20

LOGLISP: an alternative to PROLOG

J. A. Robinson and E. E. Sibert
School of Computer and Information Science
Syracuse University, USA

1. INTRODUCTION

Seven years or so after it was first proposed (Kowalski 1974), the technique of 'logic programming' today has an enthusiastic band of users and an increasingly impressive record of applications.

For most of these people, logic progamming means PROLOG, the system defined and originally implemented by the Marseille group (Roussel 1975). PROLOG has since been implemented in several other places, most notably at Edinburgh (Warren *et al.* 1977). Much of the rapid success of logic progamming is due to these implementations of PROLOG (as well as to the inspired missionary work of Kowalski, van Emden, Clark and others). The Edinburgh PROLOG system is in particular a superb piece of software engineering which allows the logic progammer to compile assertions into DEC-10 machine code and thus run logic programs with an efficiency which compares favourably with that of compiled LISP. All other implementations of logic programming (including our own, which we describe in this paper) are based on interpreters.

However, PROLOG is not, and does not claim to be, the definitive logic programming formalism. It has features which (however useful in practical applications) are somewhat foreign to Kowalski's original simple and clear conception of a deductive, assertional programming language based entirely on the LUSH resolution of Horn clauses. In this conception there are no imperative constructs and indeed there is no notion of control or of anything 'happening'. The spirit is very like that of Strachey's static Platonic vision – elaborated by Landin and later by Scott – of programs as purely denotative expressions in which entities are represented as the results of applicative combinations of functions with their arguments. Running a program is, on this view, nothing more than finding out what it denotes. The program merely does the denoting, and should not be taken for a prescription of any sort of activity, such as 'computing'.

The Strachey-Landin-Scott doctrine is of course quite compatible with the belief that one can make a machine which when given a program as input will systematically work out and deliver a description of its denotation. Landin's classic interpreter – the SECD machine – is the very essence of such a device. We can say that the computation of the machine is correct (if it is) only because we have a machine-independent notion of what the program is intended to denote.

Most of the PROLOG programs known to us appear to be motivated by the desire to make the machine behave in a certain way rather than (or, as well as) merely to obtain answers to queries. We have been saddened by the use made in applications of the (highly effective!) 'CUT' control construct. For example, the assertions

(NOT p) ← p CUT FAIL
(NOT p) ←

together (and in the order given) define NOT in the sense of 'negation as failure'. In attempting to establish (NOT p) by appeal to the first assertion, the PROLOG machine first seeks to establish p. If it succeds in establishing p, it goes on to perform the CUT so as to preclude the (incorrect) establishment of (NOT p) by the second procedure, and then (correctly) returns a failure to the caller of (NOT p). If, however, it fails to establish p, it immediately drops through to the second assertion and thereby (correctly) establishes (NOT p).

Because it appears to encourage this sort of programming, and because it has rapidly evolved into a workhorse programming language serving a busy user community not given to contemplating Platonic Forms, PROLOG may be turning into the FORTRAN of logic programming – as McDermott recently observed (McDermott 1980).

Our own early attempts (as devoted users of LISP) to use PROLOG convinced us that it would be worth the effort to create *within LISP* a faithful implementation of Kowalski's logic programming idea. We felt it would be very convenient to be able to set up a knowledge base of assertions inside a LISP workspace, and to compute the answers to queries simply by executing appropriate function calls. What seemed to be required was an extension of LISP consisting of a group of new primitives designed to support unification, LUSH resolution, and so on, as efficiently as possible. We set out to honor the principle of the separation of logic from control (no CUT, no preferred ordering of assertions within procedures nor of atomic sentences within hypotheses of assertions) by making the logic programming engine 'purely denotative'. Instead of the PROLOG method of generating the LUSH resolution proof tree one branch at a time by backtracking, we decided to generate all branches in quasi-parallel so that heuristic control techniques could be brought to bear on deciding which branch to develop at each step and so that our design would lend itself to multiprocessing at a later stage.

In the next section we give a summary description of the resulting system, which we call LOGLISP. The reader is encouraged to think of LOGLISP as LISP +

LOGIC, where LISP is what it always has been and LOGIC is the collection of new primitives which we have added to LISP.

2 LOGIC

In LOGIC we represent assertions, queries, and all other other logic-programming constructs as LISP data-objects.

2.1 Variables And Proper Names

Logical variables are represented as identifiers which begin with a lower-case letter. Proper names (i.e., individual constants, predicates, and operators) are represented as identifiers which begin with an upper-case letter or a special character such as +, *, %, =, <, >, or $. In addition, numerals and strings can be used as individual constants (but not as predicates or operators). We refer to proper names which are not numerals or strings as *proper identifiers.*

2.2 Terms

A term is either a logical variable, an individual constant, or an operator-operand combination represented as a dotted pair (F. S) whose head F is the operator and whose tail S is the operand. The operand is a list (possibly empty) of items.

2.3 Predication

A predication (= atomic sentence) is a predicate-subject combination also represented as a dotted pair. The predicate is the head of the pair, and the subject is the tail. The subject is a list (possibly empty) of terms.

Thus in LOGIC the external syntax of terms and predications is the customary LISP notation for atoms and lists. All applicative combinations are written as lists in which the head is applied to the tail. For example, we have:

(Age John (+ 3 (* 4 5)))

instead of

Age(John, +(3, *(4, 5))) .

2.4 Assertions, Procedures And Knowledge Bases

We call sentences of the form

if each member of A is true then B is true

assertions. In general the *conclusion* B is a predication while the *hypothesis* A is a list (possibly empty) of predications.

An assertion with an empty hypothesis is *unconditional,* while one whose hypothesis is nonempty is *conditional.*

An assertion in which one or more logical variables occur is called a *rule,* while one in which no logical variables occur is known as a *datum.*

A rule is to be understood as if its logical variables were all governed by universal quantifiers. Thus, assuming the rule contains the logical variables $x1, \ldots, xt$ and no others, we read it as

for all $x1, \ldots, xt$:
if each member of A is true then B is true.

Of course, if some of these variables (say, $z1, \ldots, zs$) occur only in A while the rest (say, $v1, \ldots, vr$) occur also or only in B, then by elementary logic we can read the assertion as:

for all $v1, \ldots, vr$:
if there exist $z1, \ldots, zs$ such that
each member if A is true
then B is true.

We follow Kowalski's convention of writing the conditional assertion

if each member of (A1 ... A*n*) is true
then B is true

in the reverse-arrow notation:

B ← A1 ... A*n* .

In this notation, the unconditional assertion

if each member of ( ) is true then B is true

becomes

B ← .

An assertion thus may be (1) a conditional rule, (2) an unconditional datum, (3) an unconditional rule, or (4), a conditional datum. Assertions of types (3) and (4) are rarely encountered in knowledge bases arising in applications.

A knowledge base is set of assertions. It is customary to think of a knowledge base as partitioned into *procedures*, each procedure corresponding to some predicate P and consisting of all the assertions in the knowledge base whose conclusion has *P* as its head. The name of the procedure is then by convention taken to be P.

Assertions may also be given their own individual names at the option of the user. In displaying explanations (= deductions) it is often convenient to refer to an assertion by its name instead of displaying the assertion itself.

2.5 Entering Assertions Into A LOGIC Knowledge Base

There are several ways of entering assertions into a LOGIC knowledge base.

The command

(⊢ B A1 ... A*n*)

causes the assertion

B ← A1 ... A*n*

to be added to the current knowledge base. The two-character symbol ⊢ is pronounced "assert" (some prefer "turnstile") and is the nearest we can approach with the ASCII character set to the logician's assertion sign.

It is also possible to place LOGLISP temporarily into a special input mode called the FACTS mode, in which successive assertions can be entered one after the other. The command (FACTS) causes the FACTS mode to be entered, and the message ASSERT: is then repeatedly given as a prompt for the next assertion to be typed in. When typing in the assertion B ← A1 ... A*n* to the FACTS mode, we represent it simply as the list (B A1 ... A*n*).

In either way of entering an assertion, one can attach a name *N* to it simply by putting N before the conclusion. Thus:

(⊢ N B A1 ... A*n*)

or, in FACTS mode entries,

(N B A1 ... A*n*) .

The name N can be any proper identifier.

The assert command ⊢ is an FEXPR (i.e., a LISP function which does not evaluate its argument expressions when called) and hence a little awkward to use in entering assertions other than from the terminal. For internal use the LOGIC programmer would find the corresponding EXPR function ASSERTCLS (which *does* evaluate its argument expressions) more convenient.

Finally, an entire knowledge base whose name is *N* can be read into the LOGLISP workspace from the disk by either (RESTORE *N*) – which first clears the workspace of any assertions which may be present – or (LOADLOGIC N) – which simply adds the assertions of *N* to those already present in the workspace.

Such a knowledge base can be named N and stored on the disk by the command (SAVE N), which takes all assertions currently in the workspace, creates a file named N which contains them all, and writes it out onto the disk.

Various commands are provided for displaying the contents of a knowledge base currently in the workspace. Typing (PRINTFACTS) causes the entire knowledge base to be displayed, its assertions grouped into procedures. Typing (PRINTFACTSOF *P*1 ... *Pn*) displays just the assertions comprising the procedures *P*1, ..., *Pn*.

LOGIC maintains an indexing scheme (which is automatic) whereby each proper identifier M has on its property list a complete record of all assertions in whose conclusion M occurs. This scheme is used by the internal processes of the LOGIC system during the deduction cycle, as will be explained below. It also makes it simple to respond to the command (PRINTCREFSOF M) which causes the collection of those assertions to be displayed.

The command (PRLENGTH P) gives the number of assertions currently in the procedure P. The command (PREDICATES) gives the list of all predicates – i.e., predicates – currently in the workspace.

Since assertions are LISP data-objects they may be edited with the help of the LISP resident Editor. In LOGIC there is a command EDITA (by analogy with

LISP's own EDITP, EDITV, EDITF, etc.) which calls the LISP Editor and supervises the editing session. In particular after entering the Editor with (EDITA *P*) or (EDITA (P1 ... P*n*)) – depending on whether one wishes to edit the single procedure P or several procedures P1, ..., P*n* at once – the ensuing exit command OK causes LOGIC to re-enter the (in general) modified assertions into the knowledge base just as though they had all been erased from it an were now being entered by the user for the first time.

This 'enhanced OK' is quite transparent to the user, who can use the Editor in exactly the same way for LOGIC objects as for LISP objects.

If the user wishes to erase a procedure P from the workspace, the command (ERASEP P) causes this to be done.

2.6 Queries

A query is essentially a description

$$\{(x1 \ldots xt) \mid P1 \,\&\, \ldots \,\&\, Pn\}$$

of the set of all tuples $(x1 \ldots xt)$ which satisfy some *constraint* P1 & ... & P*n* which is a conjunction of predications *Pi*.

In LOGIC such a query is written

(ALL $(x1 \ldots xt)$ P1 ... P*n*)

and is a call on the function ALL (which is a FEXPR). It returns as its (LISP) value the list of all tuples satisfying the given constraint. This list is called the *answer* to the query. The component tuples of the answer are obtained by means of the basic *deduction cycle*, explained below, which is the heart of LOGIC and which is invoked by every query.

Useful variations of the basic query construct are

(ANY K $(x1 \ldots xt)$ P1 ... P*n*)

which returns no more than K of the set of tuples satisfying the constraint,

(THE $(x1 \ldots xt)$ P1 ... P*n*)

which returns the sole member of the list (ANY 1 $(x1 \ldots xt)$ P1 ... P*n*) (rather than the list itself), and

(SETOF K X P)

which is the EXPR underlying the FEXPRs ALL, ANY and THE. SETOF takes three arguments: K, which should evaluate to an integer or the atom ALL; X, which should evaluate to an 'answer template' $(x1 \ldots xt)$; and P, which should evaluate to a 'constraint list' (P1 ... P*n*). It is particularly to be noted that the answer to a query is a LISP data object, and thus can be subjected under program control to internal analysis and minipulation, as well as being displayable at the teminal.

2.7 The Deduction Cycle

The basic process of the LOGIC system is that carried out in the deduction cycle to compute the answer to a given query.

2.7.1 Implicit Expressions – For the sake of both clarity and efficiency the representation of constraints during the deduction cycle is done by the Boyer-Moore technique. This calls for an expression C to be represented by a pair (Q E) called an *implicit expression*, in which Q is an expression known as the *skeleton part* and E is a set of variable-bindings known as the *environment part* of the implicit expression.

The idea is that the implicit expression (Q E) represents the expression which is the result if installing throughout Q the bindings given in E.

2.7.2 Bindings And Environments – In general a *binding* is a dotted pair whose head is a variable, while an *environment* is a list of bindings no two of which have the same head.

2.7.3 Immediate And Ultimate Associates – If the environment E contains the binding A.B we say that *A is defined in E* and write: (DEFINED A E). We also say that B is *the immediate associate of A in E*, and we write: B = (IMM A E).

The immediate associate of A in E might turn out to be a variable which is itself defined in E. When this is the case we often wish to track down *the ultimate associate of A in E*, namely, the first expression in the series:

A, (IMM A E), (IMM (IMM A E) E), . . . ,

which is *not* itself defined in E. This expression is given by:

(ULT A E) = if (DEFINED A E) then (ULT (IMM A E) E) else A

We can now say more precisely how an expression is represented implicitly by a skeleton-environment pair.

2.7.4 Recursive Realizations – First, we define the function RECREAL (for "recursive realization") which takes as arguments an expression X and an environment Y, as follows:

(RECREAL X Y) =
    if (CONSP X) then (RECREAL hX Y).(RECREAL *t*X Y) else
    if (DEFINED X Y) then (RECREAL (ULT X Y) Y) else X

where (CONSP X) is true if X is a dotted pair and false otherwise.

The skeleton-environment pair (Q E) then represents the expression: (RECREAL Q E).

2.7.5 Unification – The deduction cycle involves repeated use of the operation of LUSH resolution (Hill 1974), which in turn involves the unification process. Let us first discuss these two important ideas and then go on to define the deduction cycle.

Given two expressions A and B together with an environment E, we say that *A unifies with B in E* if there is an extension E′ of E such that

(RECREAL A E′) = (RECREAL B E′) .

We define the function UNIFY in such a way that if A unifies with B in E then (UNIFY A B E) is the most general extension E′ of E satisfying the above equation. If A does not unify with B in E then (UNIFY A B E) is the message "IMPOSSIBLE".

UNIFY is defined by:

(UNIFY A B E) =
if E is "IMPOSSIBLE"
then "IMPOSSIBLE"
else (EQUATE (ULT A E) (ULT B E) E)

where

(EQUATE A B E) =

if A is B	then E
else if A is a variable	then (A.B).E
else if B is a variable	then (B.A).E
else if not (CONSP A)	then "IMPOSSIBLE"
else if not (CONSP B)	then "IMPOSSIBLE"
else (UNIFY *t*A *t*B (UNIFY *h*A *h*B E)) .	

2.8 LUSH Resolution

Now suppose that we have a knowledge base D and a constraint represented by the skeleton-environment pair (Q E).

Let (VARIANT Q E D) be a variant of D having no variables in common with those in Q or in E. A variant of D is an object exactly like D with the possible exception that different identifiers are used for some or all of its variables.

Let (SELECT Q E D) be a positive integer no longer than the length of the list Q. SELECT is supposed to be able to use the content and structure of Q, E and D, if need be, to determine its result.

Finally, given a nonempty list X and a positive integer K no larger than the length of X, we speak of *splitting* X with respect to its Kth component, and we define (SPLIT X K) to be the triple (L A R) such that A is the Kth component of X and X = L*(A)*R.

(The concatenation of lists L and M is denoted by L*M, and * is of course associative).

Thus if (L A R) is (SPLIT X K) then L is the list of the first K-1 components of X and R is the list of the last ((LENGTH X) – K) components of X. In particular, when K = 1, we have

L = ()
A = *h*X
R = *t*X

With these auxiliary ideas we can now define the LUSH resolvents of an implicit constraint (Q E) with respect to a knowledge base D. They are all the implicit constraints of the form

(L*H*R (UNIFY A C E))

such that:

(1) H is the hypothesis, and C the conclusion, of an assertion in (VARIANT Q E D)
(2) (L A R) is (SPLIT Q (SELECT Q E D))
(3) A unifies with C in E.

At present the LOGLISP system uses (SELECT Q E D) = 1 for all Q, E and D. More general selection criteria permitted by the theory of LUSH resolution (Hill 1974) are currently under study, and future versions of LOGLISP may well adopt a more complex SELECT.

The 'separation of variables' accomplished by taking (VARIANT Q E D) instead of D is managed quite efficiently in LOGLISP, as indeed in all of the various PROLOG implementations known to us. It is theoretically necessary to take variants, to preserve the completeness of the resolution inference principle.

In LOGLISP the set of LUSH resolvents of (Q E) with respect to D is returned as a list (RES Q E D) which shares much of its structure with that of (Q E). The basic technique is that of Boyer and Moore.

In computing (RES Q E D) LOGIC often does not have to search the entire procedure whose predicate is the same as that of the selected predication A. This procedure may have the property that each of its assertions has a conclusion containing no variables. In this case, it is enough to try just those assertions whose conclusion actually contains every proper identifier which appears in A. These assertions are readily retrieved via the secondary indexing scheme mentioned earlier.

2.9 Definition Of The Deduction Cycle

We are now in a position to give the definition of LOGLISP's deduction cycle.

The answer to the query (SETOF K X P) is the result of the following algorithm:

```
IN:     let SOLVED be the empty set and
        let WAITING be the set containing only (P ( ))
RUN:    while  WAITING is nonempty and
               SOLVED has fewer than K members
        do 1   remove some implicit constraint C from
               WAITING and let (Q E) be (SIMPLER C D)
           2   if    Q is ( )
               then  add E to SOLVED
               else  add the members of (RES Q E D) to WAITING
OUT: return (SIMPSET X SOLVED)
```

Remarks

(1) The functions SIMPLER and SIMPSET are discussed below in the section dealing with LISP-simplification.

(2) If K is "ALL" then the test in RUN depends only on its first conjunct.

(3) The query (ALL X P1 ... P*n*) is the same as the query

(SETOF (QUOTE ALL) (QUOTE X) (QUOTE (P1 ... P*n*))))

and the query (ANY K X P1 ... P*n*) is the same as

(SETOF K (QUOTE X) (QUOTE (P1 ... P*n*)))

while the query (THE X P1 ... P*n*) is the same as

(CAR (SETOF 1 (QUOTE X) (QUOTE (P1 ... P*n*))))

(4) The 'answer template' X in a query can be a variable, or a proper name, or a list of expressions (in particular, a list of variables).

(5) (RES Q E D) can be empty. In this case the net effect is just to drop the constraint C from WAITING and to add nothing to SOLVED. C is thus an 'immediate failure' (see below).

(6) The selection of C from WAITING is made from among those constraints whose estimated 'solution cost' is least. This estimate is a linear combination of the length of C's skeleton part and the length of the deduction which produced C.

2.10 The Deduction Tree. Immediate And Ultimate Failures

Examination of the deduction cycle reveals that it is growing a tree (the 'deduction tree') whose nodes are implicit constraints. The root of the tree is the constraint of the original query, implicitly represented. The successors of a node are its LUSH resolvents.

During the growth process the fringe of the growing deduction tree consists of the nodes in WAITING together with those whose environment parts have been added to SOLVED.

The tips of the completed deduction tree fall into two classes. Those whose skeleton part is the empty list are *successes*, and their environment parts will have been added to the (now also complete) collection SOLVED. Those whose skeleton part is nonempty are *failures*, and they will have contributed nothing to SOLVED. These failures are known as *immediate*, in contrast to *ultimate*, failures. An immediate failure simply has no resolvents – none of the assertions in the knowledge base has a conclusion which unifies with its selected predication.

An ultimate failure is a node which, while not a tip of the completed deduction tree, has no descendants which are successes. All its descendants are failures. It would be splendid to be able to detect an ultimate failure other than by developing the entire subtree of its descendants and finding that all its tips are immediate failures.

2.11 LISP Simplification And The Functions SIMPLER, SIMPSET

In the deduction cycle the functions SIMPLER and SIMPSET make essential use of the LISP-simplification process.

Intuitively, what the LISP-simplification process does is to replace an expression by its *reduction* (if it has one) according to the LISP meanings (if any) which it and its subexpressions may have. For example, the expression (+ 3 4) can be reduced to the expression 7, and the expression (LESSP (ADD1 5) (TIMES 2 8)) can be reduced (in three steps) to T. The reduction of an expression is the result of persistently replacing its subexpressions by their definitional equivalents, in the manner of computation, until no further replacements are possible. Reduction is not always the same as evaluation. For example (+ *a* (+ 2 2)) reduces to (+ *a* 4). Nor can every expression be reduced – some are irreducible. So we speak in general of the *LISP-simplification* of an expression, and define this to be the expression itself, if it is irreducible, or else the (irreducible) expression which results from reducing it as far as possible.

2.11.1 SIMPSET – The set returned by the deduction cycle is computed (and represented as a list) by the function SIMPSET from the list SOLVED of environments and the answer template X of the query which activated the cycle.

SIMPSET computes the set of all expressions which are the LISP-simplifications of (RECREAL X E) for some E in SOLVED.

2.11.2 SIMPLER – The function SIMPLER transforms the implicit constraint C selected in step 1 of the RUN loop in the deduction cycle.

Remember that if C is (Q E) we are really dealing with the list P = (RECREAL Q E) which C implicitly represents.

SIMPLER's job, intuitively, is to replace the (SELECT Q E D)th predication in P by its LISP-simplification. Sometimes the (SELECT Q E D)th predication in P reduces to T, and in this case SIMPLER deletes it entirely from P (since P is representing the conjunction of its components, the resulting list P′ is equivalent to P). If Q′ is the result of deleting the (SELECT Q E D)th predication in Q from Q, we will then have that P′ is (RECREAL Q′ E). SIMPLER then repeats the whole thing again, reducing the (SELECT Q′ E D)th predication in P′, and so on until either the list of predications is empty or the selected predication does not reduce to T. The output of SIMPLER is then the implicit constraint representing (with environment part E) this final list of predications.

The practical consequence of this SIMPLER transformation is to allow the user to invoke very nearly the full power of LISP from within the hypotheses of LOGIC assertions. Not only are all the LISP primitives given their usual meanings, but any identifiers the user may have (in the usual LISP manner) defined as functions, or given values as (LISP) variables, are accorded these meanings during LISP simplification.

An immediate advantage of this feature is that LOGIC queries – which are LISP calls on the LISP functions ALL, ANY, THE, or SETOF – can be made

within the deductions going on at a higher level. Queries can invoke sub-queries and so on to any depth.

A simple illustration of this is the assertion defining NOT (which of course is *also* a LISP primitive) in the sense of 'negation as failure':

(NOT *p*) ← (NULL(ANY 1 T *p*)) .

This says that in order to establish a predication of the form (NOT *p*) it is sufficient to run the query

(ANY 1 T *p*)

and to find that it returns the empty list as answer. Note that the possible answers to (ANY 1 T *p*) are ( ) and (T). The answer (*T*) would mean that at least one way was found of proving *p*; the answer ( ) that not even one way of proving *p* was found, despite a complete search.

Thus if one is willing to assume of one's knowledge base that inability to prove a predication is tantamount to the ability to prove its negation, one may appropriately add to the knowledge base the above assertion as expressing this postulate of completeness.

2.12 Infinite Searches. The Deduction Window

Since in general there may be infinitely many components in the answer to a query, some way must be provided of gracefully truncating, after only finitely many components have been found, the otherwise infinite process of computing such an answer.

In LOGLISP a collection of parameters is provided (whose values may be set by the user or left at their default settings) which bound the size of the deduction tree in various ways. For example, the total number of constraints (= nodes) may be bounded, as may the maximum branch length, and the size of constraint lists within nodes. It is also possible to limit the number of times in any one branch that rules (as opposed to data) are applied.

The set of these bounds is called the *deduction window.*

It is worth pointing out that the deduction window can be set up for each activation of the deduction cycle simply by annotating the query appropriately. For example, the unadorned query (ALL X P1 ... P*n*) would be run with the default window, but the same query, annotated as

(ALL X P1 ... P*n* RULES: 5 TREESIZE: 1000)

would be run with bounds of 5 and 1000, respectively, in place of the default bounds.

In addition to controlling the shape and extent of the deduction tree the user may specify the coefficients used in computing the 'solution cost' of each constraint added to WAITING in the deduction cycle. Since the constraint selected from WAITING in step 1 of RUN is always one of those having least solution cost, this gives the user some control over the manner in which the deduction tree is developed.

It is also possible to specify that the deduction cycle be conducted in quasi-PROLOG style, that is, depth-first and with assertions taken strictly in the order that the user entered them.

2.13 Deduction Running Times

The running times achieved by LOGIC in answering queries are not as impressive as those of, say, the Edinburgh PROLOG system using compiled assertions. In LOGIC the assertions are, in effect, interpreted. In testing LOGIC with a variety of examples we have found that the deduction tree grows at rates of 50 or 60 nodes per second for most examples. We are currently studying ways of compiling assertions in the manner of Warren, and looking for other means of improving the running time of the deduction cycle.

The following extended example shows some of the main features of LOGLISP at work.

3. PLACES – AN 'INTELLIGENT' DATABASE

PLACES is a knowledge base containing several thousand assertions most of which are *data*, i.e., unconditional ground assertions.

Some representative data of PLACES are shown in Fig. 1.

```
(POPULATION BURMA 32200000) ←
(LATITUDE WARSAW 52.25) ←
(LONGITUDE PYONG-YANG −125.8) ←
(ADJOINS LAOS VIETNAM) ←
(COUNTRY VIENNA AUSTRIA) ←
(PRODUCES USSR OIL 491.0 1975) ←
(BELONGS IRAN OPEC) ←
(REGION ISRAEL MIDDLE-EAST) ←
(AREA ETHIOPIA 471778) ←
(GNP-PER-CAPITA NEW-ZEALAND 4250) ←
(OPEN-WATER BALTIC-SEA) ←
(NARROW DARDANELLES) ←
```

Fig. 1.

For each predicate appearing in Fig. 1, PLACES has a collection of such unconditional ground assertions – a *data procedure.* All these data procedures are comprehensive (they average several hundred assertions each) and some are in a sense complete.

The procedures POPULATION, AREA, REGION, GNP-PER-CAPITA are complete in the sense that every country in the world is covered.

The GNP-PER-CAPITA procedure gives (in US dollars) the gnp-per-capita for each country in the world for a particular year (1976).

The procedure ADJOINS provides data for a procedure BORDERS, which is a pair of rules:

(BORDERS x y) ← (ADJOINS x y)
(BORDERS x y) ← (ADJOINS y x)

which give PLACES the ability to determine which countries (or bodies of open

water) border upon which others. Since ADJOINS is a symmetric relation we need not assert it in both directions, and BORDERS uses ADJOINS accordingly.

The procedure PRODUCES gives (in millions of metric tons) the quantities of various basic commodities (oil, steel, wheat, rice) produced by most of the world's countries in two particular years (1970 and 1975). This procedure could well have covered more years and more commodities, but for the purposes of an example a few hundred assertions seemed enough to illustrate the possibilities.

While the countries of the world form (at any given time) a rather definite set, it is less clear what are the bodies of water which should be named and treated as entities in a database such as PLACES. We took the abritrary course of naming those bodies of water found on the maps of various parts of the world in the Rand McNally *Cosmopolitan World Atlas*. We ignored those bodies of water which seemed too small to be of much significance but we strove for some sort of comprehensive description of the boundary of each country. For example, the query

(ALL *x* (BORDERS *x* IRAN))

gets the answer

(STRAITS-OF-HORMUZ GULF-OF-OMAN TURKEY USSR PAKISTAN IRAQ CASPIAN-SEA AFGHANISTAN PERSIAN-GULF)

in which each of the bodies of water STRAITS-OF-HORMUZ, GULF-OF-OMAN, CASPIAN-SEA and PERSIAN-GULF is listed as having a portion of its boundary in common with that of the country IRAN.

3.1 Rules

PLACES contains, in addition to these large 'data procedures', a number of rules defining predicates useful in formulating queries. For example there is a procedure DISTANCE, which consists of the following four rules:

(DISTANCE (POSITION *la1 lo1*) (POSITION *la2 lo2*) *d*)
←(= *d* (SPHDST *la1 lo1 la2 lo2*))
(DISTANCE (POSITION *la1 lo1*) (PLACE *q*) *d*)
←(LATITUDE *q la2*)
&(LONGITUDE *q lo2*)
&(= *d* (SPHDST *la1 lo1 la2 lo2*))
(DISTANCE (PLACE *p*) (POSITION *la2 lo2*) *d*)
←(LATITUDE *p la1*)
&(LONGITUDE *p lo1*)
&(= *d* (SPHDST *la1 lo1 la2 lo2*))
(DISTANCE (PLACE *p*) (PLACE *q*) *d*)
←(LATITUDE *p la1*)
&(LATITUDE *q la2*)
&(LONGITUDE *p lo1*)
&(LONGITUDE *q lo2*)
&(= *d* (SPHDST *la1 lo1 la2 lo2*))

This procedure can be used to obtain the great-circle distance between any two cities whose latitudes and longitudes are in the data tables, or between one such city and an arbitrary position on the earth's surface (given by its latitude and longitude) or between two such arbitrary positions.

The procedure DISTANCE illustrates the ability to call user-defined LISP functions by forming terms using their names as operators. The LISP function SPHDST returns the great circle distance (in nautical miles) between any two points on the earth's surface (given by their respective latitudes and longitudes).

Thus the query:

(THE *d* (DISTANCE (PLACE SAN-FRANCISCO) (PLACE OSLO) *d*))

gets the answer:

5197.5394

There is a rule which serves to define the predicate LANDLOCKED. Intuitively, a country or body of water is landlocked if it borders upon only land. The PLACES rule which formalizes this meaning is

(LANDLOCKED *x*)
← (IS-COUNTRY *x*)
& (NULL (ANY 1 T (BORDERS *x z*) (OPEN-WATER *z*)))

This rule contains two features worthy of comment.

The predicate IS-COUNTRY, defined by the rule

(IS-COUNTRY *x*)
← (COND ((VARIABLE (LISP *x*)) (COUNTRY *y x*))
((ANY 1 T (COUNTRY *z x*))))

shows how one can use to advantage the LISP conditional form within a LOGIC predication. The effect of the conditional is to avoid redundancy in proving that a given country is a country – by finding all the various cities in it – via a check to see if the argument *x* is variable or not. If it is not, then we need find only one datum from the COUNTRY data procedure which has the given country as its second argument.

The predicate VARIABLE holds just when its argument expression is a logical variable. The term (LISP *a*), for any expression *a*, evaluates to *a* itself (in this respect it behaves like (QUOTE *a*)). However, unlike (QUOTE *a*), (LISP *a*) is not a sealed-off context, and (RECREAL (LISP *a*) E) is (LISP (RECREAL *a* E)).

The second thing worth noting about the rule for LANDLOCKED is the embedded deduction. The list returned by the call

(ANY 1 T (BORDERS *x z*) (OPEN-WATER *z*))

will be empty if and only if *x* is landlocked.

A similarly structured rule defines the predicate DOMINATES. We wish to say that a country *x* dominates a 'narrow' waterway *y* if *x* borders *y* but no other country does. Thus:

(DOMINATES x y)
← (NARROW y)
& (IS-COUNTRY x)
& (BORDERS x y)
& (NULL (ANY 1 T (BORDERS y w)
(NOT (OPEN-WATER w))
(NOT (= x w))))

3.2 Negation As Failure

The use of the predicate NOT in the procedure DOMINATES again raises the interesting general topic of 'negation as failure'.

NOT is of course a LISP-defined notion and will therefore receive appropriate treatment during the deduction cycle in the manner explained earlier. However, it is possible to include in one's knowledge base the rule we discussed earlier.

(NOT p) ← (NULL (ANY 1 T p))

which is known as the 'negation as failure' rule. PLACES has the negation as failure rule as one of its assertions. The effect of its presence in a knowledge base is to declare that the knowledge base is complete – that inability to deduce p is to be treated as tantamount to the ability to deduce the negation of p.

The version of the negation as failure rule shown above is undiscriminating as between the various predications – it is in effect the declaration that all of the data procedures are complete and that all of the general procedures are 'definitions' of their predicates. It would be possible to assert more specialized negation as failure rules, which declare that the knowledge base is complete with respect to a particular predication-pattern. For example, we might assert

(NOT (BELONGS x y)) ← (NULL (ANY 1 T (BELONGS x y)))

in order to declare that BELONGS is complete, even though we are not willing to assert the negation as failure rule for all predications p. In general, one would expect that users of LOGIC would wish to be selective in their appeal to negation as failure, in just this fashion.

These data and rules are invoked by the following queries, which illustrate some of the possibilities.

3.3 Some Sample Queries For PLACES.

The following examples consist of some specimen queries which one can make of PLACES, together with the answers that they get. In each case we first state the query in ordinary English, and then restate it in formal LOGLISP.

We are not claiming that there is a uniform procedure, known to us, by which one may translate queries from English to LOGLISP in this manner. At present, in order to express queries (and indeed, assertions) in LOGLISP, one must know the language and be able to express one's intentions in it. In this respect LOGLISP is like any other programming language. It is in fact quite easy

to learn enough LOGLISP to construct and operate one's own 'intelligent database' in the style of PLACES.

Query 1.

What are the oil production figures for the
non-Arab OPEC countries in the year 1975?

```
(ALL (x y)
    (BELONGS x OPEC)
    (NOT (BELONGS x ARAB-LEAGUE))
    (PRODUCES x OIL y 1975.))
```

Answer 1.

```
((IRAN 267.59999) (NIGERIA 88.399991)
                  (VENEZUELA 122.19999)
                  (INDONESIA 64.100000)
                  (EQUADOR 8.2000000))
```

This answer is shown just as the LISP 'prettyprint' command SPRINT types it out. It is of course possible to dress up one's output in any way one pleases. Note that ALL returns a *list* of (in this case) tuples.

Query 2.

Of all the countries which are poorer than Turkey,
which two produced the most steel in the year 1975?
How much steel was that? What are the populations
of those countries?

```
(FIRST 2.
    (QUICKSORT
    (ALL (x y w)
        (GNP-PER-CAPITA TURKEY v)
        (GNP-PER-CAPITA x u)
        (LESSP u v)
        (PRODUCES x STEEL y 1975.)
        (POPULATION x w))
    (DECREASING)
    2.))
```

Answer 2.

```
((CHINA 29.0 880000000.) (INDIA 7.8999999 643000000.))
```

This example illustrates the fact that ALL (like ANY, THE, and SETOF) returns a LISP data-object which can be handed as an argument to a LISP function. In this case QUICKSORT and FIRST are user-defined LISP functions which were created in order to serve as useful tools in posing inquiries to PLACES. (LESSP is a standard LISP primitive).

(QUICKSORT *list relation k*) returns the given *list* of tuples ordered on the *k*th component with respect to the given *relation.*

(FIRST *n list*) returns the (list of the) first *n* components of the given *list.*

(DECREASING) returns the LISP relation GREATERP (and we also have (INCREASING), which returns the relation LESSP, and (ALPHABETICALLY), which returns the relation LEXORDER).

Query 3.

Which of France's neighbours produced most wheat (in metric tons) per capita in the year 1975? How much wheat per capita was that?

```
(EARLIEST
(ALL (x y
(ALL (x y)
    (BORDERS x FRANCE)
    (PRODUCES x WHEAT z 1975.)
    (POPULATION x u)
    (= y (QUOTIENT (TIMES z 1000000.) u)))
(DECREASING)
2.)
```

Answer 3.

```
(ITALY 0.16956329)
```

(EARLIEST *list relation k*) returns the first tuple in *list* after it has been re-ordered on the *k*th component of each of its tuples with respect to the given *relation.* Note that arithmetical terms formed with LISP's arithmetic operations are evaluated by the simplification step of the deduction cycle, as explained earlier.

One could have written % instead of QUOTIENT, and * instead of TIMES.

Query 4.

Which of the NATO countries is landlocked?

```
(ALL x (BELONGS x NATO) (LANDLOCKED x))
```

Answer 4.

```
(LUXEMBOURG)
```

Query 5.

Which waterway is dominated by Panama?

```
(THE x (DOMINATES PANAMA x))
```

Answer 5.

```
PANAMA-CANAL
```

Note that THE returns PANAMA-CANAL and not (PANAMA-CANAL).

Query 6.

Describe the boundary of the USSR by giving all its neighbours in alphabetical order.

```
(ORDER (ALL x (BORDERS x USSR)) (ALPHABETICALLY))
```

Answer 6.

```
(AFGHANISTAN ARCTIC-OCEAN BALTIC-SEA BERING-SEA
BLACK-SEA BULGARIA CHINA FINLAND HUNGARY IRAN
MONGOLIA NORWAY POLAND ROMANIA TURKEY)
```

(ORDER *list relation*) returns the given *list* after ordering it with respect to the given *relation.*

Query 7.

Are there any landlocked countries in the Far East? If so, give an example.

```
(ANY 1. x (REGION x FAR-EAST) (LANDLOCKED x))
```

Answer 7.

```
(MONGOLIA)
```

Query 8.

Is there an African country which dominates an international waterway? Which country? Which waterway?

```
(ANY 1. (x y) (REGION x AFRICA) (DOMINATES x y))
```

Answer 8.

```
((EGYPT SUEZ-CANAL))
```

Query 9.

What is the average distance from London of cities in countries which have a Mediterranean coastline and which are no more densely populated than Ireland? List those countries, together with their population densities, from least crowded to most crowded.

```
(PROGN (SETQ COUNTRIES-AND-DENSITIES
    (QUICKSORT
    (ALL (x x-density)
        (POPULATION IRELAND irish-population)
        (AREA IRELAND irish-area)
        (= irish-density
            (% irish-population irish-area))
```

```
                (BORDERS x MEDITERRANEAN-SEA)
                (NOT (OPEN-WATER x))
                (POPULATION x x-population)
                (AREA x x-area)
                (= x-density (% x-population x-area))
                (NOT (> x-density irish-density)))
            (INCREASING)
            2.))
        (SETQ AVERAGE-DISTANCE
            (AVERAGE
            (ALL distance
                (MEMBER pair
                    (EVAL COUNTRIES-AND-DENSITIES))
                (= country (CAR pair))
                (COUNTRY city country)
                (DISTANCE (PLACE city)
                          (PLACE LONDON)
                          distance))))
        (GIVE AVERAGE-DISTANCE)
        (GIVE COUNTRIES-AND-DENSITIES)
        (QUOTE *))
```

Answer 9.

```
AVERAGE-DISTANCE is

    1491.1892

COUNTRIES-AND-DENSITIES is

((LIBYA 3.)
 (ALGERIA 20.)
 (ALBANIA 24.)
 (TUNISIA 101.)
 (EGYPT 102.)
 (MOROCCO 108.))
*
```

This query shows at somewhat more length what a LISP programmer might make of an inquiry which calls for a more involved investigation. Assignment to the LISP variables COUNTRIES-AND-DENSITIES of the answer to one LOGIC call for later use within another (as well as for output) illustrates one more way in which the LOGLISP programmer can fruitfully exploit the interface between LOGIC and LISP. GIVE is just a dressed-up PRINT command which not only prints the value of its argument expression but also prints the expression.

4. CONCLUDING REMARKS

We are finding that LOGLISP indeed provides the rich setting for logic programming that our early encounters with PROLOG led us to seek. In particular we find it

most convenient to be able to invoke LOGIC from LISP, and LISP from LOGIC. It is very useful to have the answer to a query be delivered as a LISP data object and to be able to subject it to arbitrary computational analysis and manipulations.

The LOGIC programmer need not rely on the system builder to build in functions and predicates – he can write his own in LISP and then invoke them from LOGIC.

Nothing in our general design philosophy precludes (as far as we can see) a much faster deduction cycle than we have attained in this, our first, version of LOGLISP. We are confident that by (among other recourses) borrowing Warren's compilation techniques we shall be able to speed things up by at least a factor of 10, and we are currently working on doing this.

REFERENCES

Boyer, R. S., & Moore, J. S., (1972). The sharing of structure in theorem proving programs. *Machine Intelligence 7*, pp. 101–116 (eds. Meltzer, B. and Michie, D.). Edinburgh: Edinburgh University Press.

Bruynooghe, M., (1976). An interpreter for predicate logic programs, Part I. *Report CW 10*, Leuven, Belgium: Applied Mathematics and Programming Division, Katholieke Universiteit.

Clark, K. L., & McCabe, F., (1979). Programmers' Guide to IC-PROLOG. *CCD Report 79/7*, London: Imperial College, University of London.

Colmerauer, A., Kanoui, H., Pasero, R., & Roussel, P. (1973). Un Systeme de Communication Homme-machine en Francais. Report, Luminy, France: Groupe Intelligence Artificielle, Universite d'Aix-Marseille.

Green, C. C. (1969). Theorem proving by resolution as a basis for question-answering systems. *Machine Intelligence 4*, (eds. Meltzer, B. and Michie, D.) Edinburgh: Edinburgh University Press.

Hill, R. (1974). LUSH-resolution and its completeness. *DCL Memo No. 78*, Edinburgh: Department of Artificial Intelligence, University of Edinburgh, 1974.

Kowalski, R. A. (1974). Predicate logic as programming language. *Proceedings IFIP Congress.*

Kowalski, R. A. (1979). *Logic for problem solving.* New York and Amsterdam: Elsevier North Holland.

Landin, P. J. (1964). The mechanical evaluation of expressions. *Computer Journal*, 6, 308–320.

McDermott, D. (1980). The PROLOG phenomenon. *SIGART Newsletter* 72, 16–20.

Roberts, G. (1977). An implementation of PROLOG. M.Sc. Thesis, Waterloo: University of Waterloo.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *J. of the Assoc. for Comput. Mach.*, 12, 23–41.

Robinson, J. A. (1979). *Logic: Form and Function.* Edinburgh: Edinburgh University Press and New York and Amsterdam: Elsevier North Holland.

Robinson, J. A., & Sibert, E. E. (1980). Logic programming in LISP. *Technical Report*, School of Computer and Information Science, Syracuse University.

Roussel, P., (1975). PROLOG: Manuel de reference et d'utilisation. Luminy, France: Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille.

Strachey, C. (1972). Varieties of programming language. *Technical Monograph PRG-1049000*, Oxford University Computing Laboratory.

van Emden, M. H. (1977). Programming in resolution logic. *Machine Intelligence 8*, pp. 266–299, (eds. Elcock, E. W. and Michie, D.). Chichester: Ellis Horwood and New York: Halsted Press.

Warren, D. H. D., Periera, L. M., & Pereira, F., (1977). PROLOG – the language and its implementation compared with LISP. Proceedings of Symposium on AI and Programming Languages, *SIGPLAN Notices*, 12, No. 8, and *SIGART Newsletters* 64, 109–115.