing Systems* (eds. Marshall C. Yovits and Scott Cameron) pp. 153-189. London: Pergamon.
- Schaefer, J. F., & Cannon, R. H. Jr. (1966). On the control of unstable mechanical systems. *Proc. IFAC 1966*. Paper 6C.
- Selfridge, O. G. (1959). Pandemonium: a paradigm of learning, in *Mechanization of Thought Processes*, Vol. 1. N.P.L. Symposium No. 10, pp. 511-531.
- Sworder, D. D. (1966). *Optimal Adaptive Control Systems*. Academic Press.
- Widrow, B., & Smith, F. W. (1964). Pattern recognising control systems, in *Computer and Information Sciences* (eds. J. T. Tou and R. H. Wilcox). Clever Hume Press.

10

A REGRESSION ANALYSIS PROGRAM INCORPORATING HEURISTIC TERM SELECTION

J. S. COLLINS

EXPERIMENTAL PROGRAMMING UNIT
UNIVERSITY OF EDINBURGH

INTRODUCTION

Consider a system in which the value of one variable (called the *dependent variable*) depends on the values of one or more other variables (called the *independent variables*). An example of such a system is a furnace whose heat output depends upon the rates at which oil and air enter the system. If we measure the values of the heat output for different values of the oil and air flows, the resulting data will tell us something of the behaviour of the system. Regression analysis provides a method by which a formula which predicts the value of the dependent variable in terms of the values of the independent variables may be determined from such data. Thus, in the case of the furnace, the regression equation would give us a formula for calculating the heat output for any given values of oil flow and air flow.

The theory of regression analysis and the development of the standard formulae used is available in many standard statistics textbooks such as Kendall and Stuart (1961).

The basic data for regression analysis consists of a number of *observations*. An observation consists of the set of values of the $n+1$ system variables. There must be at least $n+1$ observations. The number of observations is normally large compared with the number of variables. This redundancy enables random errors in the observations to be smoothed out. The difference between the number of observations and the number of variables (a measure of the redundancy of the measurements) is called the number of *degrees of freedom*.

The result of a linear regression analysis on $n+1$ variables is a *linear regression equation* which predicts the value of the dependent variable as a function of the n independent variables. Let x_0 denote the dependent variable and $x_1, x_2, \dots, x_n$ denote the n independent variables. A linear regression equation is of the form

$$y = b_0 + b_1x_1 + \dots + b_nx_n$$

where y is the predicted value of the dependent variable and $b_0, b_1, \dots, b_n$ are constants known as the *regression coefficients*. The values of the regression coefficients are computed to minimise $\sum (y - x_0)^2$ where the sum is over all observations. This method successfully smooths small random errors in the data if the order of magnitude of the errors is approximately the same for each observation. The existence of one large error in the data is likely to invalidate the regression equation. This happens because the differences between actual and predicted values of the dependent variable are squared.

A FORTRAN program for carrying out linear regression analysis is described by Dallemand (1958). The analysis is carried out in a way which makes the program particularly useful when only a subset of the set of independent variables significantly affects the behaviour of the dependent variable. The regression equation is built up one variable at a time starting with a constant term. The variable added at each stage is that independent variable which best predicts the discrepancy between the actual behaviour of the dependent variable and the prediction of the regression equation so far. If the best available independent variable does not significantly improve the accuracy of the regression equation, no variable is added and the assembly process stops. If at any time during the assembly process, a variable in the regression equation ceases to contribute significantly to the accuracy of the regression equation, that variable is deleted from the equation. The final regression equation is such that all the variables in the equation are significant and all the variables not in the equation are not significant.

Dallemand also describes an extension to the regression analysis program which allows non-linear regression equations to be produced. By computing functions of each of the values of the independent variables for each observation in the data and presenting the result to the linear regression analysis as if it were raw data, more complex relationships can be studied. Functions commonly used for this purpose are powers, reciprocals or logarithms of the independent variables.

Westervelt (1960) describes a further extension to this type of regression analysis. The main features of this development are described in the remainder of this section.

The regression equation consists of a linear combination of a number of *terms*. Each term consists of one or more *factors* and the factors are simple functions of the independent variables. The number of factors in a term is called the *interaction order* of the term. If there are n independent variables and p functions then each independent variable may be either not included or

included in a term as one of the p functions. There are thus $p + 1$ possibilities for each independent variable and therefore $(p + 1)^n$ combinations in all. This however includes the term in which all factors are absent and therefore the total number of terms is $(p + 1)^n - 1$. This number increases rapidly with p and n . Even for a relatively simple problem involving 10 functions of 5 independent variables, the total number of terms is 161 050.

Such large numbers of terms cannot be treated by a conventional regression analysis procedure which needs to store a correlation matrix which involves every pair of terms. It is therefore necessary to examine the terms a few at a time in a search procedure. The solution adopted was to try a small number of terms during each of a number of trials, retaining those terms found to be significant by the regression analysis for the following trial. This alone was not sufficient to handle the large number of possible terms. The selection of terms for each trial was made to depend upon the significance or otherwise of similar terms tested in previous trials. Thus the results of each trial help to direct the search for suitable terms into apparently promising areas of the term space.

Similarity between terms is defined by a classification scheme. Terms are classified according to the number of factors and for each factor, the independent variable concerned and the function used. As a result of previous trials, the program learns which interaction orders (number of factors in a term), independent variables and functions are promising. Westervelt gave no indication of the effectiveness of this learning mechanism.

A GENERAL REGRESSION PROGRAM

Using the schemes described by Westervelt and Dallemand, a general regression program was written. The object was to investigate the efficiency of the search process in an improved search space, and at the same time produce a useful program.

A simplified flow diagram of this general regression program is shown in Fig. 1. On entry to the program, standard initial conditions are set up. Apart from essential data, the problem specification need only contain desired departures from these standard conditions. An annotated example of data prepared for the regression program is given in Appendix 1. A set of terms is selected which are evaluated for each observation in the data. The resulting edited data are analysed by the regression program resulting in a regression equation containing those terms which are found to be significant.

If the regression equation produced satisfies the accuracy criterion specified by the user, the details of the regression equation are printed and the process is terminated. If the resulting regression equation is not sufficiently accurate, a further trial is initiated. Before a new set of terms is selected, the term selector is modified to encourage the selection of terms similar to those found to be significant in the previous trial. These terms are retained and appear in the set of terms selected for the next trial. Terms similar to those not found significant in the previous trial are discouraged. These terms are replaced by new terms for the following trial.

The most important parts of the program in respect of the investigation were the term selector and term selector modifier. Because of this and because the other parts of the program use well-known techniques which are described elsewhere, only these parts of the program are described in detail.

A term is a product of a number of factors. A factor may be one of four types of function of one of the independent variables. The four types are positive powers, $x, x^2, x^3, \dots$, reciprocal powers $x^{-1}, x^{-2}, x^{-3}, \dots$, roots $x^{1/2}, x^{1/3}, \dots$ and reciprocal roots $x^{-1/2}, x^{-1/3}, \dots$. These four function types are referred to as types 0, 1, 2 and 3 respectively. The order of a function

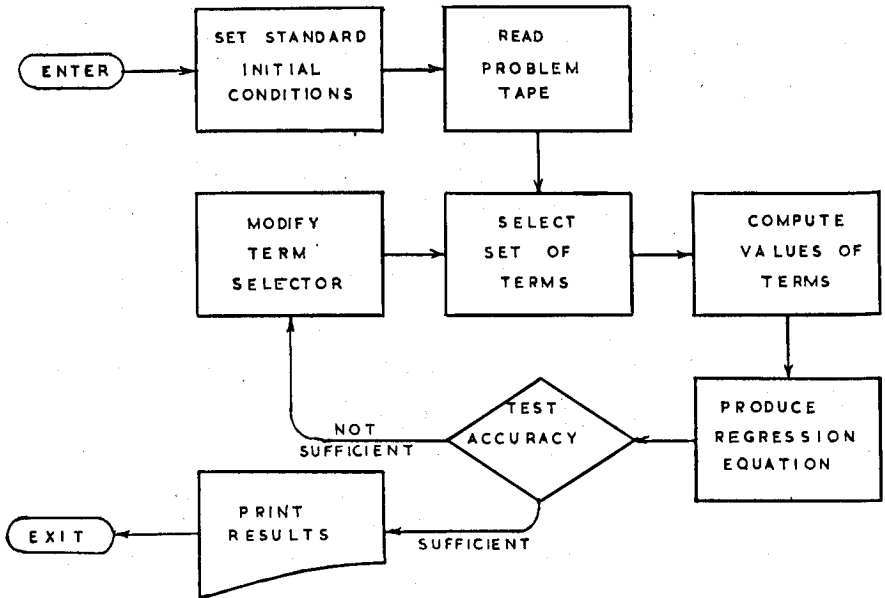


FIG. 1. Flow diagram of the regression program.

defines which of the series of functions corresponding to each function type is required. For example $x^2, x^{-2}, x^{1/2}$ and $x^{-1/2}$ are all functions of order 2.

A factor is selected with the aid of a *factor selection tree*. The factor selection tree for three independent variables and a maximum function order of three is shown in Fig. 2. A factor is selected by choosing a path in the tree from the starting node to one of the terminal nodes. Each terminal node corresponds to one distinct factor. Three consecutive choices are required to select one factor. The first choice is between the three independent variables. The second choice is between the four function types and the third choice is between the possible orders for the chosen function type. Functions of type 2 and 3 have a minimum order of 2 as an order of 1 would duplicate functions included in types 0 and 1.

The order in which the choices are made defines the manner in which all

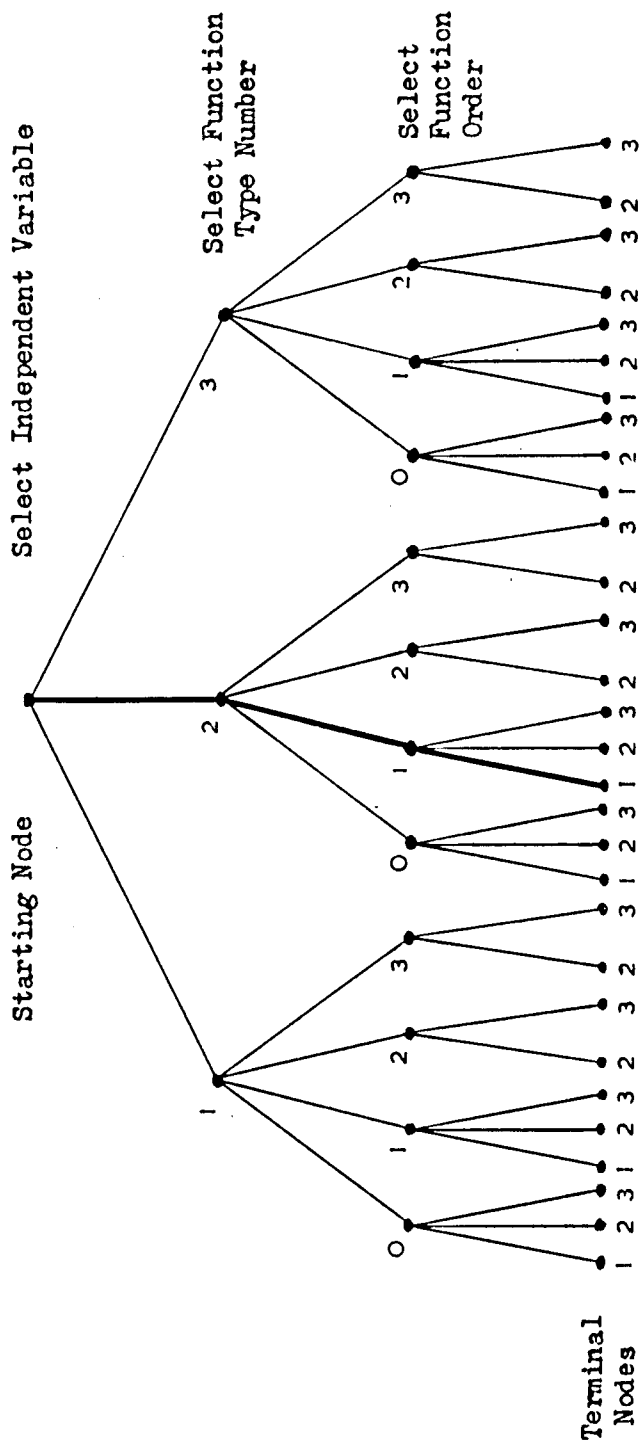


FIG. 2. The factor selection tree.

possible factors are classified. The heavy line in Fig. 2 indicates the choice path which results in the selection of the factor x_2^{-1} . Let a probability be associated with the choice of every possible branch at each decision node. There are therefore three probabilities at the starting node which indicate the relative probability of selecting each possible independent variable for a factor. The selection probability of any factor is equal to the product of the three probabilities associated with the three branches of the selection path defining the factor.

Let the three probabilities involved in the choice of the factor x_2^{-1} be each multiplied by *probability multipliers* a , b and c respectively. Not only does this affect the probability of selecting x_2^{-1} , but the selection probabilities of other factors are altered also. The selection probabilities of the factors x_2^{-2} and x_2^{-3} are multiplied by ab and the selection probabilities of x_2 , x_2^2 and x_2^3 are multiplied by a . Thus the effect of modifying the selection probability of one factor is to modify, by different amounts, the selection probabilities of other factors which are defined as similar to the given factor by the structure of the factor selection tree. The values of the probability multipliers control the degree of this effect on similar factors.

By reducing any probability to zero, the selection probability of any class of factors may be reduced to zero. This is useful, for example, to eliminate the selection of reciprocals when zeros exist in the observations.

A term is characterised by the factors it contains and the number of them. The order of factors in a term is immaterial. In order to select a term, it is first necessary to decide upon the interaction order (number of factors) of the term. The factor selection tree is then used once to select each factor of the term. The selection of further factors containing a previously selected independent variable is avoided by temporarily setting the independent variable probability to zero after the selection of each factor. There is a probability associated with each possible interaction order.

Terms are selected using a random number generator which selects between possibilities with the given probabilities. When a set of terms has been selected and a regression equation has been formed from a subset of them, the interaction order probabilities and factor selection tree probabilities are modified to encourage the selection of promising terms. The probabilities involved in the selection of terms in the regression equation are multiplied by probability multipliers greater than unity. The probabilities involved in the selection of terms not in the regression equation are multiplied by probability multipliers less than unity. Because all probabilities are multiplied by finite non-zero probability multipliers, the selection probability of no factor can ever become zero. All possible terms therefore always have a finite probability of being selected.

The choice of values for the probability multipliers naturally affects the efficiency of the search procedure. If their values are too near to unity, the search is slow to take advantage of new discoveries. If their values are too far removed from unity, the search is too erratic.

PERFORMANCE OF THE PROGRAM

A regression program, using the techniques described in the previous section has been written in Mercury Autocode for the Ferranti Mercury computer. The program can deal with up to 200 observations on up to 15 variables. In each trial of the analysis, up to 15 terms can be selected, evaluated and tested for significance by the regression algorithm.

The program has been tested on data from a variety of sources. This includes actual physical data as well as artificially generated data. Physical data analysed by the program has included ecological, engineering and simple calibration measurements. In some cases, suitable formulae have been produced by the program. In others, the significant terms discovered by the program have aided subsequent theoretical analysis.

One application of this program described elsewhere (Collins 1964) has been to assist the programmer in coding subroutines whose argument list/value relationship is known by example but not as an explicit function. The application was concerned with a search procedure and the subroutine in question was required to decide whether to proceed along a certain path or to give up. The parameters of this subroutine were a set of variables on which the choice was thought to depend. The program was run using a 'human subroutine' (arguments displayed on an on-line typewriter and value typed in by the user) and the argument list/value pairs were collected and later analysed by the regression program. The formula produced by the regression program was then placed in the search program in place of the human subroutine. The new search program was then run and its performance compared with that using the human subroutine.

From the point of view of testing the performance of the program, the most valuable tests have been on data generated by the evaluation of analytic expressions for a number of sets of values of the independent variables. By selecting expressions which can be represented exactly by the general regression program, the progress of the search towards the known goal can be observed. A test of this type is fully documented in Appendix 2.

As a result of tests with different values for the probability multipliers, the following values were found to be generally suitable.

Interaction Order:	2.0	0.50
Independent Variable:	1.5	0.67
Function Type:	1.5	0.67
Function Order:	1.25	0.80

The first probability multiplier is for terms included in the regression equation; the second which is the reciprocal of the first is for terms not included in the regression equation. The values are not critical. The function order multiplier should be closer to unity than the function type multiplier to prevent emphasis of a particular function order rather than type. Using these probability multipliers, the selection probability of a factor is multiplied or

divided by $2 \cdot 8125$ as a result of its significance or non-significance in the regression analysis. Similarly, the selection probability of a term is multiplied or divided by $2 \times 2 \cdot 8125^n$ where n is the number of factors in the term, and the selection probability of a similar term in which one factor is different is multiplied or divided by $2 \times 2 \cdot 8125^{(n-1)}$.

CONCLUDING REMARKS

A computer program for generating analytic expressions to explain experimental measurements has been described. The use of heuristics enables the program to find suitable expressions in a much shorter time than would otherwise be possible.

In some cases a theoretical justification has been found for terms produced by the program from experimental measurements. In many other cases, the program has suggested terms which have provided clues helpful in the theoretical analysis of the experimental system.

The use of a selection tree, in which selection is controlled by probabilities, has been found to be a workable method of defining similarity between discrete objects. A selection tree defines a space of all possible objects in which similar objects are closely situated. The method is analogous to the classification of books in a library in which similar books occupy neighbouring positions on the shelves. The result of this type of classification is that much less effort is required to find a book on a given subject than if the books were placed at random. Tests on the regression program have shown the classification method used to be satisfactory. There is some tendency for multiple peaks to exist in the search space but the use of random numbers for term selection prevents the search process from getting stuck at these points.

A possible modification of the factor classification scheme suitable for the current set of functions could be to classify positive powers with positive roots and negative powers with negative roots. If this is done, the significance of a factor such as $x_1^{1/2}$ increases the selection probability of a factor such as x_1 . The resulting modified factor selection tree is shown in Fig. 3. This suggestion was not implemented as it is function dependent. That is, it works for the particular functions used but not if other functions are used. It was considered desirable in this application to keep the program function independent. Because of this it is possible to use the program with any other functions simply by changing the routines responsible for calculating the values of the functions in terms of their symbolic description and for printing a description of the terms.

A noticeable property of the program is that the change in the set of terms from trial to trial behaves somewhat like a simple evolutionary system. Terms found by the regression analysis to be significant 'survive' for use in the next trial. In addition the probability of a similar term being generated is increased. Consider a particular characteristic of a term such as its interaction order. Let a and b be the interaction order probability multipliers for significant and non-significant terms respectively. For each significant

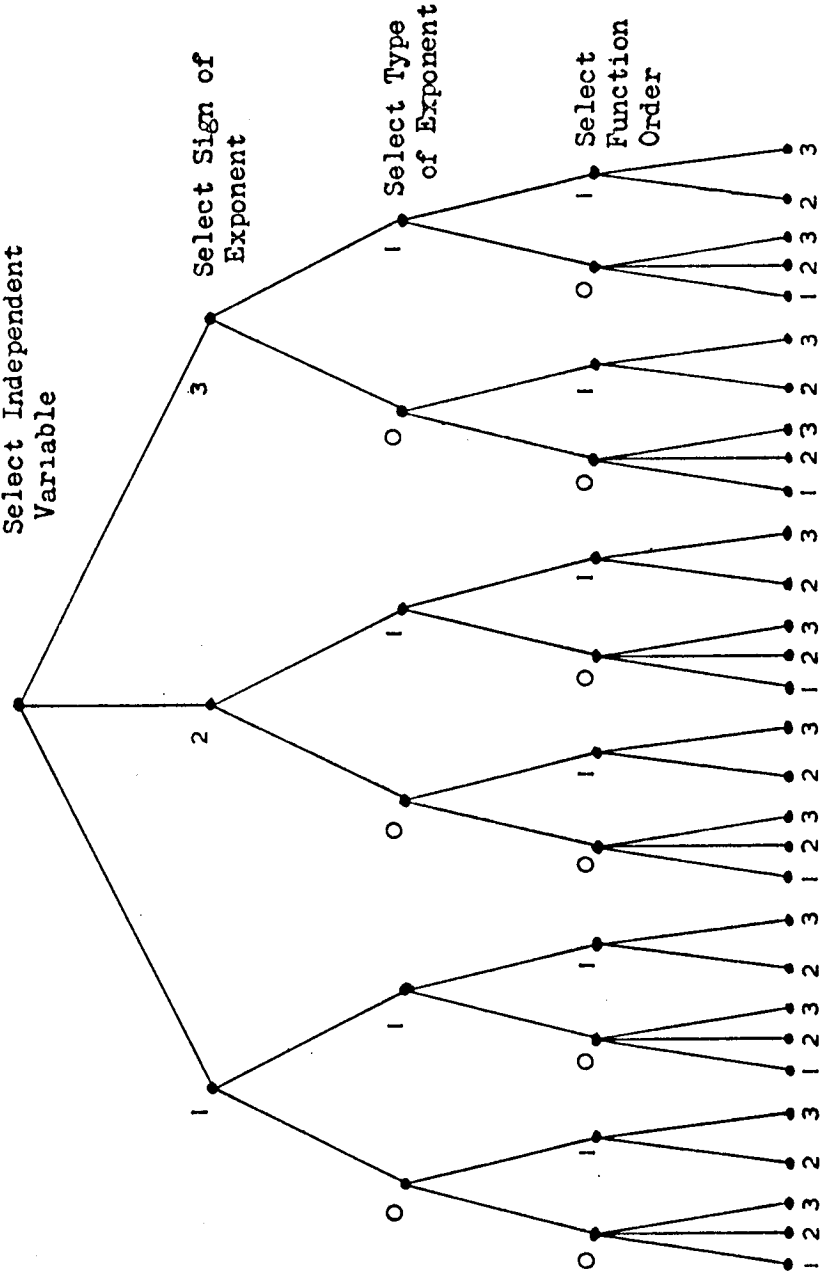


FIG. 3. Modified factor selection tree.

term with a given interaction order suppose there are n non-significant terms with a similar interaction order. When the term selector is modified, the selection probability for the given interaction order is multiplied by ab^n . For steady conditions $ab^n=1$ and hence

$$n = -\log(a)/\log(b)$$

If $b=1/a$ then $n=1$. This applies to the general regression program. Each significant term will on average be balanced by a non-significant term with the same interaction order. This property applies to other features of the terms as well as the interaction order. Thus the region surrounding the current area of interest is explored. If the number of significant terms is less than half the number of terms selected, there will be, on average, some extra terms selected with random properties. These terms explore the term space at greater distances from the current area of interest and thus help the search process to overcome the problem of multiple peaks.

ACKNOWLEDGEMENTS

The author is indebted to British Petroleum Limited for providing generous amounts of time on their Mercury computer for the development of the program described. Thanks are also due to Professor M. W. Humphrey Davies for his advice and encouragement while supervising the work.

REFERENCES

- Collins, J. S. (1964). 'The processing of lists and the recognition of patterns with application to some electrical engineering systems', Ph.D. Thesis, University of London.
- Dallemand, J. E. (1958). Stepwise regression program on the IBM 704, *General Motors Research Staff Report, GMR 199*.
- Kendall, M. G., & Stuart, A. (1961). *The Advanced Theory of Statistics*, Vol. 2. London: Griffin.
- Westervelt, F. H. (1960). 'Automatic System Simulation Programming', Ph.D. Thesis, University of Michigan.

APPENDIX 1

The following is an annotated example of some data prepared for the regression program.

Information Punched	Notes
TITLE J362, COLLINS, EXAMPLE A.	The line of text following TITLE is printed at the head of the results for identification.
VARIABLES=3	This problem involves three independent variables.
FUNCTION RANGES	Terms are made up from the following factors:
3 3 0 0	$x_1 x_1^2 x_1^3 1/x_1 1/x_1^2 1/x_1^3$
3 3 3 3	$x_2 x_2^2 x_2^3 1/x_2 1/x_2^2 1/x_2^3 x_2^{\frac{1}{2}} x_2^{\frac{3}{2}} 1/x_2^{\frac{1}{2}} 1/x_2^{\frac{3}{2}}$
3 0 3 0	$x_3 x_3^2 x_3^3 x_3^{\frac{1}{2}} x_3^{\frac{3}{2}}$
INTERACTION LIMIT=2	Only terms made up of up to 2 factors are to be selected.
PRINTING REQUIRED TRIALS EACH REGRESSION STEP	Terms generated for each trial and each step of the regression analysis are to be printed.
READ UNWEIGHTED DATA	This is the actual data on which the regression analysis is carried out. Each observation is set across a line consisting of the value of the dependent variables followed by the values of the three independent variables.
1.000 22.5 1.21 368	
2.450 24.5 1.30 924	
1.414 23.5 1.47 1145	
1.000 22.5 1.47 1120	
2.646 22.5 1.47 1161	
2.646 21.5 1.55 3124	
1.732 24.5 1.59 920	
2.450 22.5 1.67 444	
1.732 22.5 1.43 671	
0 22.5 1.50 0	
*	An asterisk terminates the data.

MACHINE LEARNING AND HEURISTIC PROGRAMMING

Information Punched	Notes
INITIAL TERMS 1 3 1 0 2 1 2 0 2 1 1 *	The user may suggest a set of terms to be included in the first trial. In this case the terms x_3 and x_1^2/x_2 are suggested. An asterisk terminates the list of initial terms.
ACCURACY=0.001	Indicates the accuracy at which analysis should stop.
TRIALS LIMIT=20	Stop anyway after 20 trials.
BEGIN	When read this causes analysis to begin.

APPENDIX 2

The following is a summary of the performance of the search procedure used in the regression program. Artificial data was generated by evaluating the expression

$$1 + 2x_1 - 3 \frac{x_1 \cdot x_3^3}{x_2} + x_4^2$$

for eighty sets of values of the independent variables (x_1, x_2, x_3 and x_4) chosen randomly in the range 0 to 10. The values of both the expression and the independent variables were punched on paper tape to an accuracy of three decimal places in a form suitable for use with the regression program. The following summary shows how the program searched through a large number of possible terms and found the original expression from which the data was generated.

Terms included in the search were made up of factors consisting of powers, reciprocals of powers, roots and reciprocals of roots each up to order 5. This allows 18 possible functions of each variable.

For each trial, 15 terms were selected, evaluated and tested by the regression procedure. Below are shown those terms in each trial which were found to contribute significantly in a regression equation. In each case, the standard error (the root mean square of the deviations between actual and predicted values) of the regression equation is given.

The terms found to be significant in one trial are always automatically included among the set of terms tested in the following trial. Such terms are marked with an asterisk in the commentary below.

Trial No.	Terms included in regression equation	St. error of Y	Notes
0	NONE	1254.5	The standard error in this case is the standard deviation of x_0 about the mean.
1	$\frac{x_1^{\frac{1}{2}} \cdot x_3^2 \cdot x_4^2}{x_2^{\frac{1}{2}}} \quad x_1^{\frac{1}{2}} \cdot x_3^3$ $x_1 \cdot x_2^{\frac{1}{2}} \cdot x_3^3 \cdot x_4^{\frac{1}{2}}$	754.30	Two out of the three terms found significant include the factor x_3^3 .
2	$*x_1^{\frac{1}{2}} \cdot x_3^3 \quad \frac{x_1^{\frac{1}{2}} \cdot x_3 \cdot x_4^2}{x_2}$ $\frac{x_1^{\frac{1}{2}} \cdot x_3^3}{x_2^3 \cdot x_4^{\frac{1}{2}}}$	645.54	The factor $1/x_2$ appears for the first time.

MACHINE LEARNING AND HEURISTIC PROGRAMMING

Trial No.	Terms included in regression equation	St. error of Y	Notes
3	$\frac{*x_1^{\frac{1}{2}} \cdot x_3^3}{x_2} \quad \frac{x_1^{\frac{1}{2}} \cdot x_3}{x_2 \cdot x_4^{\frac{1}{2}}} \quad \frac{x_1^{\frac{1}{2}} \cdot x_3^5}{x_2^4}$ $\frac{x_1^{\frac{1}{2}} \cdot x_3^3 \cdot x_4^{\frac{1}{2}}}{x_2} \quad x_1^{\frac{1}{2}} \cdot x_3^3$	280.0	The fourth term contains both the factors x_3^3 and $1/x_2$.
4	$\frac{*x_1^{\frac{1}{2}} \cdot x_3^5}{x_2^4} \quad \frac{*x_1^{\frac{1}{2}} \cdot x_3^3 \cdot x_4^{\frac{1}{2}}}{x_2}$ $*x_1^{\frac{1}{2}} \cdot x_3^3 \quad \frac{x_1^{\frac{1}{2}} \cdot x_3^3}{x_4^{\frac{1}{2}}} \quad \frac{x_3^4}{x_1^{\frac{1}{2}} \cdot x_2}$ $\frac{x_3^3}{x_2^{\frac{1}{2}}} \quad x_1^{\frac{1}{2}} \cdot x_3^3$	154.19	
5	$\frac{*x_1^{\frac{1}{2}} \cdot x_3^3 \cdot x_4^{\frac{1}{2}}}{x_2} \quad \frac{*x_3^4}{x_1^{\frac{1}{2}} \cdot x_2}$ $\frac{x_3^3 \cdot x_4^{\frac{1}{2}}}{x_2} \quad \frac{x_3^5}{x_1 \cdot x_2}$ $\frac{x_1^5 \cdot x_3^3}{x_2} \quad \frac{x_1^{\frac{1}{2}} \cdot x_3^3}{x_2}$	54.076	Three terms are near misses here including the factors x_3^3 , $1/x_2$ and a positive power or root of x_1 .
6	$\frac{*x_3^4}{x_1^{\frac{1}{2}} \cdot x_2} \quad \frac{*x_3^3 \cdot x_4^{\frac{1}{2}}}{x_2}$ $\frac{*x_3^5}{x_1 \cdot x_2} \quad \frac{*x_1^5 \cdot x_3^3}{x_2}$ $\frac{*x_1^{\frac{1}{2}} \cdot x_3^3}{x_2} \quad \frac{x_1^2 \cdot x_3^3}{x_2}$	31.864	Note how terms 4, 5 and 6 are 'converging' on one of the terms in the original equations.
7	$\frac{x_4^4}{x_2} \quad \frac{x_1 \cdot x_3^3}{x_2} \quad \frac{x_1^{\frac{1}{2}} \cdot x_3^{\frac{1}{2}}}{x_2}$	20.105	The second term is one of the three terms in the original equation.
8	$\frac{*x_1 \cdot x_3^3}{x_2} \quad \frac{x_1}{x_4^{\frac{1}{2}}} \quad \frac{x_3^{\frac{1}{2}}}{x_1^2}$ $\frac{x_3^{\frac{1}{2}} \cdot x_4^5}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}} \quad x_1$	10.900	A second term of the original equation (x_1) is now found.

Trial No.	Terms included in regression equation	St. error of Y	Notes
9	$\frac{*x_1 \cdot x_3^3}{x_2} \quad \frac{*x_3^{\frac{1}{2}}}{x_1^2} \quad \frac{*x_3^{\frac{1}{2}} \cdot x_4^5}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}}$ $\frac{x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4}{x_2^{\frac{1}{2}}}$	9.1994	x_1 is dropped out due to lack of significance.
10	$\frac{*x_1 \cdot x_3^3}{x_2} \quad \frac{*x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4}{x_2^{\frac{1}{2}}}$ $\frac{x_3^{\frac{1}{2}} \cdot x_4^{\frac{1}{2}}}{x_1^2} \quad \frac{x_4^2}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}}$ $\frac{x_4^4}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}} \quad \frac{x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2}$ $1/x_1^2$	5.0410	Positive powers of x_4 are beginning to appear.
11	$\frac{*x_1 \cdot x_3^3}{x_2} \quad \frac{*x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4}{x_2^{\frac{1}{2}}}$ $\frac{*x_4^2}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}} \quad \frac{*x_4^4}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}}$ $\frac{*x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2} \quad \frac{x_4^4}{x_1^2 \cdot x_2}$ $x_1^2 \quad \frac{x_2^{\frac{1}{2}} \cdot x_4^5}{x_1^2}$	4.3066	
12	$\frac{*x_1 \cdot x_3^3}{x_2} \quad \frac{*x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4}{x_2^{\frac{1}{2}}}$ $\frac{*x_4^2}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}} \quad \frac{*x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2}$ $\frac{*x_4^4}{x_1^2 \cdot x_2} \quad \frac{*x_1^2}{x_1^2} \quad \frac{*x_2^{\frac{1}{2}} \cdot x_4^5}{x_1^2}$ $\frac{x_1^{\frac{1}{2}} \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2^{\frac{1}{2}}}$	3.5597	

MACHINE LEARNING AND HEURISTIC PROGRAMMING

Trial No.	Terms included in regression equation	St. error of Y	Notes
13	$\frac{*x_1 \cdot x_3^3}{x_2} \quad \frac{*x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4}{x_2^{\frac{1}{2}}}$ $\frac{*x_4^2}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}} \quad \frac{*x_1 \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2}$ $*x_1^2 \quad \frac{*x_1^{\frac{1}{2}} \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2^{\frac{1}{2}}}$ $\frac{x_4^5}{x_1^2 \cdot x_2 \cdot x_3^{\frac{1}{2}}}$	3.5882	is slightly worse than that of previous trial but there is one less term in the equation.
14	$\frac{*x_1 \cdot x_3^3}{x_2} \quad \frac{*x_4^2}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}} \quad *x_1^2$ $\frac{*x_1^{\frac{1}{2}} \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2^{\frac{1}{2}}} \quad \frac{*x_4^5}{x_1^2 \cdot x_2 \cdot x_3^{\frac{1}{2}}}$ $\frac{x_3^{\frac{1}{2}} \cdot x_4^5}{x_1^{\frac{1}{2}} \cdot x_2}$	3.5365	Again, one less term than in the previous trial but increased average significance of each term.
15	$\frac{*x_1 \cdot x_3^3}{x_2} \quad \frac{*x_4^2}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}} \quad *x_1^2$ $\frac{*x_1^{\frac{1}{2}} \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2^{\frac{1}{2}}} \quad \frac{*x_3^{\frac{1}{2}} \cdot x_4^5}{x_1^{\frac{1}{2}} \cdot x_2}$ $1/x_1^{\frac{1}{2}} \quad \frac{x_4^2}{x_1^2 \cdot x_2 \cdot x_3^{\frac{1}{2}}}$	2.5800	Two terms now include x_4^2 .
16	$\frac{*x_1 \cdot x_3^3}{x_2} \quad \frac{*x_4^2}{x_1^{\frac{1}{2}} \cdot x_2^{\frac{1}{2}}}$ $*x_1^2 \quad \frac{*x_1^{\frac{1}{2}} \cdot x_3^{\frac{1}{2}} \cdot x_4^3}{x_2^{\frac{1}{2}}}$ $\frac{*x_3^{\frac{1}{2}} \cdot x_4^5}{x_1^{\frac{1}{2}} \cdot x_2} \quad \frac{*x_4^2}{x_1^2 \cdot x_2 \cdot x_3^{\frac{1}{2}}}$ $x_1^{\frac{1}{2}}$	2.5001	
17	Same as 16		

COLLINS

Trial No.	Terms included in regression equation	St. error of Y	Notes
18	$\frac{*x_1 \cdot x_3^3}{x_2} \quad *x_1^2 \quad *x_1^{\frac{1}{2}}$ $x_4^2 \quad \frac{x_3^{\frac{1}{2}}}{x_1^2 \cdot x_2}$	0.5302	The term x_4^2 of the original equation has now been found.
19	$\frac{*x_1 \cdot x_3^3}{x_2} \quad *x_4^2 \quad x_1$	0.3635	All the original terms are found. Analysis is complete. The standard error is not zero because of small random errors introduced into the original data by truncating each figure to 3 figures accuracy.

MACHINE LEARNING AND HEURISTIC PROGRAMMING

A graph of the logarithm of the standard error against the trial number is shown in Fig. 4. The graph shows the steady improvement of accuracy obtained by the process. This is a useful feature when the best regression equation for a given computational effort is required.

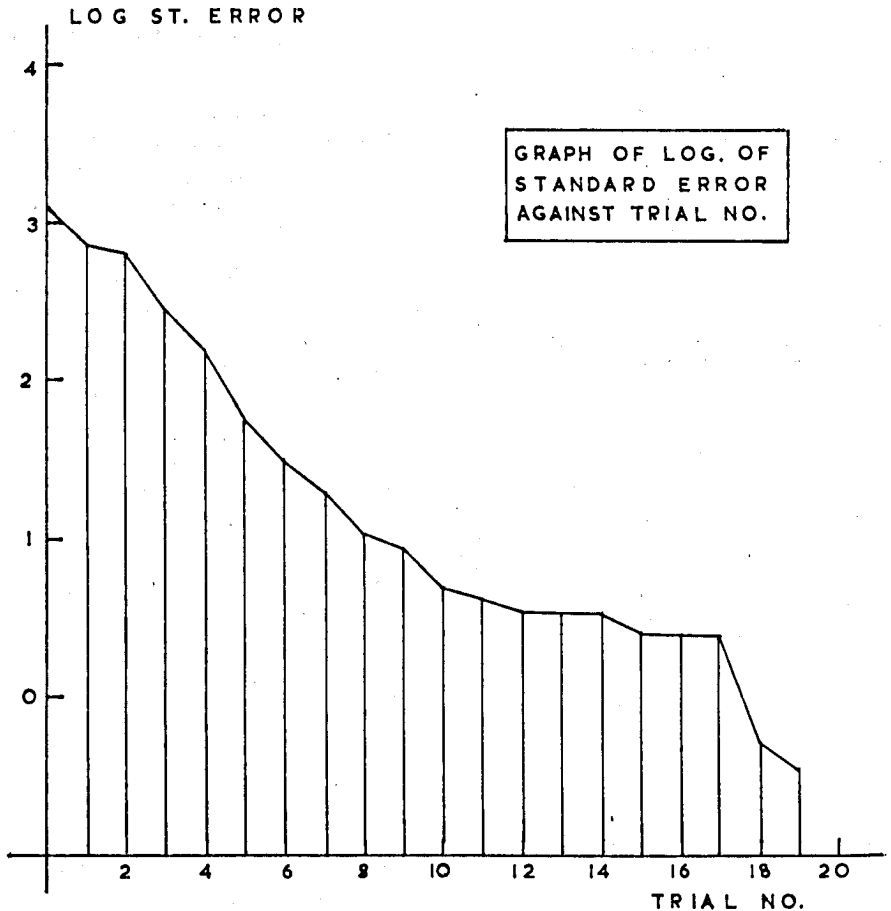
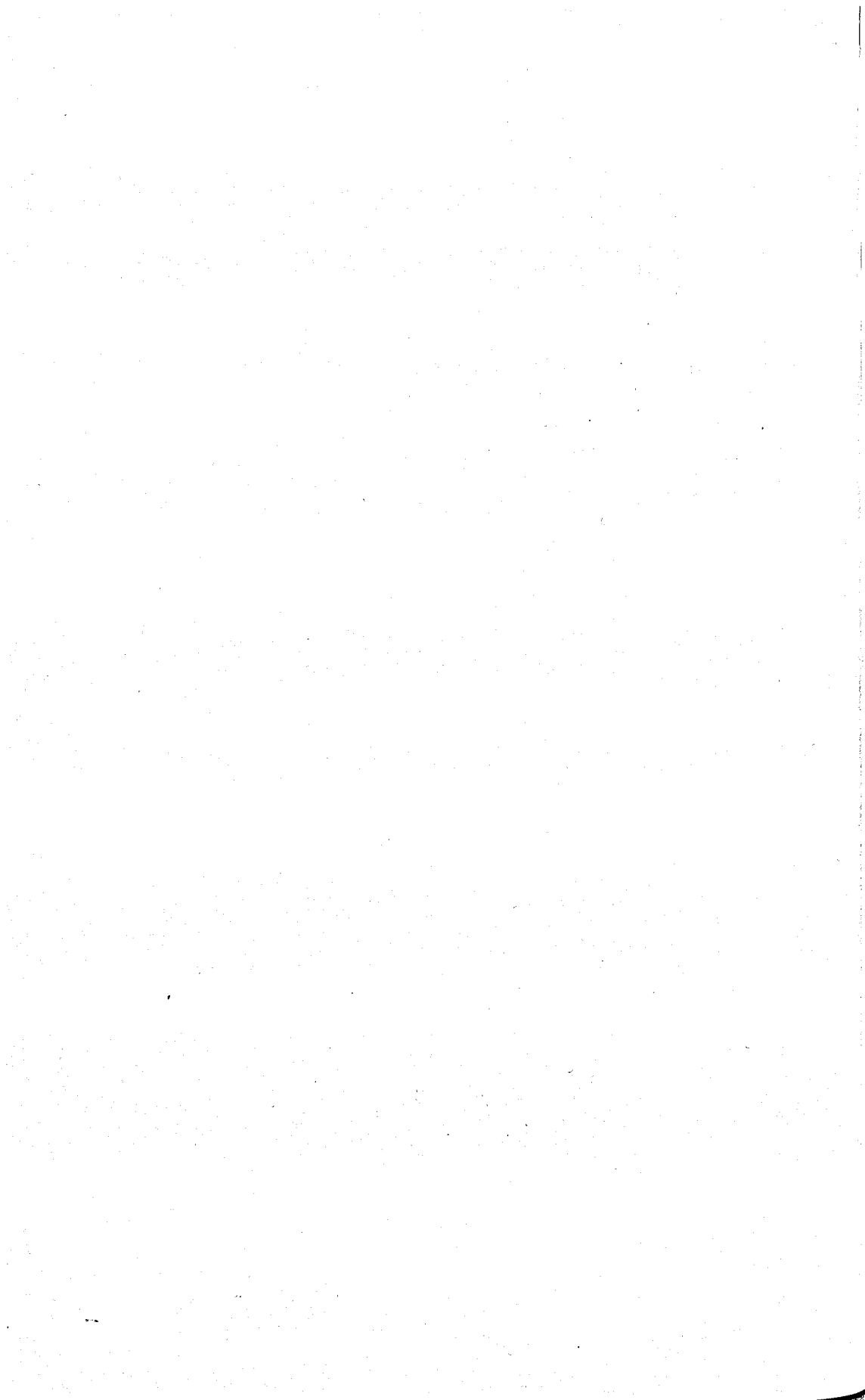


FIG. 4. The progressive improvement of the regression equation.

The analysis described above required approximately 45 minutes on the Ferranti Mercury computer, during which only 181 out of 130 320 possible terms were tested.

COGNITIVE PROCESSES: METHODS AND MODELS



A LIMITED DICTIONARY FOR SYNTACTIC ANALYSIS

P. BRATLEY and D. J. DAKIN *

DEPARTMENT OF COMPUTER SCIENCE AND ENGLISH LANGUAGE
RESEARCH UNIT
UNIVERSITY OF EDINBURGH

INTRODUCTION

Work is at present proceeding at the English Language Research Unit of the University of Edinburgh on the construction of a computer model for the perception of syntactic structure (Thorne *et al.* 1966). One of the basic principles behind the construction of the model is that it should be possible to analyse sentences without recourse to a complete dictionary of the language under consideration (Thorne 1964). However, it is not possible to dispense with a dictionary completely, as certain words (such as, for instance, determiners, prepositions and so on) play a role of major importance in the syntax. These words, which we call closed-class words, must be listed in a closed-class dictionary, and it is with the production of this dictionary that the present paper is concerned.

Consider for a moment the well-known lines:

‘Twas brillig, and the slithy toves
Did gyre and gimble in the wabe.’

It is hard to maintain that the lines actually mean anything even if one knows that *toves*, for instance, are little animals that live under sundials (or is that borogroves?). Nevertheless, it is not at all unreasonable to assert that the lines have a readily perceived syntactic structure. In everyday terms, we might say that *slithy* is clearly an adjective, *tove* is a noun, and *gyre* and *gimble* are verbs.

* The work described is supported in part by a DSIR grant, contract number ID/102/2/60.

Notice what we are doing: we are not saying, 'I know that *tove* is a noun, therefore the syntactic structure is so-and-so.' On the contrary, we are saying, 'The syntactic structure is apparent, and knowing the structure, I can now assert that *tove* is a noun.'

On the other hand, suppose we are faced with the line:

William the the and disappointed.

No syntactic structure is now recognisable, even though all the words in the line are well known to us.

We maintain that the fact that the first example apparently has a syntactic structure, while the second example does not, is explained as follows. Certain words in English, for example *the*, *and* and *in*, carry a great deal of information about the syntactic structure of the sentences in which they occur. They are called closed-class words, on the grounds that there are, for instance, only a finite number of determiners in the language, and only a finite number of conjunctions. On the other hand, words such as *iron*, *book* or *green* play a very indeterminate part in the syntactic structure, so that *iron*, for instance, may equally well be a verb, a noun or an adjective. Words such as these are called open-class words, since one can imagine that the class of nouns, say, has an infinite number of members.

The syntactic structure of a sentence is determined by the closed-class words in that sentence, and the parts that the open-class words are playing on any particular occasion can be discovered from the structure thus determined. So the first example above is, syntactically at least, perfectly normal; but the second example has the closed-class words misused, so no structure is discoverable.

A computer model based on this principle of the distinction between open- and closed-class words has been developed at Edinburgh, and is now working on KDF9. Each word of an input sentence is looked up in a closed-class dictionary, to see if it is one of the words used to determine structure. If it is, it is coded appropriately. If not, the word is simply coded as an open-class item (although as described below some notice is taken of word-endings, so that 'open-class item ending with -s', for instance, is coded differently from a simple open-class item). For example, if the input sentence were

The old grey goose is dead,

the successive words of the sentence would be coded, roughly, as

DET O/C O/C O/C SING. VERB O/C.

Of this sentence, only the words *the* and *is* need appear in the dictionary.

We estimate that a closed-class dictionary for English, to be used in this way as part of a model for the perception of syntactic structure, would need to contain at most about 2000 items. As open-class words are not listed at all, however, it will be clear that the model will be capable of dealing with sentences from an infinite vocabulary: for example, new nouns or new verbs make no difference to its functioning whatsoever.

After this very brief introduction to the ideas of open-class and closed-class, and to the idea of a closed-class dictionary being used in a model for syntax recognition, the rest of the paper will deal with some of the decisions we have taken and the problems we have met in endeavouring to construct such a dictionary.

WORD-ENDINGS

Word-endings are of two types, traditionally known as *inflection* and *derivation*. Inflection refers, among other things, to the final *-s* of plural nouns, the possessive case endings, and the various forms of a verb. Derivation refers to the processes by which endings such as *-ise*, *-ment* and *-ish* are added to existing words to form new verbs, nouns and adjectives. The two kinds of ending do not seem to be equally important. For the analysis of the sentence,

- (1) The boys fish,

information about the closed-class item, *the*, and about inflected word-endings is quite sufficient. The dictionary must contain entries to the effect that:

- (a) *the* is a determiner which may be followed by a singular or a plural noun;
- (b) an open-class item ending in *-s* may be the plural form of a noun or the singular form of a verb;
- (c) an uninflected open-class item (i.e., one without any inflected ending) may be the singular form of a noun or the plural form of a verb. (For the sake of simplicity we temporarily ignore the fact that it may also be an adjective or the first person singular form of a verb.)

With this information the program can determine that *the boys* must be the subject of the sentence, that *boys* must be a plural noun, and that *fish* must be the verb. The same information will correctly permit two analyses for:

- (2) The girl guides fish.

One analysis will treat *the girl guides* as the subject; the other will treat *the girl* as the subject and *guides* as a verb.

So far no information about derivational endings has been required. It is not clear that such information would ever be decisive. Let us consider the ending *-ment* as an illustration. Most words ending in *-ment* are nouns. If we were to include information to this effect in the dictionary, we would also have to include a list of exceptions like *vehement* which can never be nouns, and a longer list of words like *comment* and *torment* which can be either nouns or verbs. Lists of this kind are redundant in the case of sentences like:

- (3) They torment fish,
- (4) They were in torment.

Information about the closed-class items *they*, *in* and *were* is sufficient to determine whether *torment* is to be regarded as a noun or a verb. In the more critical case where information drawn from closed-class items and inflectional

endings cannot prevent an ambiguous analysis, derivational information is unlikely to remove the ambiguity. In this sentence, for example,

- (5) The boy torments fish,

the derivational information that *torments* may be a noun or a verb reveals no more than what can already be discovered from the inflected ending *-s*. It cannot prevent an analysis of the sentence treating *the boy torments* as a subject of the same structure as *the girl guides* in one analysis of sentence (2).

Information about closed-class items and inflectional endings must clearly be included in the dictionary. So, of course, must exceptions to any mechanical routine for identifying inflected endings. Words like *lens* and *hatred* can scarcely be treated as the singular and past forms of particular verbs. Information about derivational endings, on the other hand, seems at present dispensable. There are, however, two exceptions. The endings *-ly* and *-able* cannot be ignored. If they were, *accidentally* would be treated as an uninflected noun in this sentence

- (6) The boy accidentally slipped

on a par with

- (7) The boy scout slipped.

The ending *-ly* is therefore included in the dictionary as an ending for adjectives and adverbs, and all exceptions are listed. *Accidentally* would not, of course, be treated as an adjective in sentence (6), since the program contains a general rule to the effect that adjectives do not follow nouns within the same noun phrase. The few possible exceptions to this rule like *opposite* are listed in the dictionary, and so are the derivational endings *-able* and *-ible*. *Available* must obviously be treated as an adjective in

- (8) He saw all the men available

and not as an uninflected verb on a par with *die* in

- (9) He saw all the men die.

CLOSED-CLASS ITEMS

Reference has already been made to relatively small classes of closed-class items such as pronouns, prepositions, and conjunctions. These must be exhaustively listed in the dictionary for two reasons. They must not be treated as uninflected forms of nouns and verbs. They also provide specific information which can be used to determine the structure of sentences. The information contained in traditional labels such as *pronoun*, *preposition* is not, however, specific enough. The word *her*, for instance, might be traditionally classified as a pronoun and as a possessive article. But it certainly could not be a pronoun in this sentence:

- (10) Her dislikes increase as she grows older,

or a possessive article in this:

- (11) I saw her.

To prevent such analyses being made, the dictionary must contain information about *her* which indicates that as a pronoun it cannot be the subject of a verb, and as a possessive article it must be followed by a noun. The functions of words traditionally known as pronouns, articles, possessives, determiners, and so on, are specified in the dictionary by code-words, which indicate whether, among other things, they

- (a) may begin a subject noun phrase, e.g., you, she, someone, some, this, open-class items;
- (b) may begin an object noun phrase, e.g., you, her, some;
- (c) may continue a noun phrase, e.g., own, last, open-class items;
- (d) are followed by singular verbs, e.g., this, he, someone;
- (e) are followed by plural verbs, e.g., these, you, open-class items ending in -s;
- (f) may be followed by either singular or plural verbs, e.g., the, some, her, open-class items ending in -'s.

The analysis section of the program has as input simply the code-words associated in the dictionary with the actual words in the input text.

The traditional labels *preposition*, *conjunction* and *adverb* also seem too unspecific. They do not suggest, for example, that words so labelled have at least one function in common. The ambiguity of sentence (2),

- (2) The girl guides fish,

disappears in

- (12) The girl in the brown hat guides fish,
- (13) The girl guides often fish,
- (14) The girl who you noticed guides fish.

Prepositions, *conjunctions* and some kinds of *adverb* all reveal, among other things, where a particular noun phrase ends. On the other hand, not all words traditionally labelled as *prepositions*, for instance, share identical functions. If *over* and *at* were simply listed as prepositions, we would not be able to provide two analyses for the ambiguous sentence

- (15) He looked over the houses,

but only one for

- (16) He looked at the houses.

At is clearly different from *over* in that it must be followed by a noun phrase and cannot be separated from the verb as *over* is in

- (17) He looked the houses over.

Different conjunctions also have different functions. Both *whenever* and *before* can introduce whole clauses:

- (18) He stopped work whenever he felt tired,
- (19) He stopped work before he felt tired.

Before, however, cannot be followed simply by an adjective as *whenever* can in

- (20) He stopped work whenever possible,
and neither *before* nor *whenever* can introduce a subject clause, as *how* can in
(21) How he does so much work is a mystery.

The traditional label *adverb* is scarcely more satisfactory. Distinctions between adverbs of time and place do not seem to provide much information about the structure of sentences. Only one form of time adverbial need be specially noted: phrases like *the next day* and *that morning* which can function in the same circumstances as a time adverb like *yesterday*. In the sentence,

- (22) The next morning was fine,
the next morning must be analysed as the subject of the sentence. But it is certainly not the subject in

- (23) The next morning they parted friends,
and information about such possible time expressions must be entered in the dictionary to permit the correct analysis.

Adverbs of manner, on the other hand, need to be distinguished not so much from adverbs of time or place, as from sentence adverbs and intensifiers. (Intensifiers are words like *very* which can modify adjectives; sentence adverbs are words like *probably* which can modify whole sentences.) The distinction between these three kinds of adverb can be used to resolve ambiguities of the kind found in

- (24) He was trying.

Here *trying* may be either an adjective or a verb. There is no ambiguity, however, in these sentences:

- (25) He was very trying,
(26) He was strenuously trying.

Intensifiers can only modify adjectives, while adverbs of manner like *strenuously* can only qualify verbs. The two adverbs cannot be exchanged in

- (27) He was very happy,
(28) He was strenuously advancing.

However, a sentence adverb like *probably* can occur in either sentence.

- (29) He was probably happy.
(30) He was probably advancing.

Sentence adverbs are thus neutral with respect to words ending in *-ing*, and

- (31) He was probably trying

is just as ambiguous as sentence (24). Intensifiers and sentence adverbs must be listed in the dictionary. Manner adverbs like *hard* and *fast* must also be listed, but those ending in *-ly* need not be inventoried, as the ending is sufficient to identify them.

SPECIAL CLASSES OF VERB

The dictionary must also contain information about certain classes of verb. The ambiguity in sentence (24) arises partly because the verb *to be* may be followed either by an adjective or by a present participle. A different kind of ambiguity occurs in

(32) They can fish,

where *can* is either an auxiliary or a normal verb meaning *to put in cans*. Auxiliaries like *be*, *can*, *have*, and *ought* must obviously be listed in the dictionary together with information about what endings a main verb that follows them must take.

Auxiliary verbs can reasonably be considered as closed-class items. Other classes of verb, which are not *closed* in quite the same sense, must, for practical reasons, also be listed in the dictionary. Consider the problem posed by the ambiguities in:

(33) He made a good dinner for the cannibal,

(34) He proved a problem.

In these sentences, *make* and *prove* may be acting as normal verbs followed by an object, or as copulative verbs taking a complement, like *be* in

(35) He was a problem.

We could obtain the necessary two analyses for sentences (33) and (34) by including a general rule in the program that any open-class verb may be followed either by an object or by a complement. But then we would also obtain one undesirable analysis for

(36) He solved a problem,

in which *solve* is treated as a potentially copulative verb. One way of preventing such undesirable analyses is to list the small class of possible copulative verbs in the dictionary.

A different but parallel problem is exemplified in this loose paraphrase of sentence (33):

(37) The cannibal found him an excellent meal.

In this sentence, *find* can either be a double-object verb like *give* in

(38) The cannibal gave him an excellent meal,

or an object-complement verb like *consider* in

(39) The cannibal considered him an excellent meal.

Once again it would lead to a number of undesirable analyses if a general rule were included in the program that unlisted verbs may be followed by one object, by two objects, or by an object and a complement. This becomes apparent when we compare

(40) He ate his dog biscuits

with

(41) He gave his dog biscuits.

Information about word-endings reveals only that the phrase *his dog biscuits* contains a possessive article followed by an uninflected open-class item and an open-class item ending in *-s*. It might consist of either one or two noun phrases and the general rule about unlisted verbs would accordingly assign three analyses to each sentence. The number of undesirable analyses is reduced if double-object and object-complement verbs are listed in the dictionary and the general rule for unlisted verbs is that they may be followed by only one object. *Give* and *find*, but not *eat*, then appear as possible double-object verbs. *Find* and *consider*, but not *give* or *eat*, appear as object-complement verbs. The ambiguities of Sentences (37) and (41) are still detected, but sentences (38), (39) and (40) correctly receive only one analysis.

Make has already been noted as a verb which can take either a normal object or a copulative complement. It can also take a double-object as in

(42) She made him a table,

and an object followed by a complement as in

(43) She made him what he is today.

It gives rise to yet another kind of ambiguity in

(44) She made him fish for his supper,

where *make* may be either a double-object verb as in sentence (42) or an object-infinitive verb as in

(45) She made him cry.

Object-infinitive verbs must also be listed in order that they can be distinguished both from other special classes of verb and from normal verbs. Such a distinction enables the program to determine the potential difference in structure between

(46) She lets her house

and

(47) She lets her cry

where the phrases *her house* and *her cry* both have the same superficial appearance of a possessive article followed by an uninflected open-class item.

Two additional classes of special verb might be listed, as they give rise to ambiguity when followed by words ending in *-ing*. This ending has already been noted as characteristic of both verbs and adjectives. It can also act as an ending for words which behave as nominal elements. Thus *reading* is behaving as a nominal element in

(48) Reading is fun,

(49) She writes reading books for children,

but in

(50) Some children avoid reading books

the structure is ambiguous and *reading* may be either a nominal element or a verb. Verbs such as *avoid* and *like*, which permit this kind of ambiguity, can be specially listed. *Like* and several other verbs may also give rise to ambiguity when they are followed by a possessive. In the sentence

(51) I don't like your playing cards,

playing may be either a nominal element or a verb. If ambiguities of this kind are to be detected, and the differences in structure are to be revealed, the most effective procedure seems to be to list in the dictionary information about all the special classes of verb discussed in this paper, and a few other classes not specifically mentioned (see Alexander & Kunz 1964; Pride 1965).

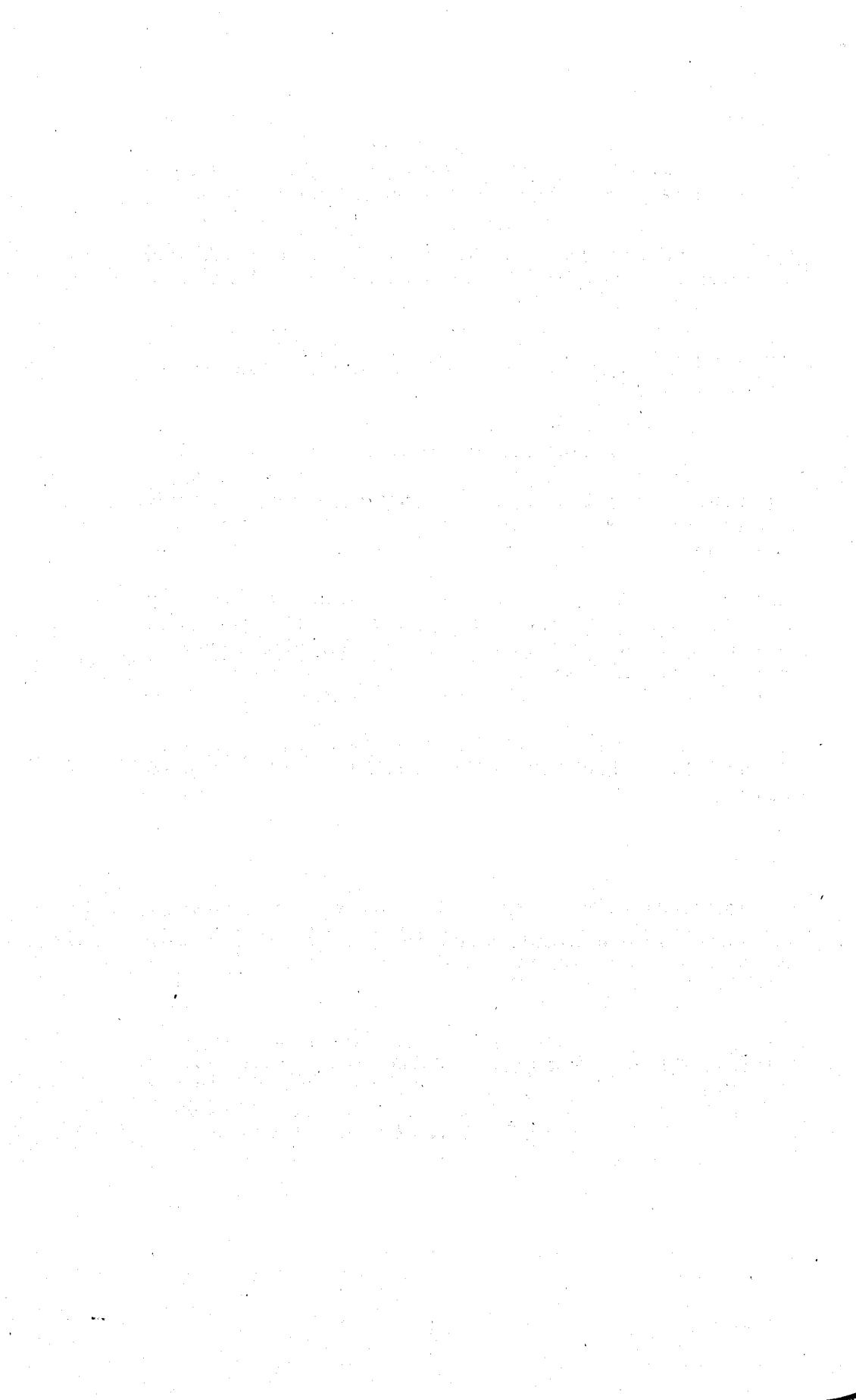
CONCLUSION

This paper has discussed the function of a dictionary in a program for parsing sentences. The program is required to assign at least one satisfactory analysis to every sentence, and more than one analysis where a sentence is, on syntactic grounds, potentially ambiguous. Since the program contains only abstract information about the structure of sentences, it must consult a dictionary to determine the relevant syntactic properties of the actual words in each sentence. The dictionary need only contain information about inflectional word-endings, about exceptions to the routine for determining word-endings, and about words which may give rise to ambiguity or which have otherwise undiscoverable syntactic functions. Such words are listed as special verbs or closed-class items.

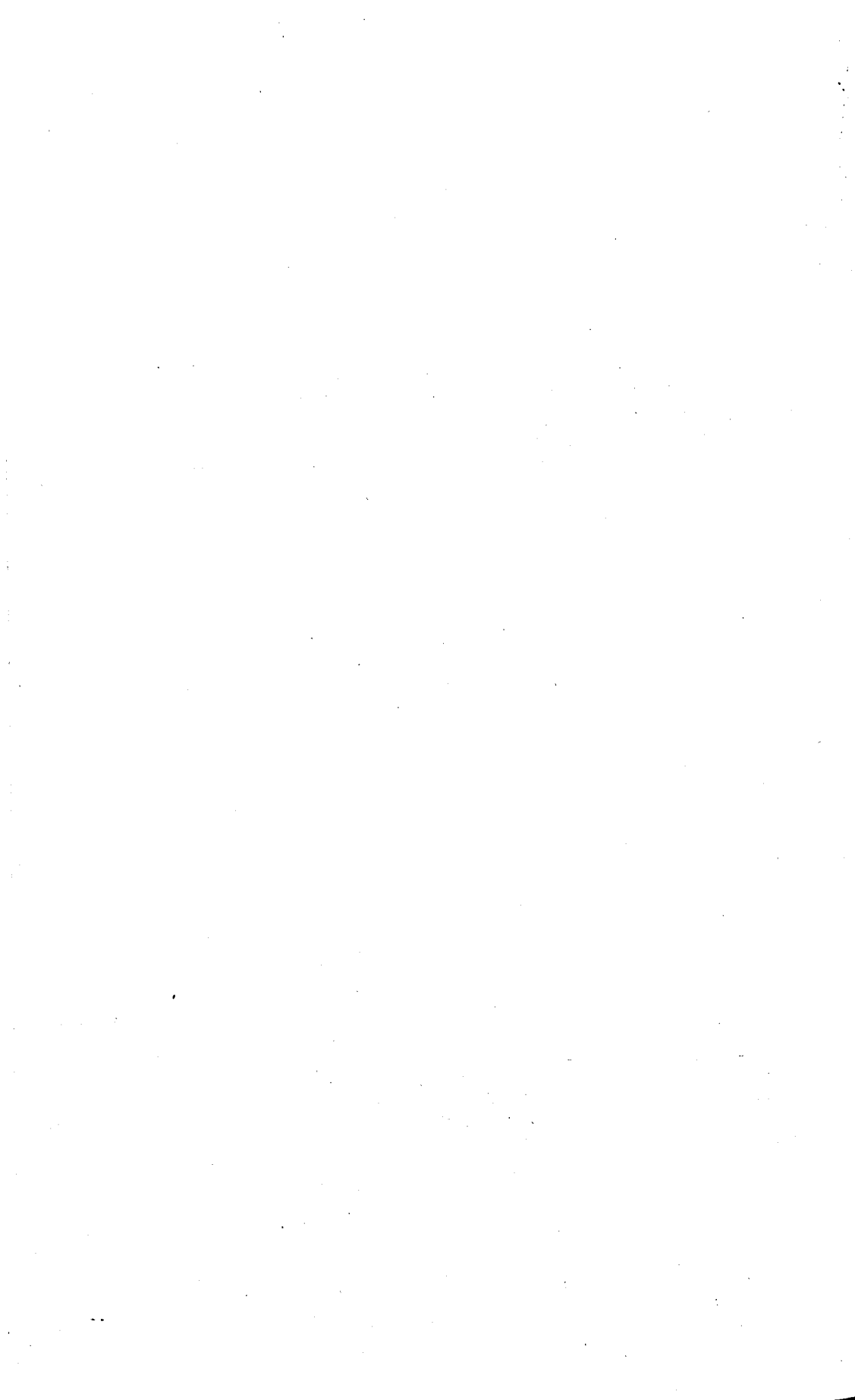
The dictionary does not resemble a normal dictionary in either function, content or length. It can be rapidly inspected for each word in any new sentence. Yet the arrangement of information in the dictionary enables the program to handle sentences containing a potentially infinite range of vocabulary.

REFERENCES

- Alexander, D., & Kunz, W. J. (1964). Some classes of verbs in English, *Linguistics Research Project mimeo*, Indiana University.
 Pride, J. B. (1965). 'Some Classes of English Verbs', *University of Leeds mimeo*.
 Thorne, J. P. (1964). Grammar and machines, *Transactions of the Philological Society*.
 Thorne, J. P., Dewar, H., Whitfield, H., & Bratley, P. (1966). A model for the perception of syntactic structure, *Colloque International sur l'Informatique*. Toulouse.



PROBLEM-ORIENTED LANGUAGES



12

POP-1: AN ON-LINE LANGUAGE

R. J. POPPLESTONE

EXPERIMENTAL PROGRAMMING UNIT
DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION
UNIVERSITY OF EDINBURGH

POP-1 is an on-line computer language, whereas most of the well-known languages (FORTRAN, MAD, ALGOL, LISP, etc.) are designed for off-line use. POP-1 is for use by a person communicating directly with a computer *via* a typewriter. It differs from JOSS (Shaw 1964) in that it is primarily intended for the sophisticated computer user.* With this in mind I have aimed at a tolerable efficiency of execution, and an ability to define and name new operations, with comprehensive monitoring facilities. On the other hand actual error messages are rather simple.

At the very lowest level, the computer can be used just as a desk calculator. For example

```
23 + 54 + 72 + 98 ⇒
```

```
** 247
```

Lines preceded by double asterisks represent program output.

If the user wants to perform a number of operations using the same number, he can declare a variable to hold it.

```
VARs A;
```

```
24.56 → A;
```

```
97 * A ⇒
```

```
** 2382.32
```

```
(27 * A + 3) / (17 - 5 * A) ⇒
```

```
** -6.2677
```

* See also MAP (Kaplau, Strong & Brackett 1966).

PROBLEM-ORIENTED LANGUAGES

All operations are organised by means of a stack, and this basic feature shows up explicitly in the case of two types of instruction:

- (i) 'argumentless' output commands (for example $\Rightarrow$ by itself will cause the number at the top of the stack to be printed);
- (ii) 'assignmentless expressions' which are interpreted as implicit assignments to the top of the stack.

With the above examples in mind, certain principles seem obvious.

(i) An on-line language must provide a way of returning the computer to a standard state no matter what state it is in at any time (analogous to the operation of clearing a desk calculator).

(ii) If an error is detected, a message as to the nature of the error must be displayed, and the machine must then enter a clearly defined state, with as little damage as possible to the results already built up.

The on-line user can make best use of such a system by building up complex entities in small units. For example, when calculating a large expression, it is better to work out parts of it and store these parts in variables, rather than try to do the whole thing at once.

In the above examples POP-1 has appeared as a language with a fixed vocabulary. In fact both the vocabulary, and to a limited extent the grammar, are extensible. In the terms of the earlier analogy it is as though we had a calculating machine equipped with an indefinite number of spare keys, on which we can stick new labels. For example, we can define an operation to find the HCF of two numbers by

```
FUNCTION HCF X Y
  VARS QUOTIENT REMAINDER
  DEFINE X//Y→REMAINDER→QUOTIENT;
  IF REMAINDER=0 THEN Y EXIT
  HCF(Y, REMAINDER) END
```

Where // produces two values, remainder and quotient. The instinctive reaction of the on-line user, having defined a function, is to test it:

```
HCF(12, 16)⇒
** 4
```

One may decide at this stage that it would be preferable to use an in-fixed operator, say ++ instead of HCF. To do this we need to re-assign the 'value' of the operator HCF, thus:

```
VALUE HCF→VALUE ++;
SETPRECEDENCE("++", 12, "SYMMETRIC");
```

The symbol 'SYMMETRIC' gives the signal that the operator '+' is to be used as an infix.

```
25 ++ 35⇒
** 5
```

The purpose of the SETPRECEDENCE operation is to declare ++ as being an

POPPLESTONE

infix operator with precedence 12. Thus addition (precedence 6) and multiplication (precedence 5) will be done before ++, i.e.,

$4 + 6 + + 7 * 5 \Rightarrow$
 $**5$

The line `VALUE HCF→VALUE ++` simply assigns to ++ the definition of HCF already established.

POP-1 grammar is a simple precedence grammar. Functions can be written either before their arguments as `HCF(X, Y)` or after as in `X Y HCF`.

Functions can be redefined at any time. This is obviously important for an on-line language, since the first definition of a function may be incorrect. In fact, a function definition is no more than an assignment of a constant (the definition) to the name. Thus one could in fact redefine +, *, -, / to have meanings over some field other than the reals (e.g., the complex numbers or a finite field).

For instance to do this for a finite field of characteristic P one writes:

```

VARS P
FUNCTION ADD X Y
  DEFINE ADD(X, Y) // P→X→Y; X END
  VALUE ADD VALUE+ →VALUE ADD →VALUE+;
```

Because of the stack basis of this sequence, the result is to interchange the values of 'ADD' and '+' and similarly for SUB and MULT. Note that the last line has the desired effect of exchanging the values of 'ADD' and '+' because it consists of a sequence of assignments first to, and then from, the stack. Further, since it is executed before the function definition itself is entered (which occurs on the first occasion that the function is called) the operator use of ADD in 'DEFINE ADD (X Y)' already has the meaning previously possessed by +. It is necessary to introduce the names ADD SUB MULT DIV because the old definitions of these operations are needed by the new ones.

Now we get, for example,

$7 \rightarrow P;$
 $4 + 5 \Rightarrow$
 $** 2$

DIV can be defined using the euclidean algorithm.

Algorithm:

```

FUNCTION DIV X Y DEFINE X * RECIPROCAL (Y) END
FUNCTION RECIPROCAL X
  VARS U V H
  DEFINE X P EUCLID →U →V →H;

  IF NOT (H=1) THEN NL TEXT P NOT PRIME; EXIT
  U END
  FUNCTION EUCLID X Y
  VARS QUOTIENT REMAINDER U V H
  DEFINE X//Y→REMAINDER →QUOTIENT';
```

PROBLEM-ORIENTED LANGUAGES

```
IF REMAINDER=0 THEN Y, 1, 0 EXIT
Y REMAINDER EUCLID →U →V →H;
H, U SUB(U, MULT (V, QUOTIENT)), V END
```

The symbol NL means 'new line', i.e., it is a formal function.

The effect of VALUE is to make the next word be treated as a variable. Its opposite is OBEY which makes the next word be treated as a function.

ARRAYS

These are treated as functions with a difference—the difference being that they have a meaning when preceded by an → (apart from the above meaning). Declaring that a variable is to hold an array is distinct from assigning the space to hold that array. The word ARRAY in any list of variables indicates that all following variables are to be treated as arrays (until NORMAL or FUNCTION or the end of the list is reached). Rectangular arrays are created by the function NEWARRAY. Thus, to handle permutations, one might write the following group of functions:

```
VARS LENGTH;
FUNCTION READPERM
VARS ARRAY X NORMAL I
DEFINE 1→I; <<1 LENGTH>>NEWARRAY→VALUE X;
L 1: NEXTATOM →X(I);
IF I=LENGTH THEN VALUE X EXIT
I+1→I; GOTO NEXT END
```

The brackets << >> enclose items which are to be evaluated and then made into a list

```
FUNCTION MULTPERM ARRAY X Y
VARS I ARRAY Z
DEFINE 1→I; 1 LENGTH NEWARRAY→VALUE Z;
NEXT: I X Y→Z(I);
IF I=LENGTH THEN VALUE Z EXIT
I+1→I; GOTO NEXT END
3→LENGTH;
VARS A B C D
READPERM 1 2 3→A; READPERM 3 2 1→B; READPERM 1 3 2→C;
VALUE MULTPERM →VALUE *;
A * B⇒
** 3 2 1
B * C⇒
** 2 3 1
```

Since arrays are created by function, one can write functions to create special arrays, e.g., in a chess program

```
FUNCTION NEWBOARD
VARS I J K ARRAY A
```

POPPLESTONE

```

DEFINE <<1 8 1>> NEWARRAY→VALUE A;
I→I;
L1: IF I> 8 THEN VALUE A EXIT
I→J;
L2: IF J> 8 THEN I+ 1→I GOTO L1
ALSO (I MASK 1)+(J MASK 1) MASK 1 THEN "BLACK" ELSE "WHITE"
ALSO →A(I, J), J+ 1→J GOTO L2 END.

```

Quotation marks enclose atoms. MASK forms the logical AND of its arguments considered as bit patterns. When the conditional THEN ... ELSE ... ALSO is obeyed, the commands between THEN and ELSE are obeyed if the top of the stack is 'true' (i.e., non zero) before THEN. Otherwise the commands between ELSE and ALSO are obeyed. ALSO is thus used as a terminator of conditionals rather similar in action to the semi-colon which is used to terminate assignments.

LIST PROCESSING

POP-1 was originally a pure list-processing language, and list-processing facilities are well developed. Let us consider the function REV to reverse a list.

```

FUNCTION REV X
DEFINE IF X ATOM THEN X EXIT
REV(X TL) <> X HD :: 1 END

```

The operator <> means join, i.e., concatenate the lists which are its operands, and :: is CONS in LISP.

Let us translate some dog-English into dog-French. Suppose we have read an English sentence as a list, and suppose we have a dictionary DIC giving what part of speech each English word is, and its French equivalent, together with information as to whether a word (if it is an adjective) comes before or after the word it qualifies. If a sentence is defined as a nounphrase followed by a transitive verb followed by a nounphrase then:

```

FUNCTION SENTENCE X D
VARS U V T
DEFINE NOUNPHRASE (X, D)→T→V;
"SENTENCE" :: V <> LOOKUP (T HD, D) HD <> (T TL NOUNPHRASE→U)::1;
U END
FUNCTION NOUNPHRASE X D
VARS U1 V T
DEFINE LOOKUP(X HD, D)→U;
IF U HD="NOUN" THEN "NOUNPHRASE" : U :: 1; X TL EXIT
NOUNPHRASE(X TL, D)→T→V;
V <> U :: 1; T END

```

These functions work by finding the first syntactic unit of the type that bears their name in the list x, giving as a result an analysis of that unit, and whatever remains in the list x. Having produced a parsed version of the sentence, we translate it with the following function:

PROBLEM-ORIENTED LANGUAGES

FUNCTION TRANSLATE X G

DEFINE

```
IF X HD="SENTENCE" THEN TRANSLATE (X TL HD, G); FRENCH (X TL TL
HD) SPR;
TRANSLATE (X TL TL TL HD, G) EXIT
IF X HD="NOUN" THEN FRENCH (X) SPR EXIT
IF X HD="ADJECTIVE" THEN IF G="MASCULINE" THEN FRENCH (X) HD SPR
EXIT
FRENCH (X) TL HD SPR EXIT
```

```
IF X HD="NOUNPHRASE" THEN GENDER (X TL HD)→G;
TRANSSELECT (X TL YL, G, "ARTICLE");
TRANSSELECT (X TL, TL, G, "BEFORE");
TRANSLATE (X TL HD, G);
TRANSSELECT (X TL TL, G, "AFTER")
EXIT
NL "FAIL" PR END
```

This works as follows. *x* is the tree representing the sentence, or a part of it, in the parsed form, containing all the information necessary for a translation. *G* is the gender of the part of the sentence being translated, if this is relevant. Thus the sentence 'The cat eats fish' is parsed as

SENTENCE

NOUNPHRASE		VERB	NOUNPHRASE	
NOUN	ADJECTIVE	MANGE	NOUN	ADJECTIVE

CHAT MASC [LE LA LES LES] ARTICLE POISSON MASC

[LE LA LES LES] ART

Note that lists are enclosed in square brackets.

When applying TRANSLATE to this structure, first we apply TRANSLATE to the nounphrase (*x* TL HD), then we find the French equivalent of the verb, and print it preceded by a space. To translate the nounphrase, we first extract the gender of the noun (GENDER (*x* TL HD)). We then apply the function TRANSSELECT to the list of adjectives following the noun.

TRANSSELECT is defined by:

FUNCTION TRANSSELECT X G S

DEFINE

NEXT: IF X ATOM THEN EXIT

IF POSSELECT (X HD)=S THEN TRANSLATE (X HD, G) ALSO

X TL → X; GOTO NEXT END

Thus transelect translates only those items in the list *x* which have the atom *s* in the position *POSSELECT*. Hence the first call translates the articles 'the' and 'a', and the second those adjectives that come before the noun in French. Next, we translate the noun itself, and then those adjectives which come after the noun.

For curiosity's sake, Fig. 1 shows some worked examples done on the console typewriter as a demonstration.

MACROS

Since POP-1 is a list-processing language, one might expect that it would be easy to provide facilities for writing macros. This is indeed the case. First we define the function *MACRO* by

```
FUNCTION MACRO      VARS X
DEFINE READ→X;
  SETTYPE(X,5,3,15,0);
  <<"FUNCTION" X>> <> INSTREAM→INSTREAM
END
SETTYPE(MACRO,5,3,15,0);
```

This is in fact itself a macro definition, and works as follows. The final *SETTYPE* command has the effect of making the function *MACRO* be executed as soon as it is read. Thus if

```
MACRO THIRD . . .
```

occurs in the input instream, then when *MACRO* is read, it is immediately obeyed. Thus the first action of *MACRO* is to read the next word, i.e., *THIRD* off the input stream and store it in *x*. *INSTREAM* is a list which, if non-empty, contains the atoms which are next to be read off the input stream. Thus after the evaluation of *MACRO* the next items to be read off the input stream will be

```
FUNCTION THIRD . . .
```

Thus the effect of *MACRO . . .* is to define *THIRD* as a function which is itself evaluated as soon as it is read. Now let us consider how to define *THIRD* as a macro which will take the third member of a list.

```
MACRO THIRD
VARS X
DEFINE MACARGS→X;
  <<"("X>> <>[TL TL HD]] <> INSTREAM→ INSTREAM END
```

Let us see that *THIRD* defined above will produce the correct reverse Polish input for the assembler.

Suppose *THIRD* is called by

```
THIRD (A <> THIRD(B))
```

as soon as *THIRD* is read (and its precedence has been set by *settype* so that it is not transferred behind the list of arguments, which would be its fate if it were an ordinary function) it is evaluated. The function *MACARGS*, which is

PROBLEM-ORIENTED LANGUAGES

23 + 54 + 72*96 - 298*34 =>

** -3143

VARS A B C%

[Semicolon Does not occur on the typewriter so use % instead]

19 -> A% 25 -> B%

A*B + 98*(75 + B) + 4*A =>

** 10351

FUNCTION REV X

DEFINE

IF X ATOM THEN NIL EXIT

REV(X TL) <> X HD END

REV([1 2 3 4 5 6]) =>

** 1

[No good so redefine REV]

FUNCTION REV X

DEFINE

IF X ATOM THEN NIL EXIT

REV(X TL) <> <<X HD >> END

REV([1 2 3 4 5 6]) =>

** [6 5 4 3 2 1]

[OK. Now load translation program from paper tape]

[THE BIG BLACK CAT EATS THE RED MEAT] SENTENCE

**PLEASE TELL ME ABOUT "BIG"

**PART OF SPEECH

ADJECTIVE

**FRENCH

**SINGULAR

GRAND

**FEMININE

GRANDE

**PLURAL

GRANDS

**FEMININE

GRANDES

**

**IS IT BEFORE OR AFTER OR ARTICLE

BEFORE

**PLEASE TELL ME ABOUT "CAT"

**PART OF SPEECH

NOUN

**FRENCH

CHAT

**IS IT MASCULINE OR FEMININE

MASCLINE

**YOU MUST REPLY ONE OF MASCULINE OR FEMININE

MASCULINE

FIG. 1. An example of a POP-1 "session".

POPPLESTONE

```

**PLEASE TELL ME ABOUT "RED"
**PART OF SPEECH                                AJECTIVE
**I DO NOT KNOW THIS PART OF SPEECH
**PART OF SPEECH                                ADJECTIVE
**FRENCH
**SINGULAR                                      ROUGE
**FEMININE                                      ROUGE
**PLURAL                                       ROUE##ROUGES
**FEMININE                                      ROUGES
**
**IS IT BEFORE OR AFTER OR ARTICLE            AFTER

```

-> A -> B%

```

A =>
** 1 [This is the null list, and is the unparsed part of
the input]

```

```

B =>
** [SENTENCE [NOUNPHRASE [NOUN CHAT MASCULINE ]
**[ADJECTIVE [NOIR NOIRE NOIRS NOIRES ]AFTER ]
**[ADJECTIVE [GRAND GRANDE GRANDS GRANDES ]BEFORE ]
**[ADJECTIVE [LE LA LES LES ]ARTICLE ]][VERB MANGE
**][NOUNPHRASE [NOUN VIANDE FEMININE ]ADJECTIVE
**[ROUGE ROUGE ROUGES ROUGES ]AFTER ]ADJECTIVE
**[LE LA LES LES ]ARTICLE ]]]

```

[This is the tree structure of the sentence represented in bracketed form - Let us explore this tree]

```

3 HD =>
** SENTENCE

```

```

B TL HD =>
** [NOUNPHRASE [NOUN CHAT MASCULINE ]ADJECTIVE
**[NOIR NOIRE NOIRS NOIRES ]AFTER ]ADJECTIVE
**[GRAND GRANDE GRANDS GRANDES ]BEFORE ]ADJECTIVE
**[LE LA LES LES ]ARTICLE ]

```

```

B TL TL HD =>
** [VERB MANGE ]

```

```

B TL TL TL HD =>
** [NOUNPHRASE [NOUN VIANDE FEMININE ]ADJECTIVE
**[ROUGE ROUGE ROUGES ROUGES ]AFTER ]ADJECTIVE
**[LE LA LES LES ]ARTICLE ]

```

```

B TL TL TL TL HD
ERROR 30 COMPILER HD
ERROR 30
[An attempt has been made to evaluate HD with an atomic argument]

```

```

B 1 TRANSLATE
LE GRAND CHAT NOIR MANGE LA VIANDE RO
UGE

```

```

FUNCTION SEN##TRANS X
DEFINE X SENTENCE TRANSLATE END

```

```

[THE LITTLE B##BLACK BOY EATS THE MEAT] TRANS
LE PETIT GARCON NOIR MANGE LA VIANDE

```

[The character "##" causes the atom just typed to be ignored]

PROBLEM-ORIENTED LANGUAGES

a library function defined in POP-1, reads the string of atoms (A <> THIRD (B)) and converts it into the list [A <> THIRD(B)] (square brackets denote list structures, round brackets are just symbols). Assuming INSTREAM to be empty (i.e., we are not in the middle of a macro call), this first call of THIRD will leave

((A <> THIRD(B)) TL TL HD)

in INSTREAM. Processing this will result in A being compiled and <> being set aside before THIRD is called again. Thus at the second call of THIRD INSTREAM will contain (B) TL TL TL HD), and the second call will turn this into ((B TL TL HD)) TL TL HD). Hence B TL TL HD will now be compiled, and then the <> operator which was set aside, and then the TL TL HD, so the total effect of the macro calls is

A B TL TL HD <> TL TL HD

which is the correct reverse Polish form of the input stream.

IMPLEMENTATION

The present implementation (on an Elliott 4100) is by means of an interpreter. A program being input is first processed to set variables and functions into the correct reverse Polish order. The resulting stream is then translated into pseudo-machine code which is obeyed by the interpretive routine. Certain words such as FUNCTION, THEN, EXIT have the effect of activating special routines as soon as they are input. The user himself can define such words, e.g., in making macro definitions. Thus the whole system is easily extensible.

AIMS

POP-1 was originally conceived as a list-processing language which has acquired an increasingly on-line emphasis. It is now adopted as a language for exploratory programming projects at the Experimental Programming Unit of the University of Edinburgh, and as such will be incorporated in the on-line time-sharing system being developed there.

EDITOR'S NOTE

POP-1 has now been superseded by a greatly extended version, POP-2, which is the subject of Chapter 14.

REFERENCES

- Kaplaw, R., Strong, S., & Brackett, J. (1966). MAP: a system of on-line mathematical analysis. *Technical report MAC-TR-24*, Massachusetts Institute of Technology.
- Shaw, J. C. (1964). JOSS: A designer's view of an experimental on-line computing system. In *AFIPS Conference Proceedings*, Vol. 26. Baltimore and London: Spartan Books.

SELF-IMPROVEMENT IN QUERY LANGUAGES

J. M. FOSTER

MINISTRY OF AVIATION
ROYAL RADAR ESTABLISHMENT
GREAT MALVERN, WORCS.

INTRODUCTION

A program, by means of which untrained operators could question a computer on-line about a store of data, was demonstrated in an elementary form by the author (Foster 1967). A user can question the computer in a subset of English and the computer can ask questions back. The language which is to be used is specified to the program initially, by means of a set of syntactic rules and associated λ -calculus expressions. A feature of the program is its ability to accept definitions of new phrases and words in terms of existing ones. For example, the user can say

The average x of y , where x is a noun and y is a noun-phrase, is a noun-phrase and means 'the sum of the age of each of y divided by the number of y '.

The computer then generates a new syntactic rule and λ -expression in such a way that if a particular example is presented, say

the average age of the men

then it is exactly as if

the sum of the age of each of the men divided by the number of the men had been the phrase used.

The demonstrations, though indicating that the method was workable, left unanswered the question whether, on the scale of a practical example, the time and space used would be tolerable. A particular doubt concerns the

definition of new phrases by the users. If the definitions, each of which was implemented by adding a new syntactic rule to the system, were allowed to proliferate then the analysis stage of the program would almost certainly become too slow. The problem which this paper discusses is that of reorganising the syntactic definitions and their λ -expressions.

This is an example of a very general and difficult problem, that of learning to recognise data with structure. Instead of tackling this problem generally the paper presents some assumptions about structures and some rules for organising them which are probably valid in the field in question.

We wish to have a program which will be given pairs of strings of characters, which are assumed to have structures in some correspondence, representing the sentence and its translation into λ -expressions. The program will organise a description of these so that later, when it is presented with a sentence, it will associate a translation with it. The program should satisfy the following restrictions.

1. If presented with one of the original examples, it should associate with it the string which was given as its pair.
2. If it recognises any strings which are not the original strings, then the translation it associates with them should be sensible.
3. If it makes assumptions which turn out to be wrong it should be capable of reorganising the description to remove the fault.

For example, if it is presented with the following pairs

a	Pick, a
b	Pick, b
$a+b$	Pick, a ; Add, b
c	Pick, c

it might perhaps make the following translation,

$c+a$	Pick, c ; Add, a
$a+b+c$	Pick, a ; Add, b ; Add, c

The next section presents the assumptions that are made, the section following it gives some rules by means of which the data can be arranged and the penultimate section discusses the application of the rules to some examples of typical query language situations.

A program to apply the rules is currently partly working.

It is not suggested that the program is directly applicable in other fields, but some related pairs of structural strings in other problems are:

1. The set of operations carried out by the λ -expression evaluator and the results produced. Improvement here would enable often-applied groups of functions to be treated as special cases (e.g., whether a function should be computed or kept as a table).
2. Sentences and actions (indeed the general problem of learning a language).
3. Physical stimuli and reactions (e.g., balancing).
4. A picture and its description (in words or actions).

DISCUSSION OF ASSUMPTIONS

The program tries to form a syntactic description which generates translation strings from input strings in the following way. Suppose that one of the substitution rules of the grammar is

$$S \rightarrow A B C$$

where S is a class-name and A , B and C are either class-names or basic symbols (words). Each such rule has with it a string of characters

$$S \rightarrow A B C \quad pq3r21s1$$

in which the small letters represent constants and the numbers represent the outputs of the first, second, etc., parts of the rule. This string is the output of this rule. For example, the grammar

$$\begin{array}{ll} S \rightarrow P & \text{Pick, 1} \\ S + P & \text{1; Add, 3} \\ P \rightarrow a & a \\ & b \\ & c \end{array}$$

would perform the translations at the end of the introduction. We are trying, therefore, to produce a grammar like this from a set of example pairs. Since the query language program contains its translations in the form of λ -expressions, these are used for the rule meanings. However, this is a very simple use of λ -expressions.

A program which recognises similarities between structures and has some ability to generalise must operate against an opposing force lest it decide that everything is like everything else. In this case the opposing force is provided by the requirement that the translation string should correspond correctly with the example string. Since the process of translation is controlled by the syntax rules, this requirement considerably restricts the alterations which can be made to the syntax, in a way which seems both sensible and effective.

Accordingly, no transformations are made unless the following condition is satisfied. It amounts to ensuring that the translation process described is adequate. Suppose that a rule is

$$S \rightarrow A B C D \quad p12q43r2q43$$

and that parts of the rule are being considered to see whether they can be made into rules. Then it would be permissible to write

$$\begin{array}{ll} S \rightarrow A B X & p12q3r2q3 \\ X \rightarrow C D & 21 \end{array}$$

or

$$\begin{array}{ll} S \rightarrow A B X & p123r23 \\ X \rightarrow C D & q21 \end{array}$$

or

$$\begin{array}{ll} S \rightarrow A X & p12r2 \\ X \rightarrow B C D & 1q32 \end{array}$$

but the attempt to write, say,

$$\begin{aligned} S &\rightarrow A X D \\ X &\rightarrow B C \end{aligned}$$

would fail because translation strings could not be generated. Hence, if we are trying to make a new rule from a part of an old one, three conditions must be satisfied.

1. A unique piece of the translation string must correspond to the part; though this identical piece may occur several times.

2. Nothing corresponding to the part may occur outside the corresponding pieces of the translation string.

3. Since the translation strings are λ -expressions in the query language application, the new rule must be a complete expression.

This condition can be expressed by saying that if parts of sentences are grouped in the syntax then they will also be grouped in the translation. This assumption is a reasonable one for simple English sentences. No harm is done to the system by a counter-example, which will simply be learnt at a higher level in the syntax, that is, as an idiom.

The other assumption is that words and phrases used in similar contexts and with similar meanings (in the sense of the above constraint) can usefully be combined in a syntactic class. So

$$\begin{aligned} S &\rightarrow A B C & 3rp21 \\ &D E C & 3rq12 \end{aligned}$$

could give

$$\begin{aligned} S &\rightarrow X C & 2r1 \\ X &\rightarrow A B & p21 \\ &D E & q12 \end{aligned}$$

This example introduces no new possible input strings but we might say that

$$\begin{aligned} S &\rightarrow A B C & 123 \\ &D B E & 123 \end{aligned}$$

gives

$$\begin{aligned} S &\rightarrow X B Y & 123 \\ X &\rightarrow A & 1 \\ &D & 1 \\ Y &\rightarrow C & 1 \\ &E & 1 \end{aligned}$$

which introduces as possible examples of S , $A B E$ and $D B C$. Clearly this can introduce errors, but this will do no harm if an unlearning mechanism is present to change the rules back if necessary.

The next section gives the details of the algorithms which are being used to implement this assumption.

TRANSFORMATION ALGORITHMS

The examples of the section following may shed light on the reasons for these rules.

Learning Rule 1

Consider all the syntactic definitions in pairs. If there are pieces which are common to two definitions which satisfy the constraint of the last section, then they may be made into a new rule. This is done unless the piece consists of a single class name.

$$\begin{array}{ll} A \rightarrow B C D & 123 \\ E \rightarrow G B C F & 1423 \end{array}$$

gives

$$\begin{array}{ll} A \rightarrow X D & 12 \\ E \rightarrow G X F & 132 \\ X \rightarrow B C & 12 \end{array}$$

The application of this rule does not change the sentences that are recognised by the program, and hence there is no need to undo this process. It merely gives a more condensed description of the sentences.

Learning Rule 2

If

$$\begin{array}{ll} Q \rightarrow T & 1 \\ & A B C \\ T \rightarrow D E & \\ & F G \end{array}$$

where $A B C$, $D E$, etc., are any sets of symbols, then replace Q by T and add $A B C$ to the alternatives for I .

This rule covers the case when things are recognised to be similar in different places. For example, if

$$\begin{array}{ll} Q \rightarrow A B & \\ & C D \\ T \rightarrow A B & \\ & C D \end{array}$$

and the translations are identical, then the following sequence guesses that Q and T are identical.

$$\begin{array}{ll} Q \rightarrow X & \\ & Y \\ T \rightarrow X & \\ & Y \\ X \rightarrow A B & \\ Y \rightarrow C D & \end{array}$$

(by Learning Rule 1)

$$\begin{array}{ll} T \rightarrow X & \\ & Y \\ X \rightarrow A B & \\ & Y \\ Y \rightarrow C D & \\ & 199 \end{array}$$

PROBLEM-ORIENTED LANGUAGES

and Q is rewritten as X (LR2 on Q)

$$X \rightarrow A B$$

Y

$$Y \rightarrow C D$$

and T is rewritten as X (LR2 on T)

Finally

$$Y \rightarrow A B$$

$C D$

and X is rewritten as Y (LR2 on X)

This rule can introduce errors, so an inverse is needed for when they are detected.

Unlearning Rule 2

$$Q \rightarrow A C$$

$B D$

$$R \rightarrow \dots Q \dots$$

and

$$Q \leftrightarrow A C \text{ in } R$$

gives

$$Z \leftrightarrow A C$$

$$Q \rightarrow A C$$

$B D$

$$Z \rightarrow B D$$

$$R \rightarrow \dots Z \dots$$

where the significance of $\leftrightarrow$ is that this rule must not be generated by any transformations.

Learning Rule 3

For all the definitions of a single class-name, pick out the similar pieces satisfying the constraint and examine the remaining corresponding pieces. The largest corresponding pieces which satisfy the constraint are formed into a rule. For example, if

$$A \rightarrow B C D E F \quad 21543$$

$$H D J K \quad 1342$$

gives

$$A \rightarrow X D Y \quad 132$$

$$X \rightarrow B C \quad 21$$

$$H \quad 1$$

$$Y \rightarrow E F \quad 21$$

$$J K \quad 12$$

This rule again can introduce errors, since it implies that

$$A \rightarrow H D E F \quad 1432$$

$$B C D J K \quad 21453$$

which might or might not be the case. We therefore need an unlearning rule.

Unlearning Rule 3

$$P \rightarrow X C Y$$

$$X \rightarrow A B$$

$$R S$$

$$Y \rightarrow D E$$

$$T U$$

and

$$P \leftrightarrow A B C T U$$

gives

$$P \rightarrow A B C Z$$

$$D E C Y$$

$$Y \rightarrow D E$$

$$T U$$

$$Z \rightarrow D E$$

$$Z \rightarrow T U$$

All these rules are based on the assumption of section 2 and satisfy the constraint of that section.

EXAMPLES

1. We consider an example showing how the word average, used in a context like the word sum, is assumed to operate in a different context by analogy with the properties of 'sum'.

Suppose that part of a grammar is

$np \rightarrow np$ divided by np	$\text{div}(1, 4)$
np of np of all np	$1(3, 6)$
the noun	2
the noun of np	$2(4)$
np of np and np	$1(\text{value}, \text{set}(3, 5))$
$\text{noun} \rightarrow \text{sum}$	sigma
number	count
ages	age
men	men

The translation of 'the sum of the ages of the men'

is

sigma (age, men)

where the definition of sigma is the sum of the value of the first parameter applied to each member of the class which is the second parameter.

The translation of 'the sum of a and b '

is

sigma (value, set(a , b))

where set makes its parameters into a single class and value gives the numerical value of a class member.

If in this grammar we define

'the average of x of all y ' where x and y are np , is a np and means 'the sum of x of all y divided by the number of y '
then we obtain (1) a new rule for np

$$np \rightarrow \text{the average of } np \text{ of all } np \\ (\lambda xy. \text{div} (\text{sigma} (x, y), \text{count}(y)))(3, 6)$$

LR3 applied to this definition and the second of np gives

$$\begin{array}{ll} np \rightarrow X \text{ of } np \text{ of all } np & 1(3, 6) \\ X \rightarrow np & 1 \\ \text{the average} & (\lambda xy. \text{div} (\text{sigma} (x, y), \text{count} (y))) \end{array}$$

LR2 then gives

$$\begin{array}{ll} np \rightarrow np \text{ of } np \text{ of all } np & 1(3, 6) \\ \text{the average} & (\lambda xy. \text{div} (\text{sigma} (x, y), \text{count}(y))) \end{array}$$

LR3 applied to the latter rule and the third of np gives

$$\begin{array}{ll} np \rightarrow \text{the } Y & 2 \\ Y \rightarrow \text{noun} & 1 \\ \text{average} & (\lambda xy. \text{div} (\text{sigma} (x, y), \text{count}(y))) \end{array}$$

and LR2 then gives

$$\begin{array}{ll} np \rightarrow \text{the noun} & 2 \\ \text{noun} \rightarrow \text{average} & (\lambda xy. \text{div} (\text{sigma} (x, y), \text{count}(y))) \end{array}$$

If to this new grammar the phrase 'the average of a and b ' is presented, it translates it as

$$(\lambda xy. \text{div} (\text{sigma} (x, y), \text{count} (y))) (\text{value}, \text{set} (a, b))$$

which happens to be correct.

2. If the definition in the last example had been

'the average of the ages of all the men' is a np and means 'the sum of the ages of all the men divided by the number of the men'

then the final result would have been the same, since 'the ages' and 'the men' would have been taken to be np by application of LR1.

3. If the program can recognise word suffixes like $-s$ or $-ed$ then it will assume that if two words are used similarly and one forms a plural with $-s$ then so does the other. It can then unlearn this if it finds itself wrong by using UR3.

4. This sequence of arithmetic expressions will produce the following grammar

a	Pick, a
b	Pick, b
c	Pick, c
$a+b$	Pick, a ; Add, b
$c+b$	Pick, c ; Add, b
$a+c$	Pick, a ; Add, c
	202

$b + c$	Pick, b ; Add, c
$a + b + c$	Pick, a ; Add, b ; Add, c
$S \rightarrow Y X$	Pick, 12
$Y \rightarrow a$	a
b	b
c	c
$X \rightarrow \phi$	
$+ Y Z$	; Add, 23
$Z \rightarrow \phi$	
$+ Y$	; Add, 2

where ϕ is the empty rule.

Note that this grammar only recognises three term sums. That is, the similarity between a , b and c has been noticed, but not the possibility of producing the grammar of the second section.

CONCLUSIONS

A program which uses the correspondence between an input string and its translation is being written to recognise some structural properties of the strings. A certain amount of generalisation (and corresponding unlearning) have been built in. The rules used can be justified only in so far as they work.

REFERENCE

- Foster, J. M. (1967). Interrogation Languages, *Machine Intelligence 1*, 267-276, ed. Collins & Michie. Edinburgh: Oliver and Boyd.

POP-2 REFERENCE MANUAL

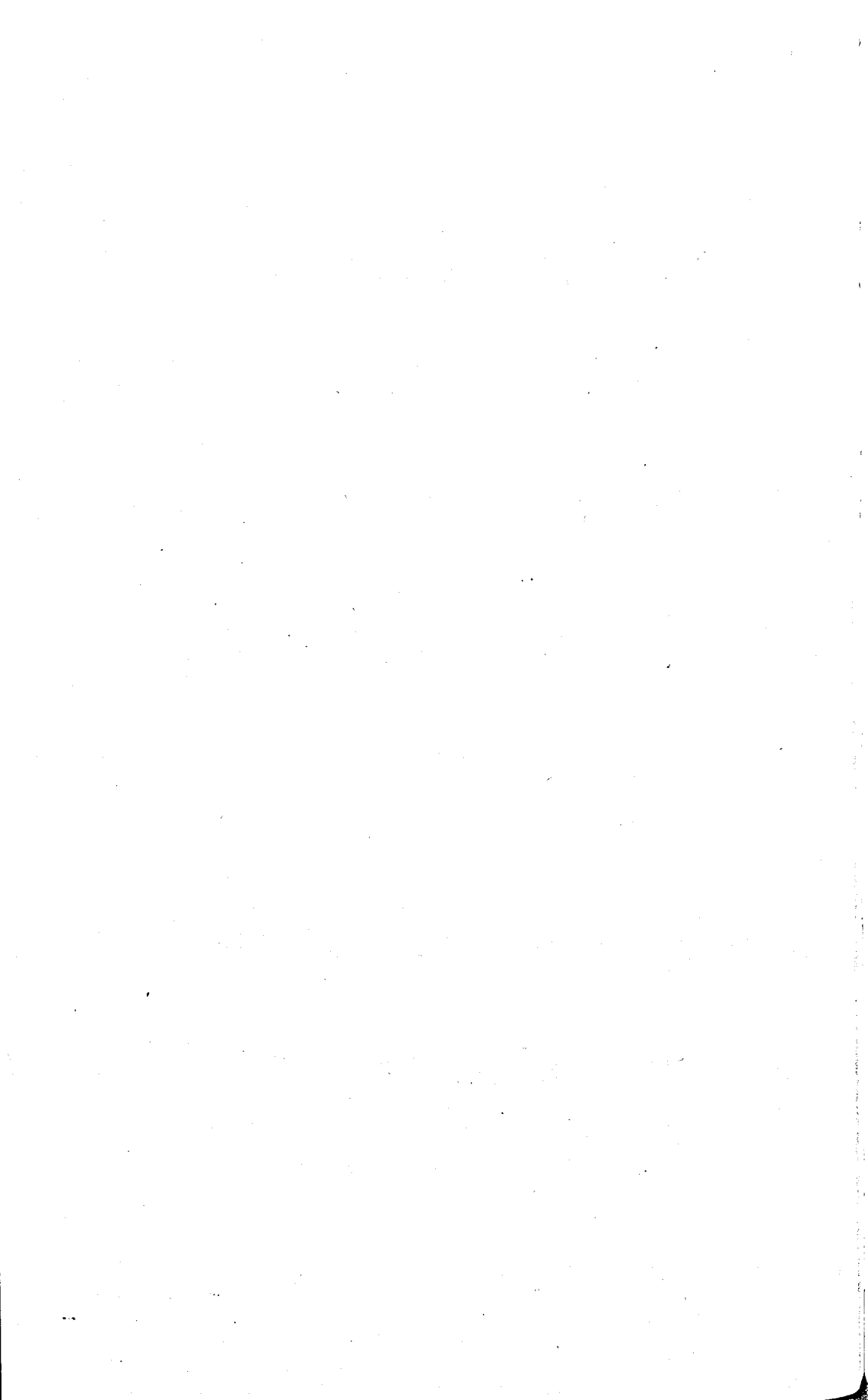
BY

R. M. BURSTALL

AND

R. J. POPPLESTONE

**DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION
UNIVERSITY OF EDINBURGH**



CONTENTS

	PAGE
1 INTRODUCTION	
1.1 Aims	209
1.2 Main features	209
1.3 Examples	210
1.4 Notation for syntactic description	212
1.5 Notation for functions	213
2 ITEMS	
2.1 Simple and compound items	214
2.2 Integers	215
2.3 Reals	215
2.4 Truth values	216
2.5 Undefined	216
2.6 Terminator	216
3 VARIABLES	
3.1 Identifiers	216
3.2 Declaration and initialisation	217
3.3 Cancellation	219
4 FUNCTIONS	
4.1 Definition of functions	219
4.2 Application of functions	220
4.3 Nonlocal variables	221
4.4 Partial application	221
4.5 Doublets	223
4.6 Arithmetic operations	224
5 EXPRESSIONS AND STATEMENTS	
5.1 Expressions	224
5.2 Precedence	226
5.3 Statements and imperatives	226
5.4 Labels and goto statements	227

CONTENTS

	PAGE
5.5 Assignment	228
5.6 Comments	229
6 CONDITIONALS	
6.1 Conditional expressions	229
6.2 Conjunctions and disjunctions	230
7 DATA STRUCTURES	
7.1 Functions of data structures	231
7.2 Records	233
7.3 Strips	234
7.4 Garbage collection	235
8 STANDARD STRUCTURES	
8.1 References	236
8.2 Pairs	236
8.3 Lists	236
8.4 Full strips and character strips	239
8.5 Arrays	239
8.6 Words	240
8.7 Functions	241
9 INPUT AND OUTPUT	
9.1 Input	242
9.2 Output	243
10 MACHINE CODE	244
11 MODES OF EVALUATION	
11.1 Immediate evaluation	244
11.2 Macros	244
11.3 Evaluation of program text	245
Acknowledgments	245

1. INTRODUCTION

1.1. Aims

The following are the main design objectives for the POP-2 language:

(i) The language should allow convenient manipulation of a variety of data structures and give powerful facilities for defining new functions over them.

(ii) The language should be suitable for taking advantage of on-line use at a console, i.e. it should allow immediate execution of statements and should have a sufficiently simple syntax to avoid frequent typing errors.

(iii) A compiler and operating system should be easy to write and should not occupy much storage.

(iv) The elementary features of the language should be easy to learn and use.

(v) The language should be sufficiently self-consistent and economical in structure to allow it to incorporate new facilities when extensions are desired.

In attaining these objectives certain other desirable features of programming languages had to be relegated to secondary importance:

(vi) Fast arithmetical facilities on integer and real numbers.

(vii) Fast subscripting of arrays.

(viii) A wide variety of elegant syntactic forms.

Naturally whether (iii) or (vi) and (vii) are attained is to a considerable extent a matter of implementation.

1.2. Main features

The following main features are provided. Roughly analogous features of some other programming languages are mentioned in brackets as a guide:

(i) Variables (cf. ALGOL but no types at compile time).

(ii) Constants (cf. ALGOL numeric and string constants, LISP atoms and list constants).

PROBLEM-ORIENTED LANGUAGES

- (iii) Expressions and statements (cf. ALGOL).
- (iv) Assignment (cf. ALGOL, also CPL left-hand functions).
- (v) Conditionals, jumps and labels (cf. ALGOL but restrictions on jumps and labels).
- (vi) Functions (cf. ALGOL procedures but no call by name, cf. CPL and ISWIM for full manipulation of functions).
- (vii) Arrays (cf. ALGOL; cf. CPL for full manipulation of arrays).
- (viii) Records (cf. COBOL, PL/1, Wirth-Hoare ALGOL records, CPL nodes).
- (ix) Words (cf. LISP atoms).
- (x) Lists (cf. LISP, IPL-V).
- (xi) Macros.
- (xii) Use of compiler during running (cf. LISP, TRAC, FORMULA ALGOL).
- (xiii) Immediate execution (cf. JOSS, TRAC).

Notes:

LISP: LISP 1.5

CPL: See Barron, D. W., *et al.* 1964. The main features of CPL, *Computer J.*, 6, 134-43.

CPL reference manual. Edited C. Strachey (privately circulated).

Wirth-Hoare ALGOL: See Wirth, N., and Hoare, C. A. R. 1966. A contribution to the development of Algol, *Communs Assn Comput. Mach.*, 9, 413-32.

TRAC: See Mooers, C. N. 1966. TRAC, a procedure describing language for the reactive typewriter, *Communs Assn Comput. Mach.*, 9, 215-24.

ISWIM: See Landin, P. J. 1966. The next 700 programming languages, *Communs Assn Comput. Mach.*, 9, 157-166.

1.3. Examples

The following is an example of POP-2 program text. The sign $\Rightarrow$ (not to be confused with that used in section 1.5 'Notation for functions') prints out some results on a newline prefixed with two asterisks. These results are included in the text below, as they would appear if the program were run on-line at a console.

```
comment arithmetic;
12.0+2.5*(1.5+2.5) $\Rightarrow$ 
**22.0
vars a b sum;
2*2 $\rightarrow$ a; 3*a $\rightarrow$ b; a*a+b*b $\rightarrow$ sum; sum  $\Rightarrow$ 
**160
function sumsq x y;
  x*x+y*y
end;
sumsq(a, b)+1 $\Rightarrow$ 
** 161
```

```

function fact n; vars p;
  1→p;
loop: if n=0 then p else n*p→p; n-1→n; goto loop close
end;
fact(fact(3))⇒
** 720

```

```

comment arrays;
vars a i j;
10→i; 20→j;
newarray([%1, i, 1, j%], sumsq)→a;
a(2, 3)⇒
** 13
10→a(2, 3); a(2, 3)⇒
** 10
function arraysum a1 a2 m n;
  newarray ([%1, m, 1, n%], lambda i j; a1(i, j)+a2(i, j) end)
end;
arraysum (a, a, 10, 20)→a; a(2, 3)⇒
** 20

```

```

comment lists;
vars u;
1→i; 2→j;
[%i, i+j, "dog", "cat" %]→u; u⇒
** [1 3 dog cat]
cons ("pig", u)⇒
** [pig 1 3 dog cat]
function append x y;
  if null (y) then [% x %] else cons(hd(y), append(x, tl(y))) close
end;
append(4, [% 1, i+1, 3%])⇒
** [1 2 3 4]

```

```

comment records;
vars consper destper forename surname male p1 p2;
recordfns("person", 500, [0 0 1])→male→surname→forename
  →destper→consper;
consper("jane", "jones", false)→p1; consper("sam", "smith", true)→p2;
surname(p1)⇒
** jones
datalist(p1)⇒
** [jane jones 0]
routine marry x y;
  if male(x) and not(male(y)) then surname(x)→surname(y) close

```

```
end;
marry(p2, p1); datalist(p1)⇒
** [jane smith 0]
```

1.4. Notation for syntactic description

We use the BNF (Backus-Naur Form) notation as used in the ALGOL report:

::= indicates a syntax definition;
 < > are used to enclose the name of a syntax class;
 | denotes disjunction (union of syntax classes).

Concatenation denotes concatenation of any elements of two syntax classes.

We also use a convenient extension of this notation due to R. A. Brooker:

* means that a class may occur n times, $n \geq 1$;
 ? means that a class may occur n times, $n = 0$ or 1 ;
 *? means that a class may occur n times, $n \geq 0$,

e.g. the definitions

```
<string> ::= <a> <string> | <a>
<bstring> ::= <b> <string>
<cstring> ::= <c> <string> | <c>
```

may be replaced by

```
<bstring> ::= <b> <a*>
<cstring> ::= <c> <a*?>
```

The characters <, > and * are used in the POP-2 reference language but no confusion should arise.

When we wish to give examples of a syntax class we use the symbol 'e.g. ::=', for example:

```
<bstring> e.g. ::= <b> <a> | <b> <a> <a> <a>
```

The character set of the POP-2 reference language is as follows.

```
<letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<sign> ::= +|-|*|/|$|&|=|<|>|:|£|↑
<separator> ::= =,|;
<period> ::= .
<sub ten> ::= 10
<bracket> ::= (|)|[|]
<bracket decorator> ::= %
<quote> ::= "
<string quote> ::= ^|`
```

Letters may be written in lower case, upper case or **heavy type** without any change of meaning. It will be conventional however to use **heavy type** letters for syntax words, i.e. those identifiers such as **function**, **then**, **end** and **cancel** which have a special meaning for the POP-2 compiler and which characterise certain syntactic forms.

Spaces, tabulate and new lines terminate identifiers, integers, reals and words but otherwise they are ignored.

A distinction is made between the reference language used in this document and a number of possible hardware languages used by particular computer implementations of POP-2. Each character in the reference language should be represented by a distinct character or sequence of characters in the hardware language. A particular letter, whether upper case, lower case, heavy type or not is regarded as the same character in the reference language.

The symbols $\rightarrow$ and $\Rightarrow$ used in this paper should be read as a typographical abbreviation for the pairs of characters $->$ (minus greater than) and $=>$ (equals greater than) respectively.

1.5. Notation for functions

It is convenient to have a notation to specify the domain and range of functions. We will consider functions having several arguments (or possibly none) and producing several results (or possibly none), the notion of functions with more than one result being an extension of normal mathematical usage (see section 4.2 'Application of functions'). We introduce a special symbol ' $\Rightarrow$ ' which is not to be confused with any identifier in the POP-2 language.

Suppose $d1, d2, \dots, dm$ and $r1, r2, \dots, rn$ are all sets of items. Then $d1, d2, \dots, dm \Rightarrow r1, r2, \dots, rn$ is the set of all functions whose domain is $d1, d2, \dots, dm$ and range $r1, r2, \dots, rn$, i.e. with arguments which are m -tuples in $d1 \times d2 \times \dots \times dm$ and with results which are n -tuples in $r1 \times r2 \times \dots \times rn$. We express the fact that a function f is a member of this set of functions by

$$f \in d1, d2, \dots, dm \Rightarrow r1, r2, \dots, rn$$

Some examples will make this clear.

$$\text{add} \in \text{integer}, \text{integer} \Rightarrow \text{integer}$$

$$\text{divrem} \in \text{integer}, \text{integer} \Rightarrow \text{integer}, \text{integer}$$

where *divrem* is 'divide with remainder', e.g. *divrem* (7, 3) = 2, 1 and *divrem* (14, 4) = 3, 2

$$\text{roundup} \in \text{real} \Rightarrow \text{integer}$$

$$\text{prime} \in \text{integer} \Rightarrow \text{truthvalue}$$

If the function has no results we use an empty pair of parentheses, thus:

$$\text{printout} \in \text{integer} \Rightarrow ()$$

The arguments or results may themselves be functions

$$\text{differentiate} \in (\text{real} \Rightarrow \text{real}) \Rightarrow (\text{real} \Rightarrow \text{real})$$

PROBLEM-ORIENTED LANGUAGES

Where we wish to discuss a number of functions all having the same domain and range it is convenient to abbreviate thus:

$f, g, \dots h \text{ all } \in \dots \Rightarrow \dots$
for
 $f \in \dots \Rightarrow \dots$
and
 $g \in \dots \Rightarrow \dots$
.....
and
 $h \in \dots \Rightarrow \dots$

Some functions do not have a fixed number of arguments and some do not have a fixed number of results (see section 4.2 'Application of functions'). In such cases we may write for example

$f \in \text{integer} \Rightarrow \text{real}, \text{integer}, \dots, \text{integer}$

for the domain or range, meaning that a real and a variable number of integers are the results.

2. ITEMS

2.1. Simple and compound items

The objects on which one can operate are called *Items*. They are divided into two distinct classes: *Compound* items, which are represented by addresses and *Simple* items which are directly represented by bitstrings which do not contain addresses (these bit strings are normally of fixed length for a given implementation, being a single machine word). The address representing a compound item points to a bit string whose length may vary from item to item. This bit string may contain other items. The areas of store immediately pointed to by two different compound items do not overlap.

The following standard function recognises compound items:

$\text{iscompnd} \in \text{item} \Rightarrow \text{truthvalue}$

Two kinds of simple item are distinguished: integers and reals. The following standard functions recognise them:

$\text{isinteger}, \text{isreal} \text{ all } \in \text{item} \Rightarrow \text{truthvalue}$

The standard function $=$ (an operation of precedence 7) is used to represent equality of items. For integers and reals it has the usual meaning. Its meaning for compound items is given in section 7.1 'Functions of data structures'.

$= \in \text{item}, \text{item} \Rightarrow \text{truthvalue}$

2.2. Integers

Integers are simple items. They may be positive, negative or zero. The size of the largest and smallest integers allowed depends on the implementation. The following functions on integers are standard:

intadd, *intsub*, *intmult*, *all* $\in$ integer, integer $\Rightarrow$ integer
 $\quad \quad \quad //$ $\in$ integer, integer $\Rightarrow$ integer, integer
intplus, *intminus* *all* $\in$ integer $\Rightarrow$ integer.
intsign $\in$ integer $\Rightarrow$ integer
intgr, *intle*, *intgreq*, *intleeq* *all* $\in$ integer, integer $\Rightarrow$ truthvalue

intadd, *intsub*, *intmult* and *intdiv* are the usual add, subtract and multiply. $//$ is divide with remainder and produces a quotient and a remainder (if $a//b$ is (q, r) , then $q*b+r=a$ and $0 \leq r < b$). It is an operation of precedence 4.

intplus carries an integer into itself and *intminus* complements an integer. *intsign* produces -1 , 0 , or $+1$ according to the sign of the integer. The remaining four functions are the relations 'greater than', 'less than', 'greater than or equal to' and 'less than or equal to'.

The syntax of integers is:

$\langle \text{integer} \rangle ::= \langle \text{octal integer} \rangle | \langle \text{binary integer} \rangle | \langle \text{decimal integer} \rangle$
 $\langle \text{octal integer} \rangle ::= 8 : \langle \text{octal digit}^* \rangle$
 $\langle \text{binary integer} \rangle ::= 2 : \langle \text{binary digit}^* \rangle$
 $\langle \text{decimal integer} \rangle ::= \langle \text{digit}^* \rangle$
 $\langle \text{octal digit} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7$
 $\langle \text{binary digit} \rangle ::= 0 | 1$

Example:

$\langle \text{integer} \rangle \text{e.g.} ::= 8:777|2:101|0|6559$

Integers may also be treated as bit-strings (the length depending on the implementation) and the following functions are standard:

logand, *logor*, *logshift* *all* $\in$ integer, integer $\Rightarrow$ integer
lognot $\in$ integer $\Rightarrow$ integer

logand and *logor* are the usual bit by bit 'and' and 'inclusive or'; *logshift* causes the first integer to be shifted left by the number of places given in the second, unless the second integer is negative when shifting to the right takes place (all new bits to fill up the end are zero in each case).

2.3. Reals

Reals are simple items. They may be positive, negative or zero. The size of the largest and smallest reals allowed and the precision depends on the implementation. The following functions on reals are standard:

realadd, *realsub*, *realmult*, *realdiv* *all* $\in$ real, real $\Rightarrow$ real
realplus, *realminus* *all* $\in$ real $\Rightarrow$ real
realsign $\in$ real $\Rightarrow$ integer
realgr, *realle*, *realgreq*, *realleq* *all* $\in$ real, real $\Rightarrow$ truthvalue

PROBLEM-ORIENTED LANGUAGES

These are the usual add, subtract, multiply and divide on reals. *realplus* carries a real into itself and *realminus* complements a real. *realsign* produces -1, 0 or +1 according to the sign of the real. The remaining four functions are the relations 'greater than', 'less than', 'greater than or equal to' and 'less than or equal to'.

There are also operations to convert a real to the nearest integer and to convert an integer to real:

$intof \in real \Rightarrow integer$
 $realof \in integer \Rightarrow real$

The syntax of reals is as follows:

$\langle real \rangle ::= \langle decimal\ integer\ ? \rangle . \langle decimal\ integer \rangle \langle exponent\ ? \rangle$
 $\langle exponent \rangle ::= {}_{10} + \langle integer \rangle | {}_{10} - \langle integer \rangle | {}_{10} \langle integer \rangle$

Example:

$\langle real \rangle$ e.g. $::= .5 | 1.99 | 1.5_{10} - 6$

2.4. Truth values

The two items *True* which is the integer 1 and *False* which is the integer 0 are called *Truthvalues*.

On entry to the POP-2 system the standard variable *true* is set to 1 and the standard variable *false* is set to 0. The following standard functions on truthvalues are provided:

$booland, boolor\ all \in truthvalue, truthvalue \Rightarrow truthvalue$
 $not \in truthvalue \Rightarrow truthvalue$

These are the usual functions 'and', 'inclusive or' and 'not' of propositional calculus.

2.5. Undefined

The standard variable *undef* has the word "*undef*" as its value on entry to the POP-2 system (see section 8.6 'Words'). The programmer may use it as the result of a function which fails to produce its normal result.

2.6. Terminator

The standard variable *termin* has the word "*termin*" as its value on entry to the POP-2 system (see section 8.6 'Words'). It may be used as the first argument of a variadic function (see section 4.2 'Application of functions') or to mark the end of an input file (see section 9.1 'Input').

3. VARIABLES

3.1. Identifiers

An item may be the *Value* of a *Variable* (a variable is not itself an item). An *Identifier* is associated with the variable and this identifier is used to

refer to it in a POP-2 program. A number of distinct variables may have the same identifier, but only one of them is *Currently associated* with it at a particular time in the evaluation process.

An identifier may be restricted to a certain range of values and it may be given special syntactic properties by being given a precedence (see section 5.2 'Precedence').

The syntax of identifiers is:

$\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \langle \text{alphanumeric} * ? \rangle | \langle \text{sign} * \rangle$
 $\langle \text{alphanumeric} \rangle ::= \langle \text{letter} \rangle | \langle \text{digit} \rangle$

Example:

$\langle \text{identifier} \rangle \text{e.g.} ::= x | y99 | \alpha | u2a | +++ | /+ | \langle \rangle | * \$ \$ *$

Syntax words such as **then**, **end**, **→** and **:** have special meanings and may not be used as identifiers. Only the first 8 characters are significant.

3.2. Declaration and initialisation

A variable is either *Global*, *Local* or *Formal*. A *Declaration* is used to introduce an identifier and associate it with a global or local variable. A *Local Declaration*, introducing a local variable, is a declaration which occurs in a function body. A *Global Declaration*, introducing a global variable, is one which does not.

An *Initialisation* is used to introduce an identifier and associate it with a formal variable and give the variable an initial value. It is achieved by including the identifier in the formal parameter list of a function (see section 4.1 'Definition of functions').

A declaration or initialisation may also specify that the identifier is restricted to take only functions as values. This is not necessary but may make the implementation more efficient. A declaration or initialisation may also specify that the identifier is an *Operation*, i.e. it is restricted to take functions as its values and is given a precedence. This restriction is associated with the unique name (see below) produced by the declaration or the initialisation.

The syntax of declarations is:

$\langle \text{declaration} \rangle ::= \text{vars} \langle \text{declaration list element} * \rangle$
 $\langle \text{declaration list element} \rangle ::= \langle \text{identifier} \rangle | \langle \text{restriction} \rangle$
 $\langle \text{restriction} \rangle ::= \langle \text{restrictor} \rangle \langle \text{identifier} \rangle | \langle \text{restrictor} \rangle (\langle \text{identifier} * \rangle)$
 $\langle \text{restrictor} \rangle ::= \text{function} | \text{operation} \langle \text{integer} \rangle$

Example:

$\langle \text{declaration} \rangle \text{e.g.} ::= \text{vars } x \ y | \text{vars } x \ y \text{ function}(f \ g) \text{ operation } 7 ==$

A declaration or initialisation has a *Scope*, which is a piece of POP-2 text. An identifier may not be used to represent a variable outside the scope of a declaration or initialisation of the identifier.

The scope of a global declaration starts at the declaration and continues until the identifier is cancelled.

The scope of a local declaration starts at the declaration and continues to the end of the innermost function body enclosing it.

The scope of an initialisation is the body of the function in which it occurs.

Each declaration or initialisation gives rise to a unique mark and this mark is associated with all occurrences of any identifier introduced by the declaration or initialisation within the scope of the declaration or initialisation. An identifier together with its unique mark is called a *Unique name*.

Thus an identifier which occurs in more than one declaration or initialisation corresponds to more than one unique name.

The generation of fresh unique names for identifiers can be suppressed by using the standard routines:

nonunique, *unique* $all \in \emptyset \Rightarrow \emptyset$

If *nonunique* is applied, all declarations or initialisations of a given identifier until *unique* is applied will give rise to the same unique name. This may save storage space and can be used when no confusion is liable to occur.

To sum up:

A new identifier is introduced by introducing a fresh sequence of characters.

A new unique name is introduced by each declaration or initialisation (unless *nonunique* has been applied).

A new variable is introduced by each dynamic activation of a declaration or initialisation.

A variable has an *Extent* which is a sequence of evaluations of expressions and statements.

The extent of a global variable starts from its declaration and continues indefinitely.

The extent of a local or formal variable starts on entry to the body of the function in which it is declared or initialised and continues until exit from the body. During this extent the extent of any other variable with the same unique name is temporarily interrupted. This is called a *Hole in the Extent* of the other variable. Its value is not altered but it cannot be accessed or changed by assignment. Thus there is only one variable *Currently Associated* with a particular unique name during any evaluation. Other variables associated with the unique name are in abeyance.

More than one global declaration of the same identifier is not permitted unless a cancellation of it intervenes in the text.

Similarly a declaration of a local variable is not permitted if there is already a declaration of a local or initialisation of a formal with the same identifier for the same function body.

A *Standard Variable* is a global variable which already has a value on entry to the POP-2 system. A *Standard Function* (or *Routine*) is one which is the value of a standard variable. Certain standard variables are *Protected*, i.e. no assignment may be made to them.

3.3. Cancellation

A cancellation terminates the scope of any declaration of an identifier and removes the effect of any restrictions placed upon the identifier. The cancellation must occur textually between the old declaration and any new declaration. It may not occur in a function body.

The syntax of cancellations is:

$\langle \text{cancellation} \rangle ::= \text{cancel } \langle \text{identifier} * \rangle$

4. FUNCTIONS

4.1. Definition of functions

A *Function* is a compound item. Definition and application of functions are treated in this section and the next. Certain properties of a function regarded as a data structure are treated in section 8.7 'Functions'.

A function consists of a *Formal Parameter List* which is a list of identifiers of formal variables, possibly an *output local list* which is a list of the identifiers of output local variables (see section 4.2 'Application of functions') and a *Body* which is an imperative sequence (see section 5.3 'Statements and imperatives').

A function which produces no results (see section 4.2 'Application of functions') is called a *Routine*.

Functions may be referred to in the program by using a function constant, called a *Lambda Expression*, or they may be standard functions provided by the POP-2 system, or they may be created by partial application or by application of a standard function which produces a function as a result.

The syntactic representation of a function constant is:

$\langle \text{formal parameter list element} \rangle ::= \langle \text{ident} \rangle \mid \langle \text{restriction} \rangle$
 $\langle \text{formal parameter list} \rangle ::= \langle \text{formal parameter list element} * ? \rangle$
 $\langle \text{output local list element} \rangle ::= \langle \text{ident} \rangle \mid \langle \text{restriction} \rangle$
 $\langle \text{output local list} \rangle ::= \Rightarrow \langle \text{output local list element} * ? \rangle$
 $\langle \text{function body} \rangle ::= \langle \text{imperative sequence} \rangle$
 $\langle \text{lambda expression} \rangle ::= \text{lambda} \langle \text{formal parameter list} \rangle \langle \text{output local list} \rangle ;$
 $\langle \text{function body} \rangle \text{ end}$

Example:

$\langle \text{lambda expression} \rangle \text{e.g.} ::= \text{lambda } x \ y; \text{ cons}(x, \text{ cons}(a, y)) \text{ end}$
 $\mid \text{lambda } x; \text{ nl}(1); \text{ print}(x) \text{ end}$

We very often wish to declare a variable and then assign a function to it. The syntactic form of this will be as follows:

$\text{vars } \langle \text{identifier} \rangle ;$
 $\langle \text{lambda} \rangle \langle \text{formal parameter list} \rangle ; \langle \text{function body} \rangle \text{ end} \rightarrow \langle \text{identifier} \rangle$

This is so common that a special syntactic form is introduced which is equivalent to it:

```

<function>::=function | routine
<function definition>::=<function><identifier><formal parameter list>;
<function body> end
    
```

The word **routine** is a synonym for **function**. It may be used for a function with no results.

If the identifier has been previously declared at this level no new declaration is implied and the function definition is equivalent simply to an assignment of a lambda expression. The identifier may be an operation identifier.

Example:

```

<function definition>e.g.::=function max x y; if x > y then x else y close
end
| routine enter u v; cons(conspair(u, v), dict)→
dict end
| function order x y ⇒ u v;
if x > y then x → u; y → v
else y → u; x → v close
end
    
```

4.2. Application of functions

An *n*-*Tuple* is an ordered sequence of *n* items ($n \geq 0$). An item is identical with the 1-tuple whose sole member is that item. An *n*-tuple and an *m*-tuple may be *Concatenated* to produce an $(n+m)$ -tuple.

A function of *n* arguments (i.e. with *n* formal parameters, excluding frozen formals; see section 4.4 'Partial application'), may be *Applied* to an *n*-tuple, whose members are called the *Actual Parameters* of the function. Application of a function to its actual parameters produces an *m*-tuple, whose members are said to be the *Results* of the function. A function producing no results (i.e. an 0-tuple) is called a *routine* (see section 4.1 'Definition of functions').

A function which does not take a fixed number of arguments is called *Variadic*. A function which does not produce a fixed number of results is called *Varireresult*.

The application of a function to its actual parameters consists of the following sequence of events:

Entry: a new variable corresponding to each formal parameter is initialised to the corresponding actual parameter value, or if it is a frozen formal to the corresponding value in the frozen value list. A new variable corresponding to each local variable declaration in the function body but not in any interior function body is then created. The variables previously associated with the identifiers of formal or local variables can no longer be referred to but their values are undisturbed.

Running: the function body is evaluated with the variables created on entry.

Exit. Any items which have been placed on the stack (see section 5.3 'Statements and imperatives') and were not there at entry are concatenated with the values of any *Output Local Variables* to form the results of the function. The variables created on entry are terminated and the variable associated with each identifier reverts to what it was on entry. There is no change in the values of variables which were previously associated with the formal or local variable identifiers and have now been reinstated. The values of formal and frozen variables are lost. The frozen formals will be reinitialised from the frozen value list on the next entry to the function normally with the same values as last time; the frozen value list can be changed by using *frozval* (see section 8.7 'Functions').

4.3. Nonlocal variables

Variables which occur in a function body and are not locals (i.e. declared in the body) or formals (i.e. elements of the formal parameter list) are called *Nonlocal* to the function. They may be globals or locals of some outer function body which textually encloses it. Care must be taken not to apply a function with nonlocals in a hole in the extent of some of its nonlocals (see section 3.2 'Declaration and initialisation') or outside their extent. Mention of the identifier of such a nonlocal would refer to a quite different variable currently associated with that unique name. The difficulty can arise for recursive functions. Analogous trouble may arise if *nonunique* is used.

To avoid such difficulties a frozen formal may be used instead of the non-local, provided that it is not desired to assign a new value to the nonlocal as a result of the call. The frozen formal can be initialised by partial application to the value that the non-local would have taken. (Note that the frozen formals can be used in this way to give the equivalent of CPL fixed functions, see *CPL Reference Manual* privately circulated by C. Strachey, Programming Research Unit, Oxford University.) In cases where assignment to the non-local is desired a frozen formal can be used and initialised to take a reference (see section 8.1 'References') as value. The component of this reference can then be assigned to, and so long as the reference is made the value of some other exterior variable the value is accessible outside the function body.

4.4. Partial application

In section 4.2 'Application of functions' we explained the method of applying a function to its arguments. There is a process somewhat analogous to application called *Partial Application*. By this means some of the formal parameters of a function may be made into *Frozen Formals*, producing a new function with fewer arguments. The frozen formals are always initialised to a fixed value when the function is applied and do not require any corresponding actual parameters (see however section 8.7 'Functions' for means of altering this fixed value). In other words the actual parameters corresponding

to the frozen formals are supplied once and for all on partial application. The values of the frozen formals are called the *Frozen Value List*.

For example by partially applying the two argument function 'multiply' to 2 we get a one argument function to double a number, and by partially applying it to 3 we get a function to triple a number. These two functions can coexist, and in general one function can be used to generate any number of others by partial application.

More formally we say that a function f of m arguments may be partially applied to an n -tuple of actual parameters with $n \leq m$. We assume for the moment that f has no frozen formals. The partial application produces a new function f' with $m-n$ ordinary formals corresponding to the first $m-n$ formals of f , and n frozen formals corresponding to the last n formals of f . The function f' has a frozen value list consisting of the n items supplied as actual parameters of the partial application.

If f itself has some frozen formals already, say k of them, then f' will have $n+k$ frozen formals and $n+k$ corresponding items in its frozen value list.

The standard function *partapply* takes a function as its first argument and a list as its second argument, and partially applies the function to the elements of the list.

partapply $\in$ function, list $\Rightarrow$ function

Note that partial application constructs a new function with a particular frozen value list, it does not alter the original function in any way. A function which has been produced as the result of partial application is called a *Closure Function*. The frozen values of a closure function can be selected or updated (see section 8.7 'Functions').

If a doublet (see section 4.5 'Doublets') is partially applied to one or more items it produces a new doublet. The selector of the new doublet is obtained by partially applying the selector of the original doublet to the given items. The update routine of the new doublet is obtained by partially applying the update routine of the original doublet to the given items.

A special syntactic form is also available for partial application. It is similar to that for ordinary application (see section 5.1 'Expressions').

$\langle \text{partial application bracket} \rangle = (\% | \%)$

$\langle \text{partial application} \rangle ::= \langle \text{non-operation identifier} \rangle (\% \langle \text{expression list} \rangle \%)$
 $| \langle \text{lambda expression} \rangle (\% \langle \text{expression list} \rangle \%)$

The value of the variable currently associated with the identifier is partially applied to the concatenation of the expressions in the expression list. Thus for example:

```
vars c; cons(%[is a number]%)  $\rightarrow$  c;
c(1)  $\Rightarrow$ 
** [1 is a number]
c(2)  $\Rightarrow$ 
** [2 is a number]
```

```
function  $f$   $x$   $y$   $z$ ; .. etc. end;
 $f(\% y1, z1 \%)\rightarrow f1; f1(x1)\Rightarrow$ 
```

4.5. Doublets

When dealing with data structures, functions called selectors are defined which may be applied to a structure to produce its components (*see* section 7.1 'Functions of data structures'). To each selector there corresponds an update routine which alters the value of the component in the structure to a given new value.

Any function may have an update routine associated with it. This will normally only be done for selector functions. The function is then called a *Doublet*. When a function is created using a lambda expression its associated update routine is not defined. An update routine may be associated with it by using the doublet *updater*, (*see* section 8.7 'Functions').

When a variable whose value is a doublet is used as the operator of a compound expression the selector function of the doublet is applied. But when such a variable is used as the operator of a quasi compound expression (i.e. as part of a destination of an assignment) the update routine is applied.

It is convenient to extend our notation for functions (*see* section 1.5 'Notation for functions') using the new symbol ' $\Rightarrow$ ' to express concisely the domain and range of the selector and update routines of a doublet. Thus if f is a doublet we write

$$f \in d1, \dots, dk \Rightarrow r$$

meaning that f has a selector s

$$s \in d1 \dots, dk \Rightarrow r$$

and an update routine u

$$u \in r, d1, \dots, dk \Rightarrow ()$$

Example:

The standard function hd used in list processing (*see* section 8.3 'Lists') is a doublet.

```
vars  $l$ ; [ $1$   $2$   $3$   $4$ ] $\rightarrow l$ ;  $hd(l) \Rightarrow$ 
**  $1$ 
 $5 \rightarrow hd(l)$ ;  $l \Rightarrow$ 
** [ $5$   $2$   $3$   $4$ ]
 $hd(l) \Rightarrow$ 
**  $5$ 
function second  $l$ ;  $hd(tl(l))$  end;
lambda  $x$   $l$ ;  $x \rightarrow hd(tl(l))$  end  $\rightarrow updater(second)$ ;
second( $l$ )  $\Rightarrow$ 
**  $2$ 
 $6 \rightarrow second(l)$ ;  $l \Rightarrow$ 
** [ $5$   $6$   $3$   $4$ ]
```


4.6. Arithmetic operations

In sections 2.2 'Integers' and 2.3 'Reals' a number of standard functions were introduced for performing arithmetic on integers and reals.

We say that an item is a *Number* if it is either a real or an integer. Arithmetic on numbers is performed by the following standard operations:

Operation	Precedence	Explanation	Result
<	7	less than	truthvalue
>	7	greater than	truthvalue
= <	7	less than or equal	truthvalue
> =	7	greater than or equal	truthvalue
+	5	add	real or integer
-	5	subtract	real or integer
*	4	multiply	real or integer
/	4	divide	real
↑	3	exponent	real

These are defined in terms of *intadd*, *realadd*, etc. and *isreal*, *isint* and *realof*. +, - and * produce an integer result if both arguments are integer otherwise a real result.

5. EXPRESSIONS AND STATEMENTS

5.1. Expressions

An *Expression* is either a simple expression, a compound expression, a conditional expression or an imperative expression (see section 5.3 'Statements and imperatives').

A *Simple expression* is either an identifier or a *Constant*, a constant being an integer, a real or a structure constant. If the simple expression is an identifier then its value is the value of the variable currently associated with that identifier. If it is a constant then its value is the item denoted by the constant. A *Structure Constant* is either a lambda expression which is dealt with in section 4.1 'Definition of functions' and in section 8.7 'Functions', a word constant, a string constant or a list constant, all of which are dealt with in section 8 'Standard structures'.

A *Compound expression* has an *Operator* which is an expression and some *Operands* which are an expression list. The value of a compound expression is found by evaluating the operands and evaluating the operator, whose value should be a function (see section 4.5 'Doublets' for the case where the operator is a doublet). The sequence in which these evaluations are carried out is not defined. The function obtained from the operator is then applied to the *n*-tuple obtained by evaluating the operands. The case where the number of arguments required by the function is not equal to the number of items obtained by evaluating the operands is dealt with in section 5.3 'Statements

and imperatives'. The results of this application are the value of the expression. Thus the value of the expression is an n -tuple, with $n=0$ if the function is a routine.

Evaluation of conditional expressions is described in section 6.1 'Conditional expressions', and that of imperative expressions in section 5.3 'Statements and imperatives'.

An expression list is evaluated by evaluating the expressions of which it consists and concatenating the results. The order in which the evaluations are made is not defined. The order in which the results of evaluating the expressions are concatenated is the order in which the expressions occur.

The syntax of expressions is given below. There are a number of syntactic forms for compound expressions. A further explanation of the syntax is given in section 5.2 'Precedence'.

$\langle \text{non-operation identifier} \rangle ::= \langle \text{identifier} \rangle \mid \text{nonop} \langle \text{operation} \rangle$
 $\langle \text{constant} \rangle ::= \langle \text{integer} \rangle \mid \langle \text{real} \rangle \mid \langle \text{structure constant} \rangle$
 $\langle \text{structure constant} \rangle ::= \langle \text{lambda expression} \rangle \mid \langle \text{quoted word} \rangle \mid \langle \text{string constant} \rangle \mid \langle \text{list constant} \rangle$
 $\langle \text{simple expression} \rangle ::= \langle \text{non-operation identifier} \rangle \mid \langle \text{constant} \rangle$
 $\langle \text{operation} \rangle ::= \langle \text{identifier} \rangle$
 $\langle \text{parentheses} \rangle ::= ()$
 $\langle \text{compound expression} \rangle ::= \langle \text{non-operation identifier} \rangle (\langle \text{expression list} \rangle)$
 $\quad \mid \langle \text{expression ?} \rangle \langle \text{operation} \rangle \langle \text{expression ?} \rangle$
 $\quad \mid \langle \text{closed expression ?} \rangle \langle \text{dot operator} \rangle$
 $\quad \mid \langle \text{structure expression} \rangle$
 $\langle \text{closed expression} \rangle ::= \langle \text{simple expression} \rangle \mid \langle \text{list expression} \rangle$
 $\quad \mid \langle \text{conditional expression} \rangle$
 $\langle \text{dot operator} \rangle ::= . \langle \text{non-operation identifier} \rangle$
 $\langle \text{structure expression} \rangle ::= \langle \text{partial application} \rangle \mid \langle \text{list expression} \rangle$
 $\langle \text{expression list} \rangle ::= \langle \text{expression ?} \rangle , \langle \text{expression list} \rangle \mid \langle \text{expression ?} \rangle$
 $\langle \text{expression} \rangle ::= \langle \text{simple expression} \rangle \mid \langle \text{compound expression} \rangle$
 $\quad \mid \langle \text{conditional expression} \rangle \mid \langle \text{imperative expression} \rangle$
 $\quad \mid (\langle \text{expression list} \rangle)$

Examples:

$\langle \text{simple expression} \rangle$ e.g. $::= x \mid \text{nonop} + \mid 3 \mid \text{lambda } x; x+1 \text{ end} \mid [3 \ 5 \ 9]$
 $\langle \text{operation} \rangle$ e.g. $::= + \mid *** \mid \text{adjoin}$
 $\langle \text{compound expression} \rangle$ e.g. $::= f(x+1, y) \mid a*(b+c) \mid x.hd$
 $\quad \mid f(\% x \%) \mid [\% x, x+1, x+2 \%]$
 $\langle \text{expression} \rangle$ e.g. $::= a \mid g(h(x+1)) \mid \text{if } x=0 \text{ then } y \text{ else } z \text{ close}$
 $\quad \mid (x+1 \rightarrow x; y+1 \rightarrow y; x*y)$
 $\quad \mid (x, y+1, z-1)$

The various syntactic forms of compound expressions denote the operator and operands in the following way:

(i) $\langle \text{non-operation identifier} \rangle (\langle \text{expression list} \rangle)$. Here the operator is the identifier and the operands are the expression list.

(ii) $\langle \text{expression ?} \rangle \langle \text{operation} \rangle \langle \text{expression ?} \rangle$. This is equivalent to: **nonop** $\langle \text{operation} \rangle (\langle \text{expression ?} \rangle, \langle \text{expression ?} \rangle)$ which is a special case of (i) above.

(iii) $\langle \text{closed expression ?} \rangle \langle \text{non-operation identifier} \rangle$. This is equivalent to: $\langle \text{non-operation identifier} \rangle (\langle \text{closed expression ?} \rangle)$ which is a special case of (i) above.

(iv) $\langle \text{structure expression} \rangle$. This is equivalent to (i) above with a special identifier for the operand. The exact rules are given in section 4.4 'Partial Application' and section 8.3 'Lists'.

Note that there is no syntactic provision above for compound expressions whose operator is an expression other than an identifier.

5.2. Precedence

If a compound expression or quasi compound expression is of the form

$\langle \text{expression ?} \rangle \langle \text{operation} \rangle \langle \text{expression ?} \rangle$

the operator is the operation. In this case ambiguity might arise in the analysis of expressions such as

$\langle \text{expression} \rangle \langle \text{operation} \rangle \langle \text{expression} \rangle \langle \text{operation} \rangle \langle \text{expression} \rangle$

which could be analysed with association to the left or to the right. This ambiguity is resolved by the notion of precedence. A precedence is a positive integer between 1 and 7 associated with an operation identifier. It is set by a declaration and can only be changed by cancellation. The operator of a sequence of expressions containing one or more operations is the operation of highest precedence or if there is more than one operation of highest precedence the rightmost of these.

It must be made clear that the difference between an operation and any other identifier which is restricted to having function values is purely a syntactic one.

It may be desired to use an operation in a context other than as the operator of a compound expression. If so it must be prefixed with the word **nonop** in which case it is treated syntactically like any other identifier. The use of **nonop** overrules the precedence of the identifier but does not remove restriction of its values to functions. This facility enables operations to appear as operands and enables assignment to operations.

Example:

> has precedence 7, + and - have precedence 5 and * has precedence 4.

$5 - x + 2 * y > 1 + 2$ is the same as $((5 - x) + (2 * y)) > (1 + 2)$.

5.3. Statements and imperatives

A statement is either an assignment, a goto statement, a machine code instruction or an expression list. It may be labelled.

An imperative is either a declaration or a statement.

The syntax is:

$$\begin{aligned} \langle \text{statement} \rangle &::= \langle \text{assignment} \rangle \mid \langle \text{goto statement} \rangle \\ &\quad \mid \langle \text{code instruction} \rangle \mid \langle \text{expression list} \rangle \\ &\quad \mid \langle \text{labelled statement} \rangle \\ \langle \text{imperative} \rangle &::= \langle \text{declaration} \rangle \mid \langle \text{statement} \rangle \\ \langle \text{imperative sequence} \rangle &::= \langle \text{imperative} \rangle; \langle \text{imperative sequence} \rangle \mid \\ &\quad \langle \text{imperative ?} \rangle \end{aligned}$$

Example:

$$\begin{aligned} \langle \text{imperative sequence} \rangle \text{e.g.} &::= \text{loop} : x-1 \rightarrow x; f(x) \rightarrow y; \text{if } x > 0 \text{ then goto} \\ &\quad \text{loop;} \\ &\quad \mid x+1 \rightarrow y; y; u \rightarrow y; \rightarrow z; \end{aligned}$$

The evaluation of an *Imperative Sequence* consists of evaluating the statements in the sequence in which they occur, except when a goto statement occurs and the sequence continues at the point indicated by the goto statement.

An *Imperative Expression* may be formed from an imperative sequence.

The syntax is:

$$\langle \text{imperative expression} \rangle ::= (\langle \text{imperative sequence} \rangle)$$

The *Stack* is an ordered sequence of items. The last item to be added to this sequence is said to be on *Top of the Stack*. Items can be added to the top of the stack or removed from the top of the stack. On entry to the POP-2 system the stack is empty. When a statement is evaluated any results produced are added to the top of the stack. The results of an imperative sequence are the items left on the stack when the sequence has been evaluated.

Evaluation of a statement which is a compound expression may affect the stack as follows. If the number of arguments required by the function obtained by evaluating the operator is not the same as the number of items produced by evaluating the operands, these items are loaded on to the stack in sequence. The function then takes its arguments off the stack, the last argument being the one which was on the top of the stack. Thus suppose that the function requires m arguments and the operands yield n items. If $m > n$ the first $m-n$ arguments are taken off the stack. If $m < n$ the first $n-m$ items produced by the operands are left on the stack. If $m = n$ the stack is not affected by evaluating the compound expression. Exactly analogous remarks apply to quasi compound expressions.

5.4. Labels and goto statements

A *Label* may be attached to a statement. Evaluation of a *Goto Statement* using that label causes the sequence of evaluation to be changed so that the labelled statement is evaluated next. A goto statement may not refer to a label outside the function body in which it occurs. If a goto statement occurs

in an operand of a compound or quasi compound expression it may not refer to a label outside that operand. The syntax is:

$\langle \text{labelled statement} \rangle ::= \langle \text{label} \rangle : \langle \text{statement ?} \rangle$
 $\langle \text{goto statement} \rangle ::= \text{goto } \langle \text{label} \rangle \mid \text{return}$
 $\langle \text{label} \rangle ::= \langle \text{identifier} \rangle$

The statement **return** causes transfer of control to the exit of the innermost current function body. There is a standard macro exit which is synonymous with **return close**.

If an identifier or sign is used for a label it may not appear as an identifier associated with a variable in the text constituting that function body.

Goto statements and labelled statements may only occur inside a function body.

Note that a label is not an item.

Example:

loop: $x+1 \rightarrow x$; $y*y \rightarrow y$;
 if $x=0$ then goto *loop close*.

5.5. Assignment

An *Assignment* consists of a *Source*, which is an expression or sequence of expressions and a *Destination List*, which is a sequence of elements each of which is either an identifier or a *Quasi Compound Expression*.

A quasi compound expression has an operator which is an expression and some operands, i.e. a sequence of expressions (possibly an empty sequence). Note that a quasi compound expression is not an expression and cannot be evaluated alone to produce an item; it is merely a component of an assignment. It is syntactically the same as a compound expression but cannot be evaluated in isolation.

An assignment is evaluated as follows. First the source is evaluated to yield an n -tuple (where $n \geq k$, k being the number of destination elements). The last k elements of this n -tuple, which we will call the *Source Items*, are then taken in sequence starting from the last and each source item is combined with the corresponding destination element (taken in sequence starting from the first) as follows:

(i) If the destination element is a variable the source item becomes the new value of that variable.

(ii) If the destination element is a quasi compound expression the operator and operands of this expression are evaluated. The value of the operator must be a doublet (see section 4.5 'Doublets') and its update routine is applied to the concatenation of the source item and the values of the operands.

The syntax of quasi compound expression is given below. A further explanation of this syntax is given in section 5.2 'Precedence'.

$\langle \text{quasi compound expression} \rangle ::= \langle \text{non-operation identifier} \rangle (\langle \text{expression list} \rangle)$

$\langle \text{expression ?} \rangle \langle \text{operation} \rangle \langle \text{expression ?} \rangle$
 $\mid \langle \text{closed expression ?} \rangle$
 $\langle \text{dot operator * ?} \rangle$

The syntax of assignments is:

$\langle \text{assignment} \rangle ::= \langle \text{expression list} \rangle \langle \text{destination *} \rangle \mid \langle \text{function definition} \rangle$
 $\mid \langle \text{macro definition} \rangle$
 $\langle \text{destination} \rangle ::= \rightarrow \langle \text{non-operation identifier} \rangle \mid \rightarrow \langle \text{quasi compound expression} \rangle$

Example:

$\langle \text{assignment} \rangle \text{e.g.} ::= x + 1 \rightarrow y \mid u + v \rightarrow a(i, j) \mid x // y \rightarrow u \rightarrow v$

In the second example $a(i, j)$ is a quasi compound expression and the whole assignment is a euphemism for $a1(u + v, i, j)$ where $a1$ is the update routine of the doublet a .

Function definitions and macro definitions are special syntactic forms for assignments.

5.6. Comments

The word **comment** and all characters after it up to and including the next semicolon are ignored.

6. CONDITIONALS

6.1. Conditional expressions

A conditional expression is composed of three components which we will call the *Condition*, the *Consequent* and the *Alternative*. The condition is an expression with a single result, a conjunction or a disjunction (see section 6.2 'Conjunctions and disjunctions'). The consequent and the alternative are imperative sequences each having the same number of results. The method of evaluation of a conditional expression is as follows:

The condition is first evaluated. If its value is the truth value *true* then the consequent is evaluated and its value becomes the value of the expression. But if the value of the condition is the truth value *false* then the alternative is evaluated and its value becomes the value of the expression.

The alternative of a conditional expression may be omitted if it is an empty imperative sequence.

It will often happen that the alternative is itself a conditional expression. The syntax of conditionals is arranged to provide a compact notation to express this:

$\langle \text{conditional body} \rangle ::= \langle \text{imperative sequence} \rangle$
 $\langle \text{conditional expression} \rangle ::= \text{if } \langle \text{condition} \rangle \text{ then } \langle \text{conditional body} \rangle$
 $\quad \langle \text{elseif clause *} ? \rangle \langle \text{else clause ?} \rangle \text{ close}$

$\langle \text{elseif clause} \rangle ::= \text{elseif } \langle \text{condition} \rangle \text{ then } \langle \text{conditional body} \rangle$
 $\langle \text{else clause} \rangle ::= \text{else } \langle \text{conditional body} \rangle$

Example:

$\langle \text{conditional expression} \rangle \text{e.g.} ::= \text{if } x > 0 \text{ and } x < 3 \text{ then } y \text{ elseif } x > 3 \text{ then } z$
 $\text{else } 0 \text{ close}$
 $\quad | \text{ if } x = 0 \text{ then } 1 \rightarrow y \text{ close}$

If there are no elseif clauses the conditional body is the consequent and the else clause is the alternative (which may be omitted). If there are elseif clauses then the first expression is the condition, the second is the consequent and the remainder is the alternative, and it is to be regarded as the conditional expression obtained by replacing the first elseif by if and inserting an extra close before the close, e.g.

$\text{if } p \text{ then } x \text{ elseif } q \text{ then } y \text{ else } z \text{ close}$

is equivalent to

$\text{if } p \text{ then } x \text{ elseif } q \text{ then } y \text{ else } z \text{ close close.}$

6.2. Conjunctions and disjunctions

A *Conjunction* is composed of two component expressions each producing a single result. The method of evaluating a conjunction is to evaluate the first component expression and if its value has the truth value *false* the value of the conjunction has truth value *false*, otherwise the second expression is evaluated and the conjunction has a truth value equal to that of the second component expression.

A *Disjunction* is composed of two component expressions each producing a single result. The method of evaluating a disjunction is to evaluate the first component expression and if its value has the truth value *true* the value of the disjunction has truth value *true*, otherwise the second expression is evaluated and the disjunction has a truth value equal to that of the second component expression.

A number of conjunctions and disjunctions can be combined to form a condition.

The syntax is:

$\langle \text{condition} \rangle ::= \langle \text{expression} \rangle \text{ and } \langle \text{condition} \rangle \mid \langle \text{expression} \rangle \text{ or } \langle \text{condition} \rangle$
 $\quad \mid \langle \text{expression} \rangle$

These three kinds of conditions are respectively a conjunction, a disjunction and an expression.

Thus **and** and **or** associate to the right.

$\langle \text{condition} \rangle \text{e.g.} ::= x < 10 \text{ and } x > 0 \mid x > 10 \text{ or } x < 0 \mid \text{null}(x) \mid b$
 $\quad \mid p(x) \text{ and } q(x) \text{ or } r(x)$

In the last example the following cases can occur ('-' means that the expression is not evaluated)

$p(x)$	$q(x)$	$r(x)$	value of condition
false	—	—	false
true	false	false	false
true	false	true	true
true	true	—	true

7. DATA STRUCTURES

7.1. Functions of data structures

A *Data Structure* is a compound item which has other items as its *Components*. For each class of data structures there is a family of functions called the *Characteristic Functions* acting upon structures of that class. These functions are a constructor, a destructor, selectors and update routines. A given compound item may represent a number of different data structures by being used in association with more than one family of functions and hence having different components.

Given values for its components it is possible to construct a data structure using a *Constructor* function, say c .

$c \in \text{component}, \dots, \text{component} \Rightarrow \text{data structure}$

It is possible to select the value of a component of a data structure. For each component there is a *Selector* function, say si .

$si \in \text{data structure} \Rightarrow \text{component}$

It is possible to update a component of a data structure, i.e. to give it a new value. For each component there is an *Update Routine*, say ui :

$ui \in \text{component}, \text{data structure} \Rightarrow ()$

When a data structure is updated the old version is overwritten.

It is convenient to define another function called a *Destructor* function, say d , which is the inverse of the constructor, i.e. given a data structure it produces its components as results.

$d \in \text{data structure} \Rightarrow \text{component}, \dots, \text{component}$

After applying the destructor to a structure, the structure is deleted.

There is a relation called equality (see section 2.1 'Simple and compound items') which may hold between two compound items. It is denoted by the standard function $=$ (an operation of precedence 7). This function is also defined for simple items with the usual meaning.

$= \in \text{item}, \text{item} \Rightarrow \text{truthvalue}$

Thus if the value of an expression ' $E1$ ' is equal to the value of an expression ' $E2$ ' then the expression

$E1 = E2$

has value *true*.

Equality means that the two compound items contain the same address, i.e. they point to the same area of store. If the items are not equal they point to entirely different areas of store. We say that a compound item is *Copied at the Top Level* if a new item is formed pointing to a new area of store which contains items equal to those of the given compound item. The new item and the previous one are not equal. They are however *Equivalent*.

Equivalent compound items are defined as items which are either equal or all of whose components are equivalent.

Updating an item alters a component item in the store area pointed to by the item but does not cause copying.

We will now give a more formal explanation of equality, but the model in terms of addresses and storage may be kept in mind.

Equality is an equivalence relation, i.e. it is

- (i) reflexive ($x=x$);
- (ii) symmetric (if $x=y$ then $y=x$); and
- (iii) transitive (if $x=y$ and $y=z$ then $x=z$).

It has the following other properties:

(iv) The value of a formal parameter variable is equal to the corresponding actual parameter.

(v) If an item is assigned to a variable then the value of the variable is equal to that item.

(vi) An item, other than a word or simple item, which is read in (see section 9.1 'Input') is not equal to any other item.

(vii) The rules for equality of words are given in section 8.6 'Words'.

(viii) Two integers or two reals are equal according to the usual rules of arithmetic. An integer is never equal to a real.

Items are equal only if their equality follows from the above properties.

We can now state some relationships between the various functions on data structures. We will use a and b for data structures, $x_1, \dots, x_i, \dots, x_k$ for items occurring as components, $s_1, \dots, s_i, \dots, s_k$ for selectors, $u_1, \dots, u_i, \dots, u_k$ for update routines, c for a constructor and d for a destructor.

(i) $s_1(a), \dots, s_k(a)$ is the same n -tuple as $d(a)$, i.e. they have equal elements.

(ii) $si(c(x_1, \dots, x_i, \dots, x_k)) = x_i$ is true.

(iii) $c(s_1(x), \dots, s_k(x)) = x$ is always false, but the left-hand expression is equivalent to x .

(iv) After evaluating $ui(x_i, a)$,
 $si(a) = x_i$ is true.

(v) After evaluating $ui(si(a), a)$, a is unchanged.

(vi) If $a=b$ then $ui(x_i, a)$ is evaluated,
 $si(b) = x_i$ is true and $a=b$ is still true.

(vii) From (i) and (ii) above $d(c(x_1, \dots, x_k))$ is the same n -tuple as $x_1, \dots, x_k$, i.e. they have equal elements.

If a and b are data structures and a is not equal to b and updating a com-

ponent of a also updates some component of b then a and b are said to *Share*.

When we wish to discuss a class of data structures which do not all have the same number of components (such as strips, *see* section 7.3 'Strips') it is convenient to define a *General Selector* function and a *General Update Routine*.

The general selector function, say s , has as arguments, an integer, i , and a data structure. It selects the i th component of the data structure.

$s \in \text{integer, data structure} \Rightarrow \text{component}$

Thus if si is the i th selector $s(i, a) = si(a)$.

Similarly for the general update routine, say u ,

$u \in \text{component, integer, data structure} \Rightarrow ()$

Thus if ui is the i th update routine, $u(xi, i, a)$ has the same effect as $ui(xi, a)$.

The programmer is able to create new kinds of data structures called records and strips (*see* section 7.2 'Records' and 7.3 'Strips'). He can also create functions by methods already described. He may be able to create other kinds of data structures using extra standard functions or machine code but this depends on the implementation. Certain classes of records and strips are standard and these are described in section 8 'Standard structures'.

There are a number of special expressions called 'structure expressions' used to construct these standard structures (*see* section 5.1 'Expressions').

Given a class or several classes of data structures with their associated functions it is possible to define functions which characterise a new family of data structures. Suppose for example that we have a class of structures with two selectors, say $s1$ and $s2$, and components which are full items and members of the same class of structures. We can then define a new class of structures whose selectors are given by:

```
function s11 a; s1(s1(a)) end; function s12 a; s1(s2(a)) end;
function s21 a; s2(s1(a)) end; function s22 a; s2(s2(a)) end;
```

If c is the constructor of the first class we define the new constructor:

```
function cc x1 x2 x3 x4; c(c(x1, x2), c(x3, x4)) end;
```

Note that if it is associated with two or more families of functions the same compound item can represent two or more structures, one of each class. However, for each class of compound items there is one *Primitive Data Structure Class* and other data structures are defined in terms of this primitive class. A primitive data structure does not share with any other primitive data structure.

7.2. Records

A *Record* is a compound item which is a member of a *Record Class*. The *Size* of a set of items is an integer item. If all the items in the set are restricted

to be non-negative integers less than 2^n , the size is the integer n , otherwise if the component is a *Full Item* (i.e. the set is not restricted) the size is the integer 0. For each component of a record there is a size associated with the set of possible values of that component. The *Specification of a Record* is the list of sizes associated with its components. A record class is a set of records which all have the same specification, and this is said to be the specification of the record class. Note that a record class is not an item. A word is associated with each record class.

A family of functions is associated with each record class to form a primitive class of data structures. This family comprises a set of selectors ($\in \text{record} \Rightarrow \text{component}$) and a set of corresponding update routines ($\in \text{component}, \text{record} \Rightarrow ()$), a constructor ($\in \text{component}, \dots, \text{component} \Rightarrow \text{record}$) and a destructor ($\in \text{record} \Rightarrow \text{component}, \dots, \text{component}$). Each selector function may be paired with the corresponding update routine to form a doublet ($\in \text{record} \Rightarrow \text{component}$). The standard function *recordfns* is used to create a new record class. It requires as arguments the word to be associated with the record class, an estimate of the number of records in the record class (this is purely to help in efficient implementation) and the specification of the record class. It produces the constructor, the destructor and the doublets for the record class. The number of its results depends on the length of the specification list. Normally the programmer will immediately assign these resulting functions to variables.

recordfns $\in$ *word*, *integer*, *specification* $\Rightarrow$ *constructor*, *destructor*,
doublet, ..., *doublet*

There is a standard function which converts a record to a list of its components:

datalist $\in$ *record* $\Rightarrow$ *list*

There is a function *dataword* which given a record produces the word associated with its record class.

dataword $\in$ *record* $\Rightarrow$ *word*

The function *copy* copies a record at the top level.

copy $\in$ *record* $\Rightarrow$ *record*

The functions *datalist*, *dataword* and *copy* are defined over records of any class, and whenever *recordfns* is used to create a new record class these three functions are extended to deal with records of that class.

The routine *enddata* may be given the word associated with a record class and removes all records in that class. It also adjusts the three functions just mentioned so that they no longer deal with that record class.

enddata $\in$ *word* $\Rightarrow$ $()$

7.3. Strips

A *Strip* is a compound item which is a member of a *Strip Class*. All components of a strip must have the same size (see section 7.2 'Records')

for definition of size) which is called the *Component Size* of the strip. All strips in a strip class must have the same component size but not necessarily the same number of components. A word is associated with each strip class.

A family of functions is associated with each strip class to form a primitive class of data structures. This family includes a general selector function ($\in integer, strip \Rightarrow component$) and a general update routine ($\in component, integer, strip \Rightarrow ()$). The selector function may be paired with the update routine to form a doublet. It also includes for each strip class an initiator function ($\in integer \Rightarrow strip$). This constructs a strip with the given number of components, but the values of these components are not defined. The initiator may be used with the update function to define a constructor function for strips of the strip class.

The standard function *stripfns* is used to create a new strip class. It takes as arguments the word to be associated with the strip class, an estimate of the total number of all components of all strips in the strip class (this is purely to help in efficient implementation) and the component size of the strip class. It produces as results the initiator function and the doublet for the strip class:

$stripfns \in word, integer, size \Rightarrow initiator, doublet$

There is a standard function which converts a strip to a list of its components. This is *datalist* (see section 7.2 'Records'). There is a function which given a strip produces the word associated with its strip class. This is *dataword* (see section 7.2 'Records').

The function *copy* copies a strip at the top level (see section 7.2 'Records').

The functions *datalist*, *dataword* and *copy* and the routine *enddata* act for strips just as for records.

7.4. Garbage collection

Storage for the construction of data structures is made available by a storage control system. This system must be able to make use of areas of store which have been used but are no longer required. This is achieved by a process known as *Garbage Collection* which is undertaken whenever the system runs short of store. This first of all discovers what items can still be referred to by the programmer, e.g. because they are the value of a variable whose extent has not finished (see section 3.2 'Declaration and initialisation'). Any items which can no longer be referred to are destroyed, i.e. their storage area is returned to the system for use in constructing other items. Since he cannot refer to them the programmer is not aware of this destruction.

If variables refer to compound items which are no longer in use, the garbage collector cannot recover the associated storage. The variable should be reset, e.g. to zero. In the case of identifiers the identifier can be cancelled (see section 3.3 'Cancellation').

To avoid too frequent garbage collection compound items can be deleted, i.e. returned to the storage control system, using the standard routine:

$delitem \in item \Rightarrow ()$

When an item is deleted its components are not deleted.

After an item has been deleted it is no longer available and the onus is on the programmer not to use it. The implementation may not give an error message if he does use it, the value simply not being defined.

8. STANDARD STRUCTURES

8.1. References

There is a standard record class called *References*. These have one component which is a full item. The word associated with the class is "*ref*". Thus before entry to the POP-2 system this class is created using *recordfns*, and the resulting functions are assigned to variables to give the following standard functions:

constructor: $consref \in item \Rightarrow reference$

destructor: $destref \in reference \Rightarrow item$

doublet: $cont \in reference \Rightarrow item$

A reference may be used e.g. as an actual parameter of a function to enable the function to cause side effects by updating the reference.

8.2. Pairs

There is a standard record class called *Pairs*. Records of this class have two components which are both full items. The word associated with the class is "*pair*". Thus before entry to the POP-2 system this class is created using *recordfns*, and the resulting functions are assigned to variables to give the following standard functions:

constructor: $conspair \in item, item \Rightarrow pair$

destructor: $destpair \in pair \Rightarrow item, item$

doublets: $front, back \text{ all } \in pair \Rightarrow item$

An *Atom* is an item which is not a pair. Atoms are recognised by the standard function *atom*.

$atom \in item \Rightarrow truthvalue$

8.3. Lists

There is a standard data structure called a *Link* which is used to construct another data structure called a *List*. Lists in POP-2 include structures analogous to LISP lists, but also structures which compute the elements dynamically (cf. P. J. Landin's 'streams').

The word "*nil*" is used to represent the *Null List* and the standard variable

nil takes this value on entry to the POP-2 system. The null list is also represented by a pair whose front is *true* and whose back is a function.

The standard function *null* recognises the null list

$null \in list \Rightarrow truthvalue$

A list is either the null list or it is a link.

A link is either:

(i) a pair whose front component is any item and whose back component is a list, or

(ii) a pair whose front component is *false* and whose back component is a function with no arguments and one result.

In case (ii) the function is one which when repeatedly applied produces a succession of items, not necessarily all the same, i.e. normally the function will side-effect itself. The last item produced should be the terminator. For example this enables us to convert an input file to a list. Lists with this sort of link are dynamic and some or all of their elements are computed rather than stored statically.

The characteristic functions of a link are:

constructor: $cons \in item, list \Rightarrow link$

destructor: $dest \in list \Rightarrow item, list$

doublets: $hd \in link \Rightarrow item$

$tl \in link \Rightarrow item$

(called the 'head')

(called the 'tail')

These functions are very similar to those for pairs, but in the case of a link of the second kind special precautions are taken to make sure that on applying the selector *tl* the front component is not lost but preserved in a pair. Thus if *x* has a list as its value and *tl(x)* is evaluated there is a side effect on *x*, but matters are so arranged that this side effect is not detectable using the list processing functions. The function *cons* is the same as *conspair* and produces a link of the first kind. The following standard function produces a link of the second kind or the null list given a function of no arguments:

$fntolist \in ((\) \Rightarrow item) \Rightarrow list$

function *fntolist* *f*; *cons*(*false*, *f*) **end**

The other characteristic functions are defined as follows:

First an auxiliary function (*not* standard) to convert the first link of a dynamic list to static form.

function *solidified* *l*; **vars** *f* *x*;

if *isfunc*(*back*(*l*))

then *back*(*l*) $\rightarrow$ *f*; *f*() $\rightarrow$ *x*;

if *x* = *termin* **then** *true* $\rightarrow$ *front*(*l*)

else *x* $\rightarrow$ *front* (*l*); *conspair*(*false*, *f*) $\rightarrow$ *back*(*l*)

close; *l*

else *l*

close

end;

PROBLEM-ORIENTED LANGUAGES

```

function hd l; front(solidified(l)) end;
lambda i l; i→front(solidified(l)) end→updater(hd);

function tl l; back(solidified(l)) end;
lambda i l; i→back(solidified(l)) end→updater(tl);
function dest l; vars f;
    if isfunc(back(l)) then back(l)→f; (f(),l)
    else (front(l),back(l))
    close
end;
function null l;
    if l=nil then true
    elseif isfunc(back(l))
        then if hd(l) or null(solidified(l)) then true else false close
        else false
    close
end;

```

A list may have no components (if it is the null list) or one or more (if it is a link).

If it is a link its first component is the head component of the link and its remaining components are the components of the list which is the tail component of the link. Thus the characteristic functions for lists can be defined in terms of those for links.

There are two special syntactic forms for constructing lists. These are list constants and list expressions. List constants may have lists, integers, reals, words or strings (see section 8.4 'Full strips and character strips') as components. The list is constructed at compile time.

```

<list constant brackets>::=[ | ]
<list constant>::=[<list constant element * ?>]
<list constant element>::=<list constant> | <character group>

```

Example:

```

<list constant>e.g.::=[1 2 DOG CAT] | [[1 2] [4 5] 6]

```

List expressions are formed by evaluating a number of expressions at run time and constructing a list.

```

<list expression brackets>::=[% | %]
<list expression>::=[%<expression list>%]

```

Thus

```

[% %] is equivalent to nil
and [% <expression> %] is equivalent to cons(<expression>, nil)
and [% <expression>, <expression list> %] is equivalent to
cons(<expression>, [%<expression list>%])

```

Example:

```

<list expression>e.g.::=[%x+1, [%x+2, x+3%], tl(y)%]

```

For convenience the following functions are standard:

$next \in list \Rightarrow item$, $list$ (similar to $dest$ but non-destructive)

$::$ (a synonym for $cons$ but an operation of precedence 2)

$\langle \rangle \in list$, $list \Rightarrow list$ (concatenates the lists, an operation of precedence 2).

8.4. Full strips and character strips

Two strip classes are standard.

The first is *Full Strips* with full items as components and associated word "strip". The characteristic functions are:

initiator: $init \in integer \rightarrow full\ strip$

doublet: $subscr \in integer, strip \Rightarrow item$

The second is *Character Strips* (also called 'Strings') (for characters see section 8.6 'Words') with component size 6 and associated word "cstrip". The characteristic functions are:

initiator: $initc \in integer \Rightarrow character\ strip$

doublet: $subsrc \in integer, strip \Rightarrow integer\ of\ size\ 6$

The components of a character strip may be any integers of size not more than 6, they need not necessarily be used to represent characters.

There is a structure constant to construct character strip constants at compile time.

$\langle string\ bracket \rangle ::= ' | \backslash$

$\langle string\ constant \rangle ::= ' \langle string\ constant\ element * ? \rangle \backslash$

$\langle string\ constant\ element \rangle ::= \langle string\ constant \rangle | \langle any\ character\ except\ a\ string\ bracket \rangle$

Example:

$\langle string\ constant \rangle e.g. ::= '... rubbish . please type 'sorry''$

Spaces and newlines are significant in string constants.

There are functions to input and output character strings stored as character strips (see section 9.1 'Input and output'). The external format is as for string constants.

8.5. Arrays

Arrays give a convenient method of accessing and updating structures indexed by integers. An array has components, which are items of a given size. Each component is associated with a sequence of integers called *Subscripts*. The number of subscripts is known as the number of *Dimensions* of the array. An array is a doublet:

$array \in subscript, \dots, subscript \Rightarrow component$

This is in contrast to strips which have a general selector and a general update routine associated with a whole class of strips and take the actual strip referred to as a parameter. Arrays can be formed from strips (or from other data structures) by using partial application. The programmer is free

to do this in any way he chooses but standard functions for creating arrays are provided.

There is a standard function to create a many dimensional array of items of any size. Updating a component of this array does not affect any other component. This function is:

$newanyarray \in boundslist, (subscript, \dots, subscript \Rightarrow component),$
 $strip\ initiator, strip\ doublet \Rightarrow array$

The array produced will normally be immediately assigned to a variable.

The boundslist is a list of integers, these two integers being alternately the lower and upper bounds for each subscript. The second parameter is a function used to initialise the components of the array. It must produce the appropriate component for each combination of subscripts. The strip doublet and strip initiator are the characteristic functions of a strip class whose components are of the same size as that required for the array components.

There is also a standard function to create arrays of full items:

$newarray \in boundslist, (subscript, \dots, subscript \Rightarrow component) \Rightarrow array$

This is obtained by partial application and is equivalent to

$newanyarray (\% \text{ init}, subscr \%).$

8.6. Words

There is a standard record class called *Words*. It has 8 components of size 6 called *Characters*, and a component called the *Meaning*. The word associated with the record class is "word". The standard functions characterising words are:

constructor: $consword \in character, \dots, character, integer \Rightarrow word$

destructor: $destword \in word \Rightarrow character, \dots, character, integer, item$

doublets: $charword \in word \Rightarrow character, \dots, character, integer$
 $meaning \in word \Rightarrow item$

Each character of the POP-2 character set corresponds to a unique integer. The correspondence rule depends on the implementation. Note that the functions above are variadic and work on a variable number of characters followed by that number as an integer. If there are less than 8 characters supplied to the constructor the remaining character components are not defined and they are not produced by the destructor or selector. The constructor does not take a meaning component as argument. The meaning of a word is *undefined* unless the word has been updated to have a particular meaning.

Words may occur in the program as quoted words, i.e. word constants, with the following syntax:

$\langle \text{unquoted word} \rangle ::= \langle \text{letter} \rangle \langle \text{alphanumeric *} \rangle \mid \langle \text{sign *} \rangle$
 $\mid \langle \text{decorated bracket} \rangle \mid \langle \text{bracket decorator} \rangle$
 $\mid \langle \text{separator} \rangle \mid \langle \text{period} \rangle \mid \langle \text{exponent} \rangle \mid \langle \text{quote} \rangle$

$\langle decorated\ bracket \rangle ::= (| (\% | \%)$
 $\quad \quad \quad | [|] | [\% | \%]$
 $\langle quoted\ word \rangle ::= " \langle unquoted\ word \rangle "$

Example:

$\langle quoted\ word \rangle e.g. ::= "big" | "++" | "\%" | ""$

Words may also occur as components of constant lists (see section 8.3 'Lists'). Only the first 8 characters are significant.

Words may also be read as data (see section 9.1 'Input'). Words which occur as constants or are read as data are *Standardised*, i.e. if a word with the same characters already exists no new word is constructed and the compound item produced is the previously existing word, but if no such word exists a new word is constructed with *undef* as its meaning. Words constructed using *consword* are also standardised, but the update routines do not standardise.

8.7. Functions

Functions are compound items. There is no constructor or destructor for functions. They can be constructed by the methods described in section 4.1 'Definition of functions', and they can be deleted by the routine *delitem* (see section 7.4 'Garbage collection'). There is a family of characteristic functions associated with the class of functions to form a primitive class of data structures.

Functions have an accessible component which may be used to associate extra information with the function. It is accessed by the standard doublet

$fnprops \in function \Rightarrow item$

Functions have an update routine (see section 4.5 'Doublets'). For a function constructed by using *lambda* or *function* or *routine* this has initially no defined value. This component may be selected or updated by using a standard doublet:

$updater \in function \Rightarrow routine$

Closure functions i.e. those constructed by partial application, have a doublet to select or update the values of their frozen formals

$frozval \in integer, closure\ function \Rightarrow item$

The integer determines which of the frozen formal values is affected, counting from the front (if a closure function is obtained by successive partial applications only the formals frozen by the last one are counted). There is also a doublet to select the function from which the closure function was constructed or replace it with another function.

$fnpart \in closure\ function \Rightarrow function$

The standard function = follows the usual rules for compound items when applied to functions, i.e. equality is preserved over assignment, updating and actual parameter/formal parameter correspondence but each construction of a function produces a different one.

The following standard function recognises functions:

$$isfunc \in item \Rightarrow truthvalue$$

9. INPUT AND OUTPUT

9.1. Input

Information which is input to the POP-2 system is organised into *Files*, each of which comes from a *Device*.

Before a file can be accessed it must be *Opened*. From then on it can be read one character at a time. Eventually it must be *Closed*.

The naming of files and devices depends on the operating system of the implementation. The names of files are lists and the names of devices may be any item. A device name may refer to more than one device.

There is a standard variresult function *popmess* used for communicating with the operating system

$$popmess \in list \Rightarrow item, \dots, item$$

This is used for various input and output purposes.

To open a file from a given device, *popmess* is used to produce a function to read characters from it i.e. a function $\in () \Rightarrow character$. The list supplied to *popmess* has a head which is an input device name and a tail which is a file name.

To close a file before reaching the end of it, *popmess* is again used. The list supplied to it has a head which is the word "close" and a tail which is a list of one element: a character reading function obtained when the file was opened. No result is produced by *popmess* in this case.

The sequence of characters making up a POP-2 text may be split up into *Character Groups* each of which represents a *Text Item*. A text item is either an integer, a real, a word or a string. It is represented by a character group, thus:

$$\langle character\ group \rangle ::= \langle integer \rangle \mid \langle real \rangle \mid \langle unquoted\ word \rangle \\ \mid \langle string\ constant \rangle$$

Character groups are terminated by spaces or newlines where necessary to separate them from the following character group.

There is a standard function to convert a function which produces a character whenever it is applied into a corresponding one which produces a text item whenever it is applied.

$$incharitem \in (() \Rightarrow character) \Rightarrow ((l) text\ item)$$

The program is input on a standard file called the *Standard Input File* from a standard device called the *Standard Input Device*. There is a standard function to read characterx from the *standard input file*:

$$charin \in () \Rightarrow character$$

The program is compiled from the text item list which is the value of the standard variable *proglis*. Initially this has as value the list of text items from the standard input file. It may be assigned to by the programmer who wishes to compile from a different source.

For convenience there is a standard function *itemread* producing the next item of the list which is the value of *proglis*.

$itemread \in () \Rightarrow \text{text item}$

It is defined thus:—

function *itemread*; *proglis* . *dest* $\rightarrow$ *proglis* **end**;

9.2. Output

Information which is output from the POP-2 system is organised into files, each of which is sent to a device (see section 9.1 'Input').

To open an output device the standard function *popmess* (see Section 9.1 'Input') is used to produce a routine to deliver characters to it i.e. a routine $\in \text{character} \Rightarrow ()$. The list supplied to *popmess* has a head which is an output device name and a tail which is a file name.

There is a standard function to convert a routine which delivers a sequence of characters to an output file into one which delivers a sequence of text items.

$outcharitem \in (\text{character} \Rightarrow ()) \Rightarrow (\text{text item} \Rightarrow ())$

Compiler messages and results of computation are normally output on a standard file called the *Standard Output File* to a standard device called the *Standard Output Device*. There is a standard routine to output characters to the standard output file:

$charout \in \text{character} \Rightarrow ()$

There is a standard variable *cucharout* which contains the routine to output characters to the *Current Output File*. This contains initially the routine for the standard output file but it may be assigned to if a different output file is to be made current. An output file is closed by outputting the terminator.

There are standard routines to output spaces or newlines to the current output file:

$sp \in \text{integer} \Rightarrow ()$

$nl \in \text{integer} \Rightarrow ()$

There is a standard function which outputs any item to this file in some suitable format and produces that item unchanged as its result.

$print \in \text{item} \Rightarrow \text{item}$

There is a standard macro which uses *print* and causes the items on the stack starting at the bottom to be printed on a newline preceded by two asterisks. These items are removed from the stack. In a function body only the top item of the stack is affected. This macro is denoted by the POP-2

identifier $\Rightarrow$ (not to be confused with the $\Rightarrow$ used in this manual to show the type of functions). A semicolon is implied before and after so that immediate evaluation can occur (see section 11.1 'Immediate evaluation').

10. MACHINE CODE

It is possible to insert sections of machine code in an imperative sequence. The rules depend on the implementation. A code instruction is represented by the identifier \$ followed by any sequence of characters which do not include ';

$\langle \text{code instruction} \rangle ::= \$ \langle \text{any sequence of characters other than ;} \rangle$

11. MODES OF EVALUATION

11.1. Immediate evaluation

A POP-2 program consists of a sequence of imperatives and cancellations:

$\langle \text{program element} \rangle ::= \langle \text{imperative} \rangle \mid \langle \text{cancellation} \rangle$

$\langle \text{program} \rangle ::= \langle \text{program element} \rangle ; \langle \text{program} \rangle$

The program elements are evaluated in sequence in the same way as an imperative sequence. Each program element is evaluated as soon as the terminal semicolon and a space has been read by the compiler. The body of any function in the program element will be compiled and kept so that it may be evaluated when that function is applied.

11.2. Macros

A *Macro* is a routine which is applied at compile time.

The definition of a macro routine is similar to that of any other routine except that **macro** is used instead of **routine** and no formal parameters are allowed.

$\langle \text{macro definition} \rangle ::= \text{macro } \langle \text{identifier} \rangle ; \langle \text{function body} \rangle \text{ end}$

A macro, like an operation, is applied whenever it is mentioned and does not need parentheses after it.

Although a macro has no parameters the function *itemread* (see section 9.1 'Input') may be used to read the text items following the macro identifier. There is a standard routine which, when applied in a macro body to a list of text items, concatenates these items to the right of the macro identifier in the program sequence of text items.

$\text{macresults} \in \text{text item list} \Rightarrow ()$

If it is applied more than once it concatenates to the right of the previously inserted items. On exit from the macro the inserted text items are evaluated as program.

Example:

```
macro  $\rightarrow$ ; vars  $x\ y$ ; itemread  $\rightarrow x$ ; itemread  $\rightarrow y$ ;  
    macresults ([% " $\rightarrow$ ",  $y$ , " $\rightarrow$ ",  $x\%$ ])
```

```
end;
```

```
7//2  $\rightarrow q\ r$ ;
```

This is the same as 7//2 $\rightarrow r\rightarrow q$;

The correspondence between a list of text items and POP-2 program is as follows. Syntax words, identifiers and unquoted words are represented by corresponding words in the list. A quoted word is represented by a word with the word quote (consisting of the character quote) before and after it in the list. Integers and reals are represented by integers and reals. String constants are represented by strings.

11.3. Evaluation of program text

A standard function *popval* is provided which will evaluate a list of text items treating it as a POP-2 imperative sequence. The sequence is evaluated immediately. It may contain function definitions and assignments to current variables. Any declarations in it which are not in a function body are global. The list must terminate with the word *goon*. For the correspondence between a list of text items and POP-2 program see section 11.2 'Macros'.

The result of the application of *popval* is the result of evaluating the imperative sequence.

$$popval \in \text{text item list} \Rightarrow \text{item}, \dots, \text{item}$$

Note that *popval* is used to evaluate an imperative sequence at run time and the list of text items may have been produced as the result of computation. It may temporarily affect the standard variable *proglis* (see section 9.1 'Input').

Example:

```
1  $\rightarrow a$ ; popval([vars  $x$ ;  $a+2 \rightarrow x$ ;  $x*x$  goon])  $\Rightarrow$   
**9
```

The standard routine *setpop* may be applied in the imperative sequence. This restores the system to execute mode. The stack is cleared. The variable currently associated with any identifier is not altered. After *setpop* has been applied the rest of the imperative sequence is ignored and all function bodies currently being evaluated are abandoned. The system then evaluates the next program element. *setpop* may also be applied in a function body.

$$setpop \in () \Rightarrow ()$$

ACKNOWLEDGMENTS

This language is a development of R. J. Popplestone's 'POP-1' programming language (see the paper in this volume). The debt to the ALGOL, LISP, CPL and

PROBLEM-ORIENTED LANGUAGES

ISWIM programming languages should be obvious. We are indebted to a number of people in this department and elsewhere for helpful discussion and criticism, to Dr David Park who contributed to discussion of the storage control scheme and to Mrs Margaret Pithie and Miss Eleanor Kerse who typed this report. Mr M. Healy kindly pointed out a number of errors and obscurities in a draft.

The work has been undertaken on a grant from the Science Research Council under the supervision of Dr D. Michie, whose encouragement has been invaluable.

INDICES

TECHNICAL TERMS

- actual parameters: 4.2
- alternative: 6.1
- applied: 4.2
- assignment: 5.5
- atom: 8.2
- body: 4.1
- character groups: 9.1
- character strips: 8.4
- characteristic functions: 7.1
- characters: 8.6
- closed: 9.1
- component size: 7.3
- components: 7.1
- compounds: 2.1
- compound expression: 5.1
- concatenated: 4.2
- condition: 6.1
- conjunction: 6.2
- consequent: 6.1
- constant: 5.1
- constructor: 7.1
- copied at the top level: 7.1
- currently associated: 3.1, 3.2
- data structure: 7.1
- declaration: 3.2
- destination list: 5.5
- destructor: 7.1
- device: 9.1
- dimensions: 8.5
- disjunction: 6.2
- doublet: 4.5
- entry: 4.2
- equivalent: 7.1
- exit: 4.2
- expression: 5.1
- extent: 3.2
- false: 2.4
- files: 9.1
- formal parameter list: 4
- formal: 3.2
- frozen formal: 4.4
- frozen value list: 4.4
- full item: 7.2
- full strips: 8.4
- function: 4.4
- garbage collection: 7.4
- general selector: 7.1
- general update routine: 7.1
- global declaration: 3.2
- global: 3.2
- goto statement: 5.4
- hole in the extent: 3.2
- identifier: 3.1
- imperative expression: 5.3
- imperative sequence: 5.3
- initialisation: 3.2
- items: 2.1
- label: 5.4
- lambda expression: 4.1
- link: 8.3
- list: 8.3
- local declaration: 3.2
- local: 3.2
- macro: 11.2
- meaning: 8.6
- n*-tuple: 4.2
- nonlocal: 4.3
- null list: 8.3
- number: 4.6
- opened: 9.1
- operands: 5.1
- operation: 3.2
- operator: 5.1
- output local list: 4.1
- output local variable: 4.2
- pairs: 8.2
- partial application: 4.4
- primitive data structure class: 7.1
- program device: 9.1
- program file: 9.1
- protected: 3.2
- quasi compound expression: 5.5
- record class: 7.2
- record: 7.2
- references: 8.1
- results device: 9.2
- results file: 9.2
- results: 4.2
- routine: 4.1
- running: 4.2
- scope: 3.2
- selector: 7.1
- share: 7.1
- simple expression: 5.1
- simple: 2.1
- size: 7.2
- source items: 5.5
- source: 5.5
- specification of a record: 7.2
- stack: 5.3
- standard function: 3.2
- standard variable: 3.2
- standard input device: 9.1
- standard input file: 9.1
- standard output device: 9.2
- standard output file: 9.2
- standardised: 8.6
- strip class: 7.3
- strip: 7.3
- structure constant: 5.1
- subscripts: 8.5
- text item: 9.1
- top of the stack: 5.3
- true: 2.4
- truthvalues: 2.4
- unique name: 3.2
- update routine: 7.1
- value: 3.1
- variable: 3.1
- variadic: 4.2
- varresult: 4.2
- words: 8.6

SYNTAX DEFINITIONS

- alphanumeric*: 3.1
- assignment*: 5.5
- binary digit*: 2.2
- binary integer*: 2.2
- bracket decorator*: 1.4
- bracket*: 1.4
- cancellation*: 3.3
- closed expression*: 5.1
- code instruction*: 10.0
- compound expression*: 5.1
- condition*: 6.2
- conditional body*: 6.1
- conditional expression*: 6.1
- constant*: 5.1
- decimal integer*: 2.2
- declaration list element*: 3.2
- declaration*: 3.2
- decorated bracket*: 8.6
- destination*: 5.5
- digit*: 1.4
- dot operator*: 5.1

INDICES

- else clause*: 6.1
- elseif clause*: 6.1
- exponent*: 1.4, 2.3
- expression list*: 5.1
- expression*: 5.1
- formal parameter list*
 - element*: 4.1
- formal parameter list*: 4.1
- function body*: 4.1
- function definition*: 4.1
- function*: 4.1
- goto statement*: 5.4
- identifier*: 3.1
- imperative sequence*: 5.3
- imperative*: 5.3
- integer*: 2.2
- label*: 5.4
- labelled statement*: 5.4
- lambda expression*: 4.1
- lambda*: 4.1
- letter*: 1.4
- list constant brackets*: 8.3
- list constant element*: 8.3
- list constant*: 8.3
- list expression brackets*: 8.3
- list expression*: 8.3
- macro definition*: 11.2
- non-operation identifier*: 5.1
- octal digit*: 2.2
- octal integer*: 2.2
- operation*: 5.1
- output local list*: 4.1
- output local list element*: 4.1
- parentheses*: 5.1
- partial application*: 4.4
- period*: 1.4
- program element*: 11.1
- program*: 11.1
- quasi compound expression*: 5.5
- quote*: 1.4
- quoted word*: 8.6
- real*: 2.3
- restriction*: 3.2
- restrictor*: 3.2
- separator*: 1.4
- sign*: 1.4
- simple expression*: 5.1
- statement*: 5.3
- string bracket*: 8.4
- string constant element*: 8.4
- string constant*: 8.4
- string quote*: 1.4
- structure constant*: 5.1
- structure expression*: 5.1
- sub ten*: 1.4
- unquoted word*: 8.6

STANDARD FUNCTIONS AND VARIABLES

- <:* 4.6
- >:* 4.6
- =<:* 4.6
- >=:* 4.6
- +*: 4.6
- : 4.6
- **: 4.6
- /*: 4.6
- ↑*: 4.6
- //2.2*
- =*: 2.1, 7.1
- ⇒*: 9.2
- ::*: 8.3
- ◇*: 8.3
- array*: 8.5
- atom*: 8.2
- back*: 8.2
- booland*: 2.4
- boolor*: 2.4
- charin*: 9.1
- charout*: 9.2
- charword*: 8.6
- closefile*: 9.1
- cons*: 8.3
- conspair*: 8.2
- consref*: 8.1
- consword*: 8.6
- cont*: 8.1
- copy*: 7.2
- cucharout*: 9.2
- datalist*: 7.2
- dataword*: 7.2
- delitem*: 7.4
- dest*: 8.3
- destpair*: 8.2
- destref*: 8.1
- destword*: 8.6
- enddata*: 7.2
- fnpart*: 8.7
- fnprops*: 8.7
- fntolist*: 8.3
- front*: 8.2
- frozval*: 8.7
- hd*: 8.3
- incharitem*: 9.1
- init*: 8.4
- initc*: 8.4
- intadd*: 2.2
- intgr*: 2.2
- intgreg*: 2.2
- intle*: 2.2
- intleeq*: 2.2
- intminus*: 2.2
- intmult*: 2.2
- intof*: 2.3
- intplus*: 2.2
- intsign*: 2.2
- intsub*: 2.2
- iscompnd*: 2.1
- isfunc*: 8.7
- isinteger*: 2.1
- isreal*: 2.1
- itemread*: 9.1
- logand*: 2.2
- lognot*: 2.2
- logor*: 2.2
- logshift*: 2.2
- macresults*: 11.2
- meaning*: 8.6
- newanyarray*: 8.5
- newarray*: 8.5
- next*: 8.3
- nextchar*: 9.1
- nl*: 9.2
- not*: 2.4
- null*: 8.3
- outcharitem*: 9.2
- outitem*: 9.2
- partapply*: 4.4
- popmess*: 9.1
- popval*: 11.3
- print*: 9.2
- proglis*: 9.1
- realadd*: 2.3
- realdiv*: 2.3
- realgr*: 2.3
- realgreg*: 2.3
- realle*: 2.3
- realleeq*: 2.3
- realmult*: 2.3
- realminus*: 2.3
- realof*: 2.3
- realplus*: 2.3
- realsign*: 2.3
- realsub*: 2.3
- recordfns*: 7.2
- setpop*: 11.3
- sp*: 9.2
- stripfns*: 7.3
- subscr*: 8.4
- subscrc*: 8.4
- tl*: 8.3
- updater*: 8.7

INDICES

SYNTAX WORDS

:: 5.4

→: 5.5

⇒: 4.1

and: 6.2

cancel: 3.3

close: 6.1

else: 6.1

elseif: 6.1

end: 4.1

function: 3.2

goto: 5.4

if: 6.4

lambda: 4.1

macro: 11.2

nonop: 5.1

operation: 3.2

or: 6.2

return: 5.4

routine: 4.1

then: 6.1

vars: 3.2

INDEX

- Alexander 2 181
- Algol 1 4, 11, 13, 20, 38, 112; 2 3-12, 17, 33-5, 43, 102, 103, 119, 134, 185, 209, 212
- Ames 1 173
- Antony 1 207
- applicators 2 35
- arithmetic operations 2 224
- arrays 2 188, 239
- assignment 2 228
- Atlas autocode 1 238, 241
- Atlas computer 1 4, 130
- backtrack analysis 1 91, 94
- Backus 2 35, 212
- Backus-Naurform 1 20
- Backus notation 1 11
- Baldrige 1 163
- Balzer 2 21
- Barron 1 257; 2 210
- Bernstein 2 97
- Beth trees 1 33
- BESM 2 98
- Birkhoff 1 32
- black box problems 2 146
- block form 2 29
- BNF grammar 1 21
- Bobrow 1 116
- Böhm 2 30
- Boolean expression 1 185
- Boolean variables 1 21
- Borel 2 105
- Borel-von Neumann theory 2 93, 105 166
- Bousfield *et al.* 1 156
- boxes 2 137, 145, 150, 151
- Brackett 2 185
- Brooker 2 53, 212
- Burstall 2 34, 209
- buying chess 2 105
- Cannon 2 145
- Capablanca 2 94
- Carnegie system 1 20
- Carson 2 57
- Cartwright 1 4
- Cayley 1 4
- Chambers 2 140
- checkers 2 42
- chess 2 90, 91, 93
- Chomsky 1 21, 211
- Church 1 48; 2 4, 5, 34, 68
- closed-class items 2 176
- Collins 2 153, 159
- COMIT 2 57
- command sequence 1 6
- compiler language 1 236
- compiler mode 1 249
- complex games 2 91
- concatenation 2 212
- conceptualisation 2 33
- conditional expressions 2 229
- conditionals 2 229
- conjunctions and disjunctions 2 230
- connection matrix 1 6
- constructors 2 35
- control data 3600 2 57
- Cook 2 65
- Cooper 2 21, 27, 28, 93
- Coral 2 38
- Coxeter 2 110
- CPL 2 4, 210
- CPL reference manual 2 221
- critical path analysis 1 6
- Croes 2 125, 126
- Curry 2 4
- CYCLOPS 1 209
- Dallemand 2 154, 155
- data structures 2 12, 231
- Davis 1 49; 2 59
- DDPS 2 40
- decision rules 2 148
- declarations 1 230f
- defined functions 1 23
- definitions: iterative 1 24; list functions 1 23; multiplication functions 1 23; recursive 1 22
- de Groot, 2 91, 96, 99, 108
- Deese 1 157, 166
- Denes 1 200
- Dennis 2 38
- designators 2 35
- Donaldson 2 145
- Doog 2 100
- Doran 2 119, 120, 123, 131
- doublets 2 223
- DT 2 2 30
- dynamic stack system 1 258
- Earley 1 22
- eight-puzzle 1 114, 125; 2 131-3
- Elcock 1 75; 2 75
- electrical networks 1 6
- Elliott 4100 2 119, 194
- Elliott 4120 2 125
- English Electric Leo Marconi 2 38
- English language research unit 2 173
- Ernest 2 119
- Euclid 2 58
- Euwe 2 99
- Evans, A. 2 21
- Evans, E. F. 1 212
- Evans, L. 2 108
- experimental programming unit 2 75, 194
- expressions and statements 2 224
- Fant 1 207
- Feldman 1 20
- Ferranti mercury 2 159

INDEX

- Feys 2 4
- FIDE 2 115
- fifteen-puzzle / 114, 125; 2 131, 132
- Fine 2 99
- flow charts / 22, 27
- formula Algol 2 210
- Fortran / 229; 2 33, 34, 43, 154, 185
- Fortran II 2 145
- Foster 2 43, 195
- Fréchet 2 105
- Fromkin / 209
- frozen formals 2 221
- frozen value list 2 222
- Fry / 200
- FSL / 20
- full strips and character strips 2 239
- functions 2 241

- Gaku / 106
- galactic emperor 2 110, 112
- game learning engine 2 140
- game-playing / 27, 68; 2 89
- garbage collection 2 235
- Gelernter / 36
- general regression program 2 155
- geometry machine / 36
- Gibson / 173
- Gillis 2 96
- Gilmore 2 61
- Gilmore example 2 62, 63
- GLEE 2 140, 142, 144, 152
- GMF 2 59, 60
- Go 2 91, 95, 100
- Golomb / 66
- Golombek 2 93
- Go-moku / 87; 2 75, 95
- Good 2 91, 93-7, 103, 105, 108, 110, 113-15
- Goon 2 244
- grammars, BNF: / 21; context free / 21; linguistic / 182
- graphs: similar / 4; linear / 3, 39; connected / 3; flat or planar / 4
- graph traverser / 41, 108; 2 119-22, 124-6, 128, 129, 131-4
- group theory / 32, 48
- Guard 2 59

- Halle / 207
- Hammond 2 95, 97
- Hampstead chess club 2 99
- Harary / 4
- Hayes / 113
- Hein / 174
- Held, M. 2 125
- Held, R. / 174
- Hellman 2 107
- Herbrand procedures / 49
- Herbrand theorem 2 58
- heuristic search principles / 48
- Hillix / 222

- Histmeyr 163
- Hodgson 2 47
- Holmes / 210
- Holt / 13
- Horowitz 2 99
- Hubel / 195
- Hultzen / 210
- Hunt 2 33

- Ianov schemata 2 34
- IBM 7090 2 107
- IBM 7094 II 2 125
- IBM Corp. 2 35
- identifiers 2 216
- immediate success / 70
- imperative expression 2 227
- imperative sequence 2 227
- incidence matrix / 7
- induction / 122
- Ingemann / 207
- Institut für instrumentelle mathematik 2 57
- International Chess Federation 2 115
- IPL5 2 38
- ISWIM 2 5-11, 13-16, 17-18, 210

- Jacobsen / 207
- Jacopini 2 30
- Jenkins / 156
- Johnson / 126
- Joss 2 185, 210

- Kaplow 2 185
- Karp 2 125
- KDF 9 2 174
- Kendall 2 153
- Kersta / 210
- Kidsgrove ALGOL 2 38
- Kister 2 97
- Knight / 76
- Kramer 2 99
- Kriegspiel 2 105
- Kufteg 2 102
- Kuno / 269
- Kunz 2 181

- labels and goto statements 2 227
- lambda expression 2 219
- Landau 2 105
- Landin / 274; 2 4-6, 9, 12, 34
- Laplace 2 141
- Lasker 2 95, 99, 100
- Lettvin / 172
- level of aspiration 2 151
- LISP 2 6, 12, 14, 34, 35, 38, 185, 189, 209, 210
- Lin 2 123-5, 127-31
- linear programming 2 123
- List-Algol 2 12, 13, 15, 16
- Lomnicki / 69
- London / 21

INDEX

- Logemann / 49
- loops 2 9
- Los Alamos 2 97
- Loveland / 49
- Luce 2 92
- McCarthy / 22; 2 22, 34, 38
 - MacLane / 32
 - McCulloch / 172; 2 94
 - Macdonald 2 89
 - macros 2 191, 224
 - MAD 2 185
 - MAP 2 185
 - Marbe's law / 158
 - matchbox engine 2 137
 - mathematical programming / 6
 - matrix storage / 233
 - Matthews 2 93, 95
 - Maturana / 172
 - Maynard Smith 2 96
 - Meltzer 2 70
 - menace 2 139, 140
 - mercury autocode 2 159
 - Messich / 162
 - meta functions 2 35
 - Michie / 113; 2 96, 106, 119, 120, 123, 125, 131, 133, 137, 140, 145
 - Miller / 196, 199
 - MIT 2 93
 - modern chess openings 2 109, 115
 - monkey puzzle sort / 9
 - Monte Carlo estimation problems 2 108
 - Mooers 2 210
 - Moray / 209
 - Morgenstern 2 92, 105
 - Mosteller 2 98
 - Mott-Smith 2 99
 - multics 2 38
 - Murray 2 75
 - Nealey 2 106, 107
 - network design / 70
 - Newcomb / 216
 - Newell 2 38, 98, 102, 103, 113, 119
 - Newell / 47, 106
 - Nim 2 142, 143
 - nodes / 3, 4
 - nonlocal variables 2 221
 - Norman / 4
 - notation for syntactic description 2 212
 - n*-tuple 2 220
 - noughts and crosses 2 140, 142, 144
 - Oetiger / 269
 - O'Donovan 2 99
 - Oldbury 2 107
 - optimal flow / 6
 - pandemonium 2 139
 - Panzerspiel 2 105
 - parenthesis counting rules / 22
 - partial application 2 221
 - PAT / 207
 - PCF 2 27-9
 - permutations, storage / 130
 - PG1 2 57, 61-70
 - Philidor 2 91
 - picture processing / 184
 - Pitts / 172
 - Poggi 2 70
 - Pole / 113
 - Polya / 4
 - Pop-1 2 134, 185, 189, 191, 194
 - Pop-2 2 209, 210, 213, 216-19, 237, 242-5
 - Popplestone 2 43, 134, 185, 209
 - Prawitz / 50; 2 59
 - precedence 2 225
 - predicates 2 35
 - Pride 2 181
 - Prinz / 4
 - probabilistic decision boxes 2 139
 - procedures 2 10
 - production language / 20
 - program(s): equivalent / 24; illegal / 21; legal / 21
 - program-user interaction / 127
 - proper cycle 2 26
 - quasi compound expression 2 228
 - Quine 2 64
 - Radoport / 162
 - Raiffa 2 92
 - randomized chess 2 115
 - Raphael / 107, 160
 - recognition processor 2 85
 - records 2 233
 - recursion: induction / 23; problems of / 263
 - Reed 2 23
 - ref-Algol 2 18
 - resolution principle / 49, 50
 - reverse polish / 6, 20, 185, 273
 - Riordan / 15
 - Roberts 2 38
 - Robinson, G. A. 2 57
 - Robinson, J. A. / 31, 48; 2 57, 60, 70
 - Roe 2 123, 126
 - Rosen 2 53
 - Rosenbloom 2 4
 - Rutledge 2 34
 - SAINT / 106
 - Samuel / 39, 91; 2 90, 96, 104, 106, 107
 - Samuel's draughts program 2 106
 - Schaefer 2 145
 - Schofield / 113; 2 131
 - Scoins 2 34
 - second-order compiler / 245
 - selectors 2 35
 - Selfridge 2 139

INDEX

- semantics 2 3
- Seshu 2 23
- Shannon 2 96, 109, 113
- Shaw 1 47, 106; 2 98 185
- shotgun 2 105
- Simon 1 47, 106; 2 98, 102, 103
- Slater 2 97
- Smith, D. L. 2 36
- Smith, F. W. 2 145
- source items 2 228
- source mode 1 251
- sparse matrix 2 38
- SRP 2 67, 68
- standard function 2 218
- standard variable 2 218
- state signals 2 147
- Steinitz 2 95
- STELLA 1 219
- Story 1 126
- Strachey 2 4, 19, 20, 35, 210, 221
- strips 2 234
- Strong 2 185
- Stuart 2 153
- subgoals 2 81
- successor family 1 70
- Suetin 2 99
- Sworder 2 145
- syntax 2 3

- Taylor 2 110
- Term 1 32
- Thorne 2 173
- tic-tac-toe 2 42
- TRAC 2 210
- transformation algorithms 2 198
- transformations 2 23

- transportation 1 6
- travelling salesman 1 6; 2 123, 125, 126, 131, 133, 134
- trees: co-tree 1 7; decision 1 66; derivative 1 13; multi-furcating 1 12; non-ordered 1 8; ordered rooted binary 1 12; ordered-rooted 1 9; ordered 1 5; planted 1 4; spanning 1 7; static 1 119; sub-1 10; rooted 1 4
- Turing 1 48; 96 113

- Vigor 2 33, 40, 134
- von Neumann 2 92, 105
- Vukovic 2 113

- Wang 1 39, 49; 2 64, 68
- Westervelt 2 154, 155
- Whitfield 1 212
- Widrow 2 145
- Wiesel 1 195
- Wirth-Hoare Algol 2 210
- Wooldridge 2 145
- word-endings 2 175
- word-recognition 1 160
- word-selection 1 160
- words 2 240
- WORSE 1 161
- Wos 2 57, 61, 64
- Wylie 2 96

- XTRAN 2 3

- Yngve 1 211

- Zugzwang 2 115

