

Logic, Computers, Turing, and von Neumann†

J. A. Robinson

Syracuse University

New York 13244-2010, U.S.A.

1 INTRODUCTION

The two outstanding figures in the history of computer science are Alan Turing and John von Neumann, and they shared the view that logic was the key to understanding and automating computation. In particular, it was Turing who gave us in the mid-1930s the fundamental analysis, and the logical definition, of the concept of ‘computability by machine’ and who discovered the surprising and beautiful basic fact that there exist universal machines which by suitable programming can be made to

†This essay is an expanded and revised version of one entitled *The Role of Logic in Computer Science and Artificial Intelligence*, which was completed in January 1992 (and was later published in the Proceedings of the Fifth Generation Computer Systems 1992 Conference). Since completing that essay I have had the benefit of extremely helpful discussions on many of the details with Professor Donald Michie and Professor I. J. Good, both of whom knew Turing well during the war years at Bletchley Park. Professor J. A. N. Lee, whose knowledge of the literature and archives of the history of computing is encyclopedic, also provided additional information, some of which is still unpublished. Further light has very recently been shed on the von Neumann side of the story by Norman Macrae’s excellent biography *John von Neumann* (Macrae 1992). Accordingly, it seemed appropriate to undertake a more complete and thorough version of the FGCS’92 essay, focussing somewhat more on the interesting historical and biographical issues. I am grateful to Donald Michie and Stephen Muggleton for inviting me to contribute such a ‘second edition’ to the present volume, and I would also like to thank the Institute for New Computer Technology (ICOT) for kind permission to make use of the FGCS’92 essay in this way.

perform any computation whatsoever. This theoretical work of Turing's had direct practical influence on the designs of the first modern general-purpose electronic digital computers in the mid-1940s. Von Neumann also had a prominent role in the emergence of these first machines, whose logical design has changed remarkably little in its general features during the almost fifty years since it was first formulated. The conventional wisdom is that von Neumann is primarily responsible for this design, which is accordingly known as the 'von Neumann architecture'. Without detracting from von Neumann's enormous importance, we maintain that this attribution somewhat distorts reality. The birth of the general-purpose computer, and with it computer science, owes far more to Turing than is usually acknowledged. Turing's contribution was both direct (arising from his own activities in the 1940s) and indirect (because of his little-known but powerful intellectual influence on von Neumann, starting from the mid-1930s).

In his recent sympathetic biography of von Neumann, Norman Macrae characterizes his subject's propensity to develop new insights with incredible speed and brilliant virtuosity, even though they were often not his own:

...Johnny grabbed other people's ideas, then by his clarity leapt five blocks ahead of them, and helped put them into practical effect (Macrae 1992, p. ix).

...Johnny borrowed (we must not say plagiarized) anything from anybody, with great courtesy and aplomb. His mind was not as original as Leibniz's or Newton's or Einstein's, but he seized other people's original (though fluffy) ideas and quickly changed them in expanded detail into a form where they could be useful ... (Macrae 1992, p. 23).

Macrae might have added that neither was von Neumann's mind as original as Turing's. Not only was Turing highly creative and imaginative, but he was also capable of intellectual feats just as dazzling as those ascribed to von Neumann.

Nor was Turing's universal machine idea at all 'fluffy'. Yet it was one of the great ideas von Neumann 'grabbed' enthusi-

astically in this fruitful way. Von Neumann always freely acknowledged this intellectual debt to Turing, not merely 'with great courtesy and aplomb' but with quite open admiration and generous praise². There was, however, no question of von Neumann's leaping 'five blocks ahead of' Turing in seeing that the general-purpose digital computer was implicit in Turing's idea. Turing had himself clearly seen virtually all the practical implications of his universal machine concept from the beginning³. Among them were the crucial role that logic would play in both the design and use of computers, and the alluring possibility of creating 'artificial intelligence' by programming a fast universal machine in suitable ways. In the course of exploiting Turing's idea, von Neumann also (when writing the EDVAC

²In a letter to Professor Brian Randell, Stanley Frankel (a Los Alamos physicist who, in collaboration with Nicholas Metropolis and at the instigation of von Neumann, had written the first major program to run on the ENIAC) writes: "I know that in 1943 or '44 von Neumann was well aware of the fundamental importance of Turing's paper of 1936 ... which describes in principle the 'Universal Computer' of which every modern computer (perhaps not ENIAC as first completed, but all later ones) is a realisation. Von Neumann introduced me to that paper and I studied it with care. Many people have acclaimed von Neumann as the 'father of the computer' ... but I am sure that he would never have made that mistake himself. He might well be called the midwife, perhaps, but he firmly emphasized to me, and to others I am sure, that the fundamental conception is owing to Turing. ... In my view von Neumann's essential role was in making the world aware of these fundamental concepts introduced by Turing and of the development work carried out in the Moore School and elsewhere. ... Both Turing and von Neumann, of course, also made substantial contributions to the 'reduction to practice' of these concepts but I would not regard these as comparable in importance with the introduction and explication of the concept of a computer able to store in its memory its program of activities and of modifying the program in the course of these activities." (Randell 1972, p.10)

³In (Carpenter and Doran 1986), p. 125, Mike Woodger writes: "During his stay at Princeton University shortly before the war Turing had discussed with John von Neumann the possibility of constructing a high-speed automatic computer with radio valves used essentially as switches. Both men were experts in the field of mathematical logic, and while the subject had very little bearing on the design or use of such machines it enabled them to see at once how the general problems of control and manipulation of sequences of binary digits could be affected."

Report in the spring of 1945) ‘grabbed’ the artificial neuron concept originated by McCulloch and Pitts in 1943, and the digital electronic techniques of Eckert and Mauchly, which he first heard about from Hermann Goldstine on the platform of the Aberdeen (Maryland) railway station in August 1944. Turing knew of the practical feasibility of large-scale digital electronic switching devices from his involvement in the early 1940s with the cryptanalytic ‘Colossus’ machines at Bletchley Park.

Mathematical logic was the primary professional field of both men. It gave them a view of computing which was more general than the view held by almost all of their colleagues, most of whom were either applied scientists interested in solving particular problems by numerical computation, or engineers interested in devising special tools to speed up traditional computing processes. The enormous practical significance of the idea of universality, or general-purposeness, seems at first to have been appreciated by hardly anyone other than Turing and von Neumann⁴. Indeed the conceptual power and significance even of the logical idea itself was at first understood only by the logicians and mathematicians who had followed the argument of (Turing 1936) and who were familiar with its background.

2 LOGIC AND THE COMPUTER

I expect that digital computing machines will eventually stimulate a considerable interest in symbolic logic ... [Turing 1947]. (Carpenter and Doran 1986, p. 122)

Logic and technology came together and gave us the computer. All modern general-purpose digital computers are phys-

⁴One of the few who noticed was, interestingly, J. R. Womersley, who as Superintendent of the Mathematics Division of the National Physical Laboratory was to recruit Turing there in 1945 in order to develop the ACE. He had read (Turing 1936) before the war, in 1937 or 1938, when he was employed at the Woolwich Arsenal on practical computation. He immediately saw its practical significance and was accordingly inspired to try to build a relay-based ‘Turing Machine’ (presumably a universal Turing Machine) with C. I. Norfolk, a telephone engineer. After some preliminary experiments the work was abandoned. See (Hodges 1983), p. 306 and note 5.30 on p. 556.

ical embodiments of the same logical abstraction Turing's universal machine. The practical exploitation of the universal machine concept, starting in the mid-1940s, owed much to Turing himself as well as to von Neumann. Turing and von Neumann not only played leading roles in the design and construction of the first working computers, but were also largely responsible for laying out the general logical foundations for understanding the computation process, for developing computing formalisms, and for initiating the methodology of programming: in short, for founding computer science as we now know it.

Today, logic continues to be a fertile source of abstract ideas for novel computer architectures for inference machines, dataflow machines, database machines, and rewriting machines. It provides a unified view of computer programming (which is essentially a logical task), and offers a systematic framework for reasoning about programs. Logic has been important in the theory and design of high-level programming languages. Indeed logical formalisms are the immediate models for two major logic programming language families: Church's lambda calculus (Church 1941) for functional programming languages such as LISP, SCHEME, ML, LUCID and MIRANDA, and the Horn-clause-resolution predicate calculus for relational programming languages such as PROLOG, PARLOG, and GHC. Peter Landin noted over twenty years ago (Landin 1966) that ALGOL-like languages, too, were merely 'syntactically sugared' only-slightly-augmented versions of Church's lambda calculus, and recently, another logical formalism, Martin-Löf's Intuitionistic Type Theory, has served (in, for example, Constable's NUPRL) as a very-high-level programming language, a notable feature of which is that a proof of a program's correctness is an automatic accompaniment of the program-writing process.

To design, understand and explain computers and programming languages; to compose and analyze programs and reason correctly and cogently about their properties; these are to practice an abstract logical art built on (in H. A. Simon's apt phrase) a 'science of the artificial' which studies rational artifacts in abstraction from the practical engineering details of their physical realization, yet maintains an analytical eye on their intrinsic ef-

iciency. The formal logician has had to become also an abstract engineer. In earlier times, the logician could be, and often was, cavalier about matters of efficiency when offering an algorithm for carrying out some logical construction. The point was often merely to establish that an algorithm existed, and not necessarily to give one which might in fact be practically useful.

3 LOGIC AND ARTIFICIAL INTELLIGENCE

In artificial intelligence (AI) research, logic has been used (for example, by McCarthy and Nilsson) as a rational model for knowledge representation and (for example, by Plotkin and Mugleton) as a guide for the organization of machine inductive inference and learning. It has also been used (for example, by Wos, Bledsoe, and Stickel) as the theoretical basis for powerful automated deduction systems which have proved theorems of interest to professional mathematicians. Logic's roles in AI, however, have been more controversial than its roles in the theory and practice of computing. Until the difference (if any) between natural intelligence and artificial intelligence is better understood, and until more experiments have tested the claims both of logic's advocates and of logic's critics concerning its place in AI research, the controversies will continue.

4 LOGIC AND THE ORIGIN OF THE COMPUTER

Logic's dominant role in the invention of the modern computer is not widely appreciated. The computer as we know it today was invented by Turing in 1936, an event triggered by an important logical discovery announced by Kurt Gödel in 1930. Gödel's discovery (Gödel 1931) decisively affected the outcome of the so-called Hilbert Program. Hilbert's goal was to formalize all of mathematics and then give positive answers to three questions about the resulting formal system: is it consistent? is it complete? is it decidable? Gödel found that no sufficiently rich formal system of mathematics can be both consistent and complete. In proving this, Gödel invented, and used, a high-level symbolic programming language: the formalism of primitive recursive functions. As part of his proof, he composed an elegant

modular functional program (a set of connected definitions of primitive recursive functions and predicates) which constituted a detailed computational presentation of the syntax of a formal system of number theory, with special emphasis on its inference rules and its notion of proof. This computational aspect of his work was auxiliary to his main result, but is enough to have established Gödel as the first serious programmer in the modern sense. Gödel's computational example inspired Turing a few years later, in 1936, to find an explicit but abstract logical model not only of the computing process, but also of the computer itself. Using these as auxiliary theoretical concepts, Turing disposed of the third of Hilbert's questions by showing (Turing 1936) that the formal system of mathematics is not decidable. Although his original computer was only an abstract logical concept, during the following decade (1937–1946) Turing became a leader in the design, construction, and operation of the first real computers.

The problem of answering Hilbert's third question was known as the Decision Problem. Turing interpreted it as the challenge either to give an algorithm which correctly decides, for all formal mathematical propositions A and B, whether B is formally provable from A, or to show that there is no such algorithm. Having first clearly characterized what an algorithm is, he found the answer: there is no such algorithm.

For our present purposes the vital part of Turing's result is his characterization of what counts as an algorithm. He based it on an analysis of what a 'computing agent' does when making a calculation according to a systematic procedure. He showed that, when boiled down to bare essentials, the activity of such an agent is nothing more than that of (as we would now say) a finite-state automaton which interacts, one at a time, with the finite-state cells comprising an infinite memory.

Turing's machines are plausible abstractions from real computers, which, for Turing as for everyone else in the mid-1930s, meant a person who computes. The abstract Turing machine is an idealized model of any possible computational scheme such a human worker could carry out. His great achievement was to show that some Turing machines are 'universal' in that they

can exactly mimic the behavior of any Turing machine whatever. All that is needed is to place a coded description of the given machine in the universal machine's memory together with a coded description of the given machine's initial memory contents. How Turing made use of this universal machine in answering Hilbert's third question is not relevant to our purpose here. The point is that his universal machines are the abstract prototypes of today's stored-program, general-purpose computers. The coded description of each particular machine is the program which causes the universal machine to act like that particular machine.

Abstract and purely logical as it is, Turing's work had an obvious technological interpretation: there is no need to build a separate machine for each computing task. Only one machine is needed — a universal machine — because it can be made to perform any conceivable computing task simply by writing a suitable program for it. Turing himself set out to build a universal machine.

He began his detailed planning in 1944. After 1943 he had gradually transferred his activities from Bletchley Park to Hanslope Park, ten miles distant from Bletchley, in order to work on speech-encipherment devices. He had spent four months — from November 1942 to March 1943 — in the United States, during which he discussed his 1936 paper and its practical implications with Claude Shannon at the Bell Laboratories⁵. There has been much speculation concerning the plausible possibility that during his time in the United States Turing met with von Neumann for discussions about the possibility of practical general-purpose computers. The only evidence of any kind we have for this is the fact that Lord Halsbury in 1959 referred (in *Ten Years of Computer Development*, *Computer Journal* 1, pp. 153–159) to:

...a meeting of two minds which cross-fertilised one another at a critical epoch in the technological development which they exploited. I refer of course to the meeting of the late Doctors Turing and von Neumann during the war, and all that came thereof

⁵(Hodges 1983), 241 – 255.

But when Professor Brian Randell⁶ later wrote to Lord Halsbury for more details he received the reply:

I am afraid I cannot tell you more about the meeting between Turing and von Neumann except that they met and sparked one another off. Each had, as it were, half the picture in his head and the two halves came together in the course of their meeting. I believe both were working on the mathematics of the atomic bomb project.

I recently became aware of the existence of a photograph, inscribed *National Cash Register Co., Dayton, Ohio, July 1943*, showing Turing together with a group of naval officers, one of whom is identified⁷ as William Friedman, the famous American code-breaker. This is the only evidence we have that Turing made at least one other visit to the United States after the November 1942 to March 1943 visit.

When the war ended in 1945 he moved to the National Physical Laboratory to pursue his goal full time. His real motive was already to investigate the possibility of artificial intelligence, a possibility he had frequently discussed with Donald Michie, I. J. Good, and other colleagues. He wanted, as he put it, to build a brain. By the end of 1945 Turing had completed his design for the ACE computer, based on his abstract universal machine. In designing the ACE, he was able to draw on his expert knowledge of the sophisticated new electronic digital technology which had been used at Bletchley Park to build special-purpose code-breaking machines (such as the Colossus). In the event, the ACE

⁶This account is taken from (Randell 1972).

⁷By Henry Tropp, a historian of computing, who has the photograph in his possession [personal communication, 25 June 1992]. It was Professor J. A. N. Lee who alerted me to the existence of this photograph. Tropp had earlier written (*The Effervescent Years*, IEEE Spectrum, February 1974, p. 76): "The National Cash Register Company offers a particularly intriguing 'might have been'. NCR actually had an electronic computing device constructed during the late 1930s. It was a high-speed arithmetic machine which could add, subtract and multiply electronically, and presumably this machine could have become the first commercial electronic computer had the company wished to pioneer in this field." Tropp was sent the photograph by the daughter of the late Joseph Desch, the NCR employee who designed this electronic device at Dayton in 1939.

would not be the first physical universal machine, for there were others who were after the same objective, and who beat NPL to it. Turing's 1936 idea had started others thinking. By 1945 there were several people planning to build a universal machine. One of these was John von Neumann⁸.

Turing and von Neumann first met in 1935, when Turing was still an unknown 23-year-old Cambridge graduate student. Von Neumann was already world-famous for his work in many scientific fields, including theoretical physics, logic and set theory, and several other important branches of mathematics. Ten years earlier, he had been one of the leading logicians working on Hilbert's Program, but after Gödel's discovery he suspended his specifically logical researches and turned his attention to physics and to mathematics proper. In 1930 he emigrated to Princeton, where he remained for the rest of his life.

Turing spent two years (from mid-1936 to mid-1938) in Princeton, obtaining a doctorate under Alonzo Church, who in 1936 had independently solved the Decision Problem. Church's method (Church 1936) was quite different from Turing's and was not as intuitively convincing. During his stay in Princeton, Turing had many conversations with von Neumann, who was enthusiastic about Turing's work and offered him a job as his research assistant. Turing turned it down in order to resume his research career in Cambridge, but his universal machine had already become an important item in von Neumann's formidable intellectual armory. Then came the war. Both men were soon completely immersed in their absorbing and demanding wartime scientific work.

⁸Ironically, von Neumann's EDVAC Draft Report had already been completed in June 1945 and a copy of it soon afterwards reached J. R. Womersley at the National Physical Laboratory. Womersley immediately visited Turing and showed the EDVAC Report to him as part of a (successful) attempt to recruit him for his project to design and build a comparable machine. Turing joined the NPL on 1 October 1945, and within three months had completed the ACE Proposal. So after having freely broadcast the logical seeds almost a decade previously, Turing was at last preparing to harvest the results, but only in the knowledge that on the neighboring farm the ears had ripened even faster.

By 1943, von Neumann was deeply involved in many projects, a recurrent theme of which was his search for improved automatic aids to computation. In late 1944 he became a consultant to a University of Pennsylvania group, led by J. P. Eckert and J. W. Mauchly, which was then completing the construction of the ENIAC computer (which was programmable and electronic, but not universal, and its programs were not stored in the computer's memory). Although he was too late to influence the design of the ENIAC, von Neumann essentially supervised the design of the Eckert–Mauchly group's second computer, the EDVAC. Most of his attention in this period was, however, focussed on designing and constructing his own much more powerful machine in Princeton at the Institute for Advanced Study (IAS) computer (von Neumann 1946). The EDVAC and the IAS machine both exemplified the so-called von Neumann architecture, a key feature of which is the fact that instruction words are stored along with data in the memory of the computer, and are therefore modifiable just like data words, from which they are not intrinsically distinguished.

The IAS computer was a success (Aspray 1990). Many close copies were eventually built in the 1950s, both in US government laboratories (the AVIDAC at Argonne National Laboratory, the ILLIAC at the University of Illinois, the JOHNIAC at the Rand Corporation, the MANIAC at the Los Alamos National Laboratory, the ORACLE at the Oak Ridge National Laboratory, and the ORDVAC at the Aberdeen Proving Grounds), and in foreign laboratories (the BESK in Stockholm, the BESM in Moscow, the DASK in Denmark, the PERM in Munich, the SILLIAC in Sydney, the SMIL in Lund, and the WEIZAC in Israel); and there were at least two commercial versions of it (the IBM 701 and the International Telemeter Corporation's TC-1).

The EDSAC, a British version of the EDVAC, was running in Cambridge by June 1949, the result of brilliantly fast construction work by M.V. Wilkes following his attendance at a 1946 EDVAC course. Turing's ACE project was, however, greatly slowed down by a combination of British civil-service foot-dragging and his own lack of administrative deviousness, not to mention his growing preoccupation with AI. In May 1948 Turing resigned

from NPL in frustration and joined the small computer group at the University of Manchester, whose small but universal machine started useful operation the very next month and thus became the world's first working universal computer. All of Turing's AI experiments, and all of his computational work in developmental biology, took place on this machine and its successors, built by others but according to his own fundamental idea.

Von Neumann's style in expounding the design and operation of EDVAC and the IAS machine was to suppress engineering details and to work in terms of an abstract logical description, based on the diagrammatic notation of McCulloch and Pitts' artificial neuron networks (McCulloch and Pitts 1943). He discussed both its system architecture and the principles of its programming entirely in such abstract terms. We can today see that von Neumann and Turing were right in following the logical principle that precise engineering details are relatively unimportant in the essential problems of computer design and programming methodology. The ascendancy of logical abstraction over concrete realization has ever since been a guiding principle in computer science, which has kept itself organizationally almost entirely separate from electrical engineering. The reason it has been able to do this is that computation is primarily a logical concept, and only secondarily an engineering one. To compute is to engage in formal reasoning, according to certain formal symbolic rules, and it makes no logical difference how the formulas are physically represented, or how the logical transformations of them are physically realized.

Of course no one should underestimate the enormous importance of the role of engineering in the history of the computer. Turing and von Neumann did not. They themselves had a deep and quite expert interest in the very engineering details from which they were abstracting, but they knew that the logical role of computer science is best played in a separate theater.

5 LOGIC AND PROGRAMMING

Since coding is not a static process of translation, but rather the tech-

nique of providing a dynamic background to control the automatic evolution of a meaning, it has to be viewed as a logical problem and one that represents a new branch of formal logics. [von Neumann 1947]

Much emphasis was placed by both Turing and von Neumann, in their discussions of programming, on the two-dimensional notation known as the *flow-diagram*. This quickly became a standard logical tool of early programming, and it can still be a useful device in formal reasoning about computations. The later ideas of Hoare, Dijkstra, Floyd, and others on the logical principles of reasoning about programs were anticipated by both (Turing 1949) and (von Neumann 1947). They stressed that programming has both a static and a dynamic aspect. The static text of the program itself is essentially an expression in some formal system of logic: a syntactic structure whose properties can be analyzed by logical methods alone. The dynamic process of running the program is part of the semantic meaning of this static text.

6 AUTOMATIC PROGRAMMING

Turing's friend Christopher Strachey was an early advocate, in the early 1950s, of using the computer itself to translate automatically from high-level 'mathematical' descriptions into low-level 'machine-language' prescriptions. His idea was to try to liberate the programmer from concern with 'how' to compute so as to be able to concentrate on 'what' to compute: in short, to think and write programs in a more natural and human idiom. Turing himself was not much interested in this idea, which he had already in 1947 pointed out as an 'obvious' one. In fact, he seems to have had a hacker's pride in his fluent machine-language virtuosity. He was able to think directly and easily in terms of bare bit patterns and of the unorthodox number representations such as the Manchester computer's reverse (i.e., low-order digits first) base-32 notation for integers. In this attitude, he was only the first among many who have stayed somewhat aloof from higher-level programming languages and higher-level machine architectures, on the grounds that a real professional

must be aware of and work closer to the actual realities of the machine. One senses this attitude, for example, throughout Donald Knuth's monumental treatise on the art of computer programming.

It was not until the late 1950s (when FORTRAN and LISP were introduced) that the precise sequential details of how arithmetical and logical expressions are scanned, parsed, and evaluated could routinely be ignored by most programmers and left to the computer to work out. This advance brought an immense simplification of the programming task and a large increase in programmer productivity. There soon followed more ambitious language design projects such as the international ALGOL project, and the theory and practice of programming language design, together with the supporting software technology of interpreters and compilers, quickly became a major topic in computer science. The formal grammar used to define the syntax of ALGOL was not initially accompanied by an equally formal specification of its semantics; but this soon followed. Christopher Strachey and Dana Scott developed a formal 'denotational semantics' for programs, based on a rigorous mathematical interpretation of the previously uninterpreted, purely syntactical, lambda calculus of Church. It was, incidentally, a former student of Church, John Kemeny, who devised the enormously popular 'best-selling' programming language, BASIC.

7 DESCRIPTIVE AND IMPERATIVE ASPECTS

There are two sharply-contrasting approaches to programming and programming languages: the descriptive approach and the imperative approach. The descriptive approach to programming focusses on the static aspect of a computing plan, namely on the denotative semantics of program expressions. It tries to see the entire program as a timeless mathematical specification which gives the program's output as an explicit function of its input (whence arises the term 'functional' programming). This approach requires the computer to do the work of constructing the described output automatically from the given input according to the given specifications, without any explicit direction from

the programmer as to how to do it.

The imperative approach focusses on the dynamic aspect of the computing plan, namely on its operational semantics. An imperative program specifies, step by step, what the computer is to do, what its 'flow of control' is to be. In extreme cases, the nature of the outputs of an imperative program might be totally obscure. In such cases one must (virtually or actually) run the program in order to find out what it does, and try to guess the missing functional description of the output in terms of the input. Indeed it is necessary in general to 'flag' a control-flow program with comments and assertions, supplying this missing information in order to make it possible to make sense of what the program is doing when it is running.

Although a purely static, functional program is relatively easy to understand and to prove correct, in general one may have little or no idea of the cost of running it, since that dynamic process is deliberately kept out of sight. On the other hand, although an operational program is relatively difficult to understand and prove correct, its more direct depiction of the actual process of computation makes an assessment of its efficiency relatively straightforward. In practice, most commonly-used high-level programming languages — even LISP and PROLOG — have both functional and operational features. Good programming technique requires an understanding of both. Programs written in such languages are often neither wholly descriptive nor wholly imperative. Most programming experts, however, recommend caution and parsimony in the use of imperative constructs. Some even recommend complete abstention. Dijkstra's now-classic Letter to the Editor (of the Communications of the ACM), entitled 'GOTO considered harmful' is one of the earliest and best-known such injunctions.

These two kinds of programming were each represented in pure form from the beginning: Gödel's purely descriptive recursive function formalism and Turing's purely imperative notation for the state-transition programs of his machines.

8 LOGIC AND PROGRAMMING LANGUAGES

In the late 1950s at MIT John McCarthy and his group began to program their IBM 704 using symbolic logic directly (McCarthy 1960). Their system, LISP, is the first major example of a logic programming language intended for actual use on a computer. It is essentially Church's lambda calculus (Church 1941), augmented by a simple recursive data structure (ordered pairs), the conditional expression, and an imperative 'sequential construct' for specifying a series of consecutive actions. In the early 1970s Robert Kowalski in Edinburgh (Kowalski 1974) and Alain Colmerauer in Marseille (Colmerauer 1973) showed how to program with another, only slightly augmented, system of symbolic logic, namely the Horn-clause-resolution form of the predicate calculus. PROLOG is essentially this system of logic, augmented by a sequentializing notion for lists of goals and lists of clauses, a flow-of-control notion consisting of a systematic depth-first, back-tracking enumeration of all deductions permitted by the logic, and a few imperative commands (such as the 'cut'). PROLOG is implemented with great elegance and efficiency using ingenious techniques originated by David H. D. Warren. The principal virtue of logic programming in either LISP or PROLOG lies in the ease of writing programs, their intelligibility, and their amenability to metalinguistic reasoning. LISP and PROLOG are usually taken as paradigms of two distinct logic programming styles (functional programming and relational programming), which on closer examination turn out to be only two examples of a single style (deductive programming). The general idea of purely descriptive deductive programming is to construe computation as the systematic reduction of expressions to a normal form. In the case of pure LISP, this means essentially the persistent application of reduction rules for processing function calls (Church's beta-reduction rule), the conditional expression, and the data-structuring operations for ordered pairs. In the case of pure PROLOG, it means essentially the persistent application of the beta-reduction rule, the rule for distributing AND through OR, the rule for eliminating existential quantifiers from conjunctions of equations, and

the rules for simplifying expressions denoting sets. By merging these two formalisms one obtains a unified logical system in which both flavors of programming are available both separately and in combination with each other.

LISP, PROLOG, and their cousins have thus demonstrated the possibility, indeed the practicality, of using systems of logic directly to program computers. Logic programming is more like the formulation of knowledge in a suitable form to be used as the axioms of automatic deductions by which the computer infers its answers to the user's queries. In this sense this style of programming is a bridge linking computation in general to AI systems in particular. Knowledge is kept deliberately apart (in a 'knowledge base') from the mechanisms which invoke and apply it. Robert Kowalski's well-known equational aphorism '*algorithm = logic + control*' neatly sums up the necessity to pay attention to both descriptive and imperative aspects of a program, while keeping them quite separate from each other so that each aspect can be modified as necessary in an intelligible and disciplined way.

The classic split between procedural and declarative knowledge again shows up here: some of the variants of PROLOG (the stream-parallel, committed-choice nondeterministic languages such as ICOT's GHC) are openly concerned more with the control of events, sequences, and concurrencies than on the management of the deduction of answers to queries. The uneasiness caused by this split will remain until some way is found of smoothly blending procedural with declarative within a unified theory of computation.

Nevertheless, with the advent of logic programming in the wide sense, computer science has outgrown the idea that programs can only be the kind of action-plans required by Turing-von Neumann symbol-manipulating robots and their modern descendants. The emphasis is (for the programmer, but not yet for the machine designer) now no longer entirely on controlling the dynamic sequence of such a machine's actions, but increasingly on the static syntax and semantics of logical expressions, and on the corresponding mathematical structure of the data and other objects which are the denotations of the expressions.

It is interesting to speculate how different the history of computing might have been if in 1936 Turing had proposed a purely descriptive abstract universal machine rather than the purely imperative one that he actually did propose; or if, for example, Church had done so. We might well now have been talking of 'Church machines' instead of Turing machines.

Let us call a *Church machine* an automaton whose states are the expressions of some formal logic. Each of these expressions denotes some entity, and there is an obvious semantic notion of *equivalence* among the expressions: 'A and B are equivalent' means 'A and B denote the same entity'. For example, the expressions

$$(23 + 4)/(13 - 4), \quad 1.3 + 1.7, \quad (\lambda z. (2z + 1)^{1/2})(4)$$

are equivalent, because they all denote the number three. A Church machine computation is a sequence of its states, starting with some given state and then continuing according to the transition rules of the machine. If the sequence of states eventually reaches a terminal state, and (therefore) the computation stops, then that terminal state (expression) is the output of the machine for the initial state (expression) as input. In general the machine computes, for a given expression, another expression *which is equivalent to it* but which is *as simple as possible*. For example, the expression '3' is as simple as possible, and is equivalent to each of the above expressions, and so it would be the output of a computation starting with any of the expressions above. These simple-as-possible expressions are said to be in 'normal form'. The 'program' which determines the transitions of a Church machine through its successive states is a set of 'rewriting' rules, together with a criterion for applying some one of them to any expression. A rewriting rule is given by two expressions, called the 'redex' and the 'contractum' of the rule, and applying the rule to an expression changes (rewrites) it to another expression. The new expression is a copy of the old one, except that the new expression contains an occurrence of the contractum in place of one of the occurrences of the redex.

For example: if the initial state is $(23 + 4)/(13 - 4)$ then the

transitions are:

$$\begin{array}{ll} (23 + 4)/(13 - 4) & \text{becomes } 27/(13 - 4), \\ 27/(13 - 4) & \text{becomes } 27/9, \\ 27/9 & \text{becomes } 3. \end{array}$$

but if the initial state is $\lambda z.(2z + 1)^{1/2} (4)$, then the transitions are:

$$\begin{array}{ll} (\lambda z.(2z + 1)^{1/2})(4) & \text{becomes } ((2 \times 4) + 1)^{1/2}, \\ ((2 \times 4) + 1)^{1/2} & \text{becomes } (8 + 1)^{1/2}, \\ (8 + 1)^{1/2} & \text{becomes } 9^{1/2}, \\ 9^{1/2} & \text{becomes } 3. \end{array}$$

Most of us are trained in early life to act like a simple purely arithmetical Church machine. We all learn some form of numerical rewriting rules in elementary school, and use them throughout our lives (but of course Church's lambda notation is not taught in elementary school, or indeed at any time except when people later specialize in logic or mathematics; but it ought to be). Since we cannot literally store in our heads all of the infinitely many redex-contractum pairs $\langle 23 + 4, 27 \rangle$, $\langle 2+2, 4 \rangle$, etc., infinite sets of these pairs are logically coded into simple finite algorithms. Each such algorithm (for addition, subtraction, and so on) yields the contractum for any given redex of its particular type. We hinted earlier that an expression is in normal form if it is as simple as possible. To be sure, that is a common way to think of normal forms, and in many cases it fits the facts. Actually, to be in normal form is not necessarily to be in as simple a form as possible. What counts as a normal form will depend on what the rewriting rules are. Normal form is a relative notion: given a set of rewriting rules, an expression in normal form is *one which contains no redex*.

In designing a Church machine care must be taken that no expression is the redex of more than one rule. The machine must also be given a criterion for deciding which rule to apply to an expression which contains distinct redexes, and also for deciding which occurrence of that rule's redexes to replace, in case there are two or more of them. A simple criterion is always

to replace the leftmost redex occurring in the expression.

A Church machine, then, is a machine whose possible states are all the different expressions of some formal logic and which, when started in some state (i.e., when given some expression of that logic) will 'try' to compute its normal form. The computation may or may not terminate: this will depend on the rules, on the initial expression, and on the criterion used to select the redex to be replaced. Some of the expressions for some Church machines may have no normal form. Since for all interesting formal logics there are infinitely many expressions, a Church machine is not a finite-state automaton; so in practice the same provision must be made as in the case of the Turing machines for adjoining as much external memory as needed during a computation.

Church machines can also serve as a simple model for parallel computation and parallel architectures. One has only to provide a criterion for replacing more than one redex at the same time. In Church's lambda calculus one of the rewriting rules ('beta-reduction') is the logical version of executing a function call in a high-level programming language. Logic programming languages based on Horn-clause-resolution can also be implemented as Church machines, at least as far as their static aspects are concerned.

In the early 1960s Peter Landin, then Christopher Strachey's research assistant, undertook to convince computer scientists that not merely LISP, but also ALGOL, and indeed all past, present, and future programming languages are essentially the abstract lambda calculus in one or another concrete manifestation. One need add only an abstract version of the 'state' of the computation process and the concept of 'jump' or change of state. Landin's abstract logical model combines declarative programming with procedural programming in an insightful and natural way (Landin 1966).

Landin's thesis also had a computer-design aspect, in the form of his elegant abstract logic machine (the SECD machine) for executing lambda calculus programs. The SECD machine language is the lambda calculus itself: there is no question of 'compiling' programs into a lower-level language (but more re-

cently Peter Henderson has described just such a lower-level SECD machine which executes compiled LISP expressions). Landin's SECD machine is a sophisticated Church machine which uses stacks to keep track of the syntactic structure of the expressions and of the location of the leftmost redex (Landin 1963).

We must conclude that the descriptive and imperative views of computation are not incompatible with each other. Certainly both are necessary. There is no need for their mutual antipathy. It arises only because enthusiastic extremists on both sides sometimes claim that computing and programming are 'nothing but' the one or the other. The appropriate view is that in all computations we can expect to find both aspects, although in some cases one or the other aspect will dominate and the other may be present in only a minimal way. Even a pure functional program can be viewed as an implicit 'evaluate this expression and display the result' imperative (as in LISP's classic read-eval-print cycle).

9 LOGIC AND ARTIFICIAL INTELLIGENCE

In AI a controversy sprang up in the late 1960s over essentially this same issue. There was a spirited and enlightening debate over whether knowledge should be represented in procedural or declarative form. The procedural view was mainly associated with Marvin Minsky and his MIT group, represented by Hewitt's PLANNER system (Hewitt 1969) and Winograd's application of it (Winograd 1972) to support a rudimentary natural language capability in his simple simulated robot SHRDLU. The declarative view was associated with Stanford's John McCarthy, and was represented by Green's QA3 system (Green 1969) and by Kowalski's eloquent advocacy of my resolution logic (Robinson 1965a, 1965b), suitably restricted to Horn clauses and applied parsimoniously according to restrictions which he and others had devised in the late 1960s and early 1970s, as a deductive programming language. Kowalski was able to make the strong case that he did because of Colmerauer's development of PROLOG as a practical *logic programming* language (Kowalski 1974). Eventually Kowalski found an elegant way to end the debate,

by pointing out a procedural interpretation for the ostensibly purely declarative Horn clause sentences in logic programs⁹.

There is a big epistemological and psychological difference between simply *describing* a thing and *giving explicit instructions for constructing* it, which corresponds to the difference between descriptive and imperative programming. Of course, some descriptions are already explicit recipes for construction, as for example

the square root of the sum of three cubed and three squared

But one cannot always so immediately see how to construct the denotation of a descriptive expression *efficiently*. For example, the meaning of the descriptive expression

the smallest integer which is the sum of two cubes in two different ways.

seems quite clear. We certainly understand the expression, but those who don't already (probably from reading of Hardy's famous visit to Ramanujan in hospital) know that it denotes 1729 will have to do some work, including some searching, to figure it out for themselves. It is easier to see that 1729 is the sum of two cubes in two different ways if one is given as a hint the two equations

$$1729 = 13^3 + 12^3$$

$$1729 = 10^3 + 9^3$$

⁹Donald Michie [personal communication, 16 May 1992] writes: "The first practical demonstration of Kowalski's fundamental idea was Maarten [van Emden]'s successful implementation and execution of Quicksort using Horn-clause Logic with an SL resolution theorem prover ... It was I who pushed Maarten into the Quicksort attempt." This makes the point that at Edinburgh in the early 1970s a theorem prover based on Kowalski and Kuehner's *linear resolution with selection function*, when implemented with the *structure-sharing techniques* pioneered by Boyer and Moore, and when limited to Horn-clause problems, already amounted to the *logic programming* which was soon to be closely associated with PROLOG. One should also mention even earlier anticipations of resolution-based logic programming: the 1969 QA3 question-answering system (Green 1969), and the 1969 ABSYS system (Foster and Elcock 1969). See (Elcock 1992) and also (Kowalski 1988).

but it needs at least a little work to find these equations oneself. Then to see that 1729 is the *smallest* integer with this property, one has to see somehow that all smaller integers lack it, and this means checking each one, either literally, or by some clever shortcut. To *find* 1729, then, as the denotation of the expression, one has to carry out the all of this work, in some form or another. There are of course many different ways to organize the task, some of which are much more efficient than others, some of which are less efficient, but more intelligible, than others. So to write a general computer program which would automatically and efficiently reduce the expression

the smallest integer which is the sum of two cubes in two different ways

to the expression '1729' and equally well handle other similar expressions, is not at all a trivial task.

10 AI AND PROGRAMMING

Automatic programming has never really been that. It is no more than the automatic translation of one program into another. So there must be some kind of program (written by a human, presumably) which starts off the chain of translations. An assembler and a compiler both do the same kind of thing: each accepts as input a program written in one programming language and delivers as output a program written in another programming language, with the assurance that the two programs are equivalent in a suitable sense. The advantage of this technique is of course that the source program is usually more intelligible and easier to write than the target program, and the target program is usually more efficient than the source program because it is typically written in a lower-level language, closer to the realities of the machine which will do the ultimate work. The advent of such automatic translations opened up the design of programming languages to express 'big' ideas in a style 'more like mathematics' (as Christopher Strachey put it). These big ideas are then translated into smaller ideas more appropriate for machine languages. Let us hope that one day we can look back at all the paraphernalia of this program-translation tech-

nology, which is so large a part of today's computer science, and see that it was only an interim technology. There is no law of nature which says that machines and machine languages are condemned, intrinsically, to being low-level. Surely we must strive towards machines whose topmost levels match our own.

Turing and von Neumann both made important contributions to the beginnings of AI, although Turing's contribution is the better known. His essay *Computing Machinery and Intelligence* is surely the most quoted single item in the entire literature of AI, if only because it is the original source of the famous so-called Turing Test (Turing 1950). The recent revival of interest in artificial neural models for AI applications recalls von Neumann's deep interest in computational neuroscience, a field he richly developed in his later years and which was absorbing all his prodigious intellectual energy during his final illness. When he died in early 1957 he left behind an uncompleted manuscript which was posthumously published (von Neumann 1958) as the book *The Computer and the Brain*.

11 LOGIC AND PSYCHOLOGY IN AI

I do not mean to say that there is anything wrong with logic; I only object to the assumption that ordinary reasoning is largely based on it. *M. L. Minsky, The Society of Mind*

AI has from the beginning been the arena for an uneasy co-existence between logic and psychology as its leading themes, as epitomized in the contrasting approaches to AI of John McCarthy and Marvin Minsky. McCarthy has maintained since 1957 that AI will come only when we learn how to write programs (as he put it) which have common sense and which can take advice. His putative AI system is a (presumably) very large knowledge base made up of declarative sentences written in some suitable logic (until quite recently he has taken this to be the first order predicate calculus), equipped with an inference engine which can automatically deduce logical consequences of this knowledge (McCarthy 1958). Many well-known AI problems and ideas have arisen in pursuing this approach: the Frame Problem, Nonmonotonic Reasoning, the Combinatorial Explo-

sion, and so on.

This approach demands a lot of work to be done on the epistemological problem of declaratively representing knowledge and on the logical problem of designing suitable inference engines. Today the latter field is one of the flourishing special subfields of AI. Mechanical theorem-proving and automated deduction have always been a source of interesting and hard problems. After over three decades of trying, we now have well-understood methods of systematic deduction which are of considerable use in practical applications.

Minsky maintains that humans rarely use logic in their actual thinking and problem solving, but adds that logic is not a good basis even for artificial problem solving — that computer programs based solely on McCarthy's logical deductive knowledge-base paradigm will fail to display intelligence because of their inevitable computational inefficiencies; that the predicate calculus is not adequate for the heuristically effective representation of most knowledge; and that the exponential complexity of predicate calculus proof procedures will always severely limit what inferences are possible.

Because it claims little or nothing, the view can hardly be refuted that humans undoubtedly are in some sense (biological) machines whose design, though largely hidden from us at present and obviously exceedingly complicated, calls for some finite arrangement of material components all built ultimately out of 'mere' atoms and molecules and obeying the laws of physics and chemistry. So there is an abstract design which, when physically implemented, produces (in ourselves, and the animals) intelligence. Intelligent machines can, then, be built. Indeed, they can, and do routinely, build and repair themselves, given a suitable environment in which to do so. Nature has already achieved NI — natural intelligence. Its many manifestations serve the AI research community as existence proofs that intelligence can occur in physical systems. Nature has already solved all the AI problems, by sophisticated schemes only a very few of which have yet been understood.

12 THE STRONG AI THESIS

Turing believed, indeed was the first to propound, the Strong AI thesis that artificial intelligence can be achieved simply by appropriate programming of his universal computer. Turing's Test is simply a detection device, waiting for intelligence to occur in machines: if a machine is one day programmed to carry on fluent and intelligent-seeming conversations, will we not, argued Turing, have to agree that this intelligence, or at least this apparent intelligence, is a property of the program? What is the difference between apparent intelligence, and intelligence itself? The Strong AI thesis is also implicit in McCarthy's long-pursued project to reconstruct artificially something like human intelligence by implementing a suitable formal system. Thus the Turing Test might (on McCarthy's view) eventually be passed by a deductive knowledge base, containing a suitable repertory of linguistic and other everyday human knowledge, and an efficient and sophisticated inference engine. The system would certainly have to have a mastery of (both speaking and understanding) natural language. Also it would have to exhibit to a sufficient degree the phenomenon of 'learning' so as to be capable of augmenting and improving its knowledge base to keep it up-to-date both in the small (for example in dialog management) and in the large (for example in keeping up with the news and staying abreast of advances in scientific knowledge). In a recent vigorous defense of the Strong AI thesis, Lenat and Feigenbaum argued that if enough knowledge of the right kind is encoded in the system it will be able to 'take off' and autonomously acquire more through reading books and newspapers, watching TV, taking courses, and talking to people (Lenat and Feigenbaum 1991).

It is not the least of the attractions of the Strong AI thesis that it is empirically testable. We will know if someone succeeds in building a system of this kind: that indeed is what Turing's Test is for.

13 EXPERT SYSTEMS

Expert systems are limited-scale attempted practical applications of McCarthy's idea. Some of them, such as the Digital

Equipment Corporation's XCON system for configuring VAX computing systems (Bachant and McDermott 1984), and the highly specialized medical diagnosis systems, such as KARDIO (Bratko *et al.* 1989), have been quite useful in limited contexts, but there have not been as many of them as the more enthusiastic proponents of the idea might have wished. The well-known book by Feigenbaum and McCorduck on the Fifth Generation Project was a spirited attempt to stir up enthusiasm for Expert Systems and Knowledge Engineering in the United States by portraying ICOT's mission as a Japanese bid for leadership in this field (Feigenbaum and McCorduck 1983).

There has indeed been much activity in devising specialized systems of applied logic whose axioms collectively represent a body of expert knowledge for some field (such as certain diseases, their symptoms, and treatments) and whose deductions represent the process of solving problems posed about that field (such as the problem of diagnosing the probable cause of given observed symptoms in a patient). This, and other, attempts to apply logical methods to problems which call for inference-making, have led to an extensive campaign of reassessment of the basic classical logics as suitable tools for such a purpose. New, nonclassical logics have been proposed (fuzzy logic, probabilistic logic, temporal logic, various modal logics, logics of belief, logics for causal relationships, and so on) along with systematic methodologies for deploying them (truth maintenance, circumscription, nonmonotonic reasoning, and so on). In the process, the notion of what is a logic has been stretched and modified in many different ways, and the current picture is one of busy experimentation with new ideas.

14 LOGIC AND NEUROCOMPUTATION

Von Neumann's view of AI was a 'logico-neural' version of the Strong AI thesis, and he acted on it with typical vigor and scientific virtuosity. He sought to formalize, in an abstract model, aspects of the actual structure and function of the brain and nervous system. In this he was consciously extending and improving the pioneering work of McCulloch and Pitts, who had

described their model as ‘a logical calculus immanent in nervous activity’. Here again, it was logic which served as at least an approximate model for a serious attack on an ostensibly nonlogical problem. A particularly interesting example of his foray into the practical engineering of artificial neural networks can be found in his demonstration of how reliability (fault tolerance) can be obtained by proper design methods even when the elementary neuron devices are prone to error (von Neumann 1956).

Von Neumann’s logical study of self-reproduction as an abstract computational phenomenon was not so much an AI investigation as an essay in quasi-biological information processing. It was certainly a triumph of abstract logical formalization of an undeniably computational process. The self-reproduction method evolved by Nature, using the double-helix structure of paired complementary coding sequences found in the DNA molecule, is a marvellous solution of the formal problem of self-reproduction. Von Neumann was not aware of the details of Nature’s solution when he worked out his own logical, abstract version of it as a purely theoretical construction, shortly before Crick and Watson unravelled the structure of the DNA molecule in 1953. Turing, too, was working at the time of his death on another, closely-related problem of theoretical biology — morphogenesis — in which one must try to account theoretically for the unfolding of complex living structural organizations under the control of the programs coded in the genes. This is not exactly an AI problem. One cannot help wondering whether Turing may have been disappointed, at the end of his life, with his lack of progress towards realizing AI. If one excludes some necessary philosophical clarifications and preliminary methodological discussions, nothing had been achieved beyond his invention of the computer itself.

The empirical goal of finding out how the human mind actually works, and the theoretical goal of reproducing its essential features in a machine, are not much closer in the early 1990s than they were in the early 1950s. After forty years of hard work we have ‘merely’ produced some splendid tools and thoroughly explored plenty of blind alleys. We should not be surprised, or even disappointed. The problem is a very hard one.

The same thing can be said about the search for controlled thermonuclear fusion, or for a cancer cure. Our present picture of the human mind is summed up in Minsky's recent book *The Society of Mind*, which offers a plausible general view of the mind's architecture, based on clues from the physiology of the human brain and nervous system, the computational patterns found useful for the organization of complex semantic information-processing systems, and the sort of insightful interpretation of observed human adult- and child-behavior which Freud and Piaget pioneered. Logic is given little or no role to play in Minsky's view of the mind (Minsky 1985).

Minsky rightly emphasizes (as logicians have long insisted) that the proper role of logic is in the context of justification rather than in the context of discovery. Newell, Shaw, and Simon's well-known propositional calculus theorem-proving program, the Logic Theorist, illustrates this distinction admirably (Newell *et al.* 1957). The Logic Theorist is a discovery simulator. The goal of their experiment was to make their program discover a proof (of a given propositional formula) by 'heuristic' means, reminiscent (they supposed) of the way a human would attack the same problem. As an algorithmic theorem-prover (one whose goal is to show formally, by any means, and presumably as efficiently as possible, that a given propositional formula is a theorem) their program performed nothing like as well as the best nonheuristic algorithms. The logician Hao Wang soon rather sharply pointed this out (Wang 1960), but it seems that the psychological motivation of their investigation had eluded him (as indeed it has many others). They themselves very much muddled the issue by contrasting their heuristic theorem-proving method with a ridiculously inefficient, purely fictional, 'logical' one which consisted of enumerating all possible proofs in lexicographical order and waiting for the first one to turn up with the desired proposition as its conclusion. This Swiftian parody may have been no more than a rhetorical flourish which got out of control, but it strongly suggested that they believed it really is more efficient to seek proofs heuristically, as in their program, than algorithmically with a guarantee of success. Indeed in the exuberance of their comparison they also provocatively

coined the wicked but amusing phrase ‘British Museum algorithm’ for this lexicographic-enumeration-of-all-proofs method — the intended sting in the epithet being that just as, *given enough time*, a systematic lexicographical enumeration of all possible texts will of course eventually succeed in listing any given text in the vast British Museum Library, so a logician, *given enough time*, must eventually succeed in proving any given provable proposition by proceeding along similar lines. Their implicit thesis was that *a proof-finding algorithm which is guaranteed to succeed for any provable input is necessarily unintelligent*. This may well be so: but that is not at all the same as saying that it is necessarily inefficient.

Interestingly enough, something like this thesis was anticipated in (Turing 1947):

...if a machine is expected to be infallible, it cannot also be intelligent. There are several mathematical theorems which say almost exactly that.

15 CONCLUSION

Logic’s abstract conceptual gift of the universal computer has needed to be changed remarkably little since 1936. Until very recently, all universal computers have been realizations of the same abstraction. Minor modifications and improvements have of course been made, perhaps the most striking one being internal memories organized into addressable cells, designed to be randomly accessible, rather than merely sequentially searchable (although external memories remain essentially sequential, requiring search). Other improvements have consisted largely of building into the finite hardware some of the functions which would otherwise have to be carried out by software (although in the recent RISC architectures this trend has actually been reversed). For over fifty years, successive models of the basic machine have been ‘merely’ faster, cheaper, physically smaller versions of *the same device*. In the past, then, computer science has pursued what is in fact an essentially *logical* quest: exploration of the literally unbounded possibilities of von Neumann’s version of the universal Turing machine. The technological chal-

lenge, of continuing to improve its physical realizations, has been largely left to the electrical engineers and computer architects, who have performed miracles and show every sign of continuing to do so.

In the future, we must hope that the logician and the engineer will find it possible and natural to work even more closely together to devise new kinds of *higher-level* computing machines which, by making programming easier and more natural, will help to bring artificial intelligence closer. That future has been under way for at least the past decade. Today we are already beginning to explore the possibilities of, for example, the Connection Machine (Hillis 1985), various kinds of neural network machines, and massively parallel machines for logical knowledge-processing.

It is this future that the bold and imaginative Fifth Generation Project has been all about. Japan's ten-year-long ICOT-based effort has stimulated (and indeed challenged) many other technologically advanced countries to undertake ambitious logic-based research projects in computer science. As a result of ICOT's international leadership and example, the computing world has been reminded not only of how central the role of logic has been in the past, as generation has followed generation in the modern history of computing, but also of how important a part it will surely play in the generations yet to come.

REFERENCES

- Aspray, W., and Burks, A. W. (1987). *Papers of John von Neumann on Computing and Computer Science*. MIT Press.
- Aspray, W. (1990). *John von Neumann and the Origins of Modern Computing*. MIT Press.
- Bachant, J., and McDermott, J. (1984). *R1 revisited: four years in the trenches*. The AI Magazine, Volume 5, Number 3, Fall 1984.
- Bratko, I., Mozetic, I., and Lavrac, N. (1989). *KARDIO: a study in Deep and Qualitative Knowledge for Expert Systems*. MIT Press.
- Carpenter, B. E. and Doran, R. W. (editors) (1946). *A. M. Turing's ACE Report of 1946 and other Papers*. MIT Press.

- Church, Alonzo (1936). *An unsolvable problem of elementary number theory*. American Journal of Mathematics, 58, pp. 345 – 363.
- Church, Alonzo (1936). *A note on the Entscheidungsproblem*. Journal of Symbolic Logic, Volume 1 (1936), pp. 40 – 41. Correction, *ibid.*, pp. 101 – 102.
- Church, Alonzo (1941). *The calculi of lambda-conversion*. Annals of Mathematics Studies Number 6. Princeton University Press.
- Colmerauer, A., Kanoui, H., Pasero, R., and Roussel, P. (1973). *Un Systeme de communication homme-machine en Francais*. Groupe d'Intelligence Artificielle, Universite d'Aix Marseille II, Luminy, France.
- Davis, Martin (1965). *The Undecidable*. Evergreen Press.
- Feigenbaum, Edward A., and McCorduck, Pamela (1983). *The Fifth Generation: Artificial Intelligence and Japan's Computer Challenge to the World*. Addison-Wesley.
- Foster, J. M., and Elcock, E. W. (1969). *Absys-1: an incremental compiler for assertions Q- an introduction*. In Machine Intelligence, 4, edited by D. Michie, pp. 423 – 429.
- Gödel, Kurt (1931). *Über formal unentscheidbare Sätze der Principia Mathematica und verwandte Systeme I*. Monatshefte für Mathematik und Physik, 38, pp. 173 – 198. English translations in (Davis 1965), (van Heijenoort 1971).
- Green, C. C. (1969). *Application of Theorem-proving to Problem-solving*. Proceedings of IJCAI 1 (Walker, D. E., and Norton, L. M., editors), Washington D. C., 1969, pp. 219 – 240.
- Henderson, Peter (1980). *Functional Programming*. Prentice-Hall.
- Hewitt, C. (1969). *PLANNER: a Language for proving Theorems in Robots*. Proceedings of IJCAI 1 (Walker, D. E., and Norton, L. M., editors), Washington D. C., pp. 295 – 301.
- Hillis, W. D. (1985). *The Connection Machine*. MIT Press.
- Hodges, Andrew (1983). *Alan Turing: The Enigma*. Simon and Schuster.
- Kowalski, R. A. (1984). *Predicate calculus as a programming language*. IFIP 1974 Proceedings, North Holland, pp. 569 – 574.
- Kowalski, R. A. (1988). *The early years of Logic Programming*. Communications of the Association for Computing Machinery, 31, pp. 38 – 42.
- Landin, P. J. (1963). *The Mechanical Evaluation of Expressions*.

- Computer Journal, 6, pp. 308 – 320.
- Landin, P. J. (1966). *The Next 700 Programming Languages*. Communications of the Association for Computing Machinery, 9, pp. 157 – 166.
- Lenat, Douglas B., and Feigenbaum, Edward A. (1991). *On the Thresholds of Knowledge*. Artificial Intelligence, 47, pp. 185 – 250.
- Macrae, Norman (1992). *John von Neumann*. Pantheon Books.
- McCarthy, John (1958). *Programs with Common Sense*. Proceedings of a Symposium on the Mechanisation of Thought Processes, Volume 1, pp. 77 – 84. National Physical Laboratory, London.
- McCarthy, John (1960). *Recursive Functions of Symbolic Expressions and their Computation by Machine*. Communications of the Association for Computing Machinery, 3, pp. 184 – 195.
- McCulloch, W. S., and Pitts, W. (1943). *A Logical Calculus of Ideas Immanent in Nervous Activity*. Bulletin of Mathematical Biophysics, 5, pp. 115 – 133.
- Minsky, Marvin (1985). *The Society of Mind*. Simon and Schuster.
- Newell, A., Shaw, J. C., and Simon, H. A. (1957). *Empirical explorations with the logic theory machine: a case study in heuristics*. Proceedings of the Western Joint Computer Conference, pp. 218 – 239.
- Randell, Brian (1972). *On Alan Turing and the Origins of Digital Computers*. Machine Intelligence, 7, pp. 3 – 20.
- Randell, Brian (1982) (editor). *The Origins of Digital Computers*. Third Edition, Springer Verlag.
- Robinson, J. A. (1965) *A machine-oriented logic based on the resolution principle*. Journal of the Association for Computing Machinery, 12, pp. 23 – 41.
- Robinson, J. A. (1965). *Automatic deduction with hyper-resolution*. International Journal of Computer Mathematics, 1, pp. 227 – 234.
- Shannon, Claude E., and McCarthy, John (1956) (editors). *Automata Studies*. Annals of Mathematics Studies Number 34. Princeton University Press.
- Stern, Nancy (1981). *From ENIAC to UNIVAC. An Appraisal of the Eckert-Mauchly Computers*. Digital Press.
- Taub, A. H. (1960) (editor). *John von Neumann: Collected Works*.

- 6 volumes. Macmillan, 1960 – 1963.
- Turing, Alan Mathison (1936). *On Computable Numbers, with an application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society, series 2, *Volume 42* (1936–37), pp. 230 –265. A Correction, *ibid.*, *Volume 43* (1937), pp. 544 – 546.
- Turing, Alan Mathison (1945). *Proposal for Development in the Mathematics Division of an Automatic Computing Engine (ACE)*. Completed by the end of 1945¹⁰. Published in (Carpenter and Doran 1986), pp. 20 – 105.
- Turing, Alan Mathison (1947). *Lecture to the London Mathematical Society on 20 February 1947*. Published in (Carpenter and Doran 1986), pp. 106 – 124.
- Turing, Alan Mathison (1947). *Intelligent Machinery*. Written in September 1947. First published in *Machine Intelligence 5*, edited by Bernard Meltzer and Donald Michie, Edinburgh University Press, 1969.
- Turing, Alan Mathison (1949). *Checking a Large Routine*. In *The Collected Works of A. M. Turing, Volume 1, Mechanical Intelligence*, 1992, pp. 129 – 131.
- Turing, Alan Mathison (1950). *Computing Machinery and Intelligence*. *Mind, Volume 59*, Number 236.
- van Heijenoort, Jean (1971) (editor). *From Frege to Gödel: A Source Book in Mathematical Logic, 1879 – 1931*. Harvard University Press, 1967.
- von Neumann, John (1945). *First Draft of a Report on the EDVAC*. Written in June 1945. Moore School of Electrical Engineering, University of Pennsylvania, Contract No. W-670-ORD-4926, June 1945. Excerpts in (Randell 1982). First full publication in (Stern 1981). Also in (Aspray–Burks 1987), pp. 17 – 82.
- Burks, Arthur W., Goldstine, Hermann H., and von Neumann, John (1946). *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*. Part I, *Volume 1* of a Report prepared for U.S. Army Ordnance Department under contract W-36-034-ORD-7481. (Taub 1960), *Volume 5*, pp. 34 – 79. Also in (Aspray–Burks 1987), pp. 97 – 142.
- Goldstine, Hermann H., and von Neumann, John (1947). *Planning*

¹⁰Cf. (Hodges 1983) note 6.5, p. 557.

and Coding of Problems for an Electronic Computing Instrument. Part II, *Volume 1* of a Report prepared for U.S. Army Ordnance Department under contract W-36-034-ORD-7481. (Taub 1960), *Volume 5*, pp. 80 – 151. Also in (Aspray-Burks 1987), pp. 553 – 602.

von Neumann, John (1956). *Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components*. In (Shannon and McCarthy 1956), pp. 43 – 98. Also (Taub 1960), *Volume 5*, pp. 329 – 378, and (Aspray – Burks 1987), pp. 553 – 602.

von Neumann, John (1958). *The Computer and the Brain*. Yale University Press, 1958.

Wang, Hao (1960). *Towards Mechanical Mathematics*. IBM Journal of Research and Development, *4*, pp. 2 – 22.

Winograd, T. (1972). *Understanding Natural Language*. Academic Press.

Woodger, M. (1958). *The History and Present Use of Digital Computers at the National Physical Laboratory*. *Process Control and Automation*, November 1958, pp. 437 – 442. Reprinted in (Carpenter and Doran 1986).