LOGIC AND THE DEVELOPMENT OF PROGRAMMING LANGUAGES, 1930–1975

Peter Mark Priestley

University College London

Thesis submitted in fulfillment of requirements for the degree of PhD.

May 2008

I, Peter Mark Priestley, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Abstract

Compared with the history of computing hardware, the history of software is in a relatively undeveloped state. In particular, the history of programming languages still consists for the most part of technical accounts presenting a rather Whiggish perspective on developments. Given the importance of software in the contemporary world, however, it is important to develop a more sophisticated understanding of the medium in which it is expressed.

This thesis considers some aspects of this history with the aim of examining the influence of formal logic on the evolution of notations for expressing computer programs. It is argued that this was not a natural or inevitable application of theory to practice, as is sometimes suggested, but a complex and contingent process with a rich history of its own.

Two introductory chapters discuss the work on computability carried out by logicians in the mid-1930s, and the controversial topic of the role of logic in the invention of the computer. The body of the thesis proceeds chronologically, considering machine codes, the introduction of higher level notations, structured programming and software engineering, and the early object-oriented languages.

The picture that emerges is that formal logic was deliberately employed by programming language designers to provide a model for a theoretical understanding of programming languages and the process of program development. This led to a flourishing research programme in the 1960s and 1970s, many of whose results are of lasting significance and value. The thesis concludes by examining the early history of object-oriented languages, arguing that this episode shows the emergence of limits to the applicability of the logical research programme.

Contents

Acknowledgements

I would like to acknowledge the practical and financial support provided by the University of Westminster during the work on this thesis, and also the invaluable support and enthusiasm shown throughout by my supervisor, Professor Donald Gillies.

Typographical conventions

The literature shows little consistency in the way that the names of programming languages are written: variants such as ‘FORTRAN’, ‘FORTRAN’ and ‘Fortran’ are commonly found. In this thesis, the following conventions are adopted: names which are more or less pronouncable are treated as proper names and written as ‘Fortran’, whereas unpronouncable acronyms are written as ‘NPL’. In direct quotation, however, the style adopted by the original source is preserved.

The name ‘Mark I’ is used throughout to refer to the machine designed by Howard Aiken and also known as the ‘Automatic Sequence Controlled Calculator’ or ‘ASCC’. A persuasive rationale for this practice has been given by I. Bernard Cohen in his biography of Aiken [Cohen, 1999, p. xix–xx].

Chapter 1

Introduction

The roots of modern digital computing lie in the desire to automate certain processes of calculation and collation. In the nineteenth century, Charles Babbage's Difference Engines were intended to compute various arithmetical tables, such as tables of logarithms [Babbage, 1864], and Herman Hollerith designed a punched-card system to facilitate the processing of the results of the 1890 census in the USA [Austrian, 1982]. This latter work led directly to the creation of a large punched-card industry in the first half of the twentieth century which automated many data processing tasks in industry and commerce [Aspray, 1990a].

Babbage's work did not result in any comparable application, but in the 1930s a number of independent developments were started by people inspired, like Babbage, by a desire to escape the labour of extended calculation. In 1935, Konrad Zuse had recently graduated as an engineer and, apparently motivated by a desire to automate the long and complex calculations he had had to perform, "decided to become a computer developer" [Zuse, 1993, p. 34]. He set up what he described as an "inventor's workshop" in his parents' apartment in Berlin, and by 1936 had started work on the Z1, the first in a line of machines leading in 1941 to the Z3, a device that has been described as the "first fully operational program-controlled computing machine in the world" [Ceruzzi, 1983, p. 29]. In 1937, Howard Aiken at Harvard University noted the repetitive nature of the computations involved in calculating approximate values of functions or performing numerical integration using infinite series [Aiken, 1937]. He designed a large-scale calculator intended to automate these and similar calculations. Known later as 'Mark I', this machine was developed in partnership with IBM and became operational in 1944 [Cohen, 1999].

The construction of automatic computing machines raised the question of how to specify the computational process that the machines should carry out. One approach was to build specialized machines that could carry out one particular task, but except in specialized domains such as cryptography, this approach was not followed: Babbage, Zuse and Aiken all designed general purpose machines which would be capable of carrying out a wide range of computations. At the time, large-scale computations were carried out by people, known as 'computers', who followed computational plans specified by instructions given on printed forms and sheets. Machines like Zuse's and Aiken's were designed as direct replacements for the humans in such a scenario, a point that Alan Turing made explicit in his theoretical analysis of computation [Turing, 1936]. The metaphor of 'giving instructions to the machine', usually in the form of a deck of punched cards, gradually emerged as a way of describing the way in which computations were specified.

As well as manipulating physical objects, such as punched cards intended for machine processing, people working with automatic computers began to represent the instructions symbolically for the purposes of designing or communicating computational plans. This led to a convergence with the discipline of mathematical logic which, since the end of the nineteenth century had been investigating the properties of symbolic notations that could be processed 'mechanically'. By the 1930s a notion of formal language had been developed which captured the important properties of such notations. For people with a background in mathematical logic, such as Turing and John von Neumann, it was natural to see the instructions given to an automatic computer as terms in a mechanically processable language, and hence to see an analogy between logic and the activity of programming automatic computers.

This was the beginning of what has turned out to be a long and involved relationship between logic and computer science, the anticipated significance of which was described by John McCarthy in the following terms: "It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last" [McCarthy, 1963a, p. 69]. Historical studies have been written describing various aspects of this relationship, covering for example the areas of artificial intelligence [Pratt, 1987] and theoretical computer science [Mahoney, 1997]. The principal aim of this thesis is to explore the history of this relationship in a different area, namely the development of programming notations and languages.

The significance of this development is not restricted to the history of computing. Following the publication of Frege's *Begriffschrift* in 1879, mathematical logic had developed a notation in which mathematical proofs could be completely formalized, and many metalinguistic properties of this notation had been formulated and proved. By contrast, no formal notation existed for the expression of algorithms. Although algorithmic procedures had been known since the earliest days of recorded mathematics, they were still described at best semi-formally: even the computation plans used in large-scale manual calculations required some degree of interpretation by the human computers carrying out the work.

When computational plans were first developed for automatic computers, there was no longer any possibility of the computing agent interpreting the instructions given to it. To ensure complete explicitness and the possibility of mechanical interpretation, plans were initially thought of simply as sequences of the basic operations, such as the addition of two numbers, that a machine could perform. Compared with a notation like the predicate calculus, however, there are at least two significant problems with this approach. Firstly, it is not a universal notation: programs are written in a dialect specific to one type of machine, and so cannot easily be read by or shared with workers accustomed to another type. Secondly, there is no *a priori* reason to suppose that a sequence of basic instructions provides a usable notation in which humans can write, study, or reason about the properties of programs.

From this perspective, the history of programming languages can be read as the search for an adequate formal notation for the expression of algorithms, a notation which would be as theoretically fruitful and as universally accepted as the notation of the predicate calculus was for the expression of proofs. Programming languages can therefore be seen not merely as a specialized topic within the history of computing, but as having a larger significance within the history of mathematics, completing the project of formalization which had come to prominence in the nineteenth century.

This viewpoint provides one way of understanding why logicians saw relevance for their work in the new and esoteric activity of programming automatic computers. This thesis examines how logic came to have a concrete influence on the development of programming languages and some aspects of the nature of that influence. The period under consideration is roughly the years from 1930 to 1975. By 1936, mathematical logic had developed a theoretical analysis of the concept of computability, and a number of practical projects in automatic computation were beginning to get under way. By the end of the period a degree of unanimity had been reached about many of the desirable features of programming languages, and the results obtained by researchers were beginning to have a significant effect on the industrial practice of programming, and indeed have continued to provide a foundation for the development of subsequent languages. A sense of closure at this time was sometimes recognized by contemporary writers, such as Peter Wegner who in 1976 suggested that “[i]t may well be that programming language professionals did their work so well in the 1950s that most of the important concepts have already been developed” [Wegner, 1976, p. 1224]. The focus of the thesis is on the programming languages developed for scientific and commercial data processing applications. More speculative developments, and in particular artificial intelligence (AI), were clearly influenced by logic as much as, if not more than, the mainstream. The relationship between logic and AI has long been recognized and subjected to detailed critical examination [Birnbaum, 1991, for example], however, and is not considered in any detail here.

1.1 The historiography of programming languages

This section briefly reviews prominent work on the history of programming languages, in order to gain an overview of that history and also some insight into the way the history has been written.

One of the earliest surveys of programming languages was published in 1964 by Saul Rosen, a mathematician who had been involved in computing in both academia and industry since the 1940s [Rosen, 1964]. Rosen provided a narrative account of early developments culminating with the distribution of Fortran II in 1957, a language which Rosen credited with being responsible for a “revolution” in the field of scientific computing. He then gave a detailed account of the development of the Algol 60 language in the years 1958–1960, and a survey of the languages that had been developed for data processing applications, leading up to the introduction of Cobol in 1961.

Rosen’s account combines a classification of existing languages with more detailed consideration of specific languages which were judged to be particularly significant or noteworthy. As was common at the time, Rosen distinguished scientific from data processing applications, and identified what he considered to be the most significant language in each area, Fortran and Cobol respectively. Algol 60 was also identified as a key development, although Rosen expressed reservations about its practical significance.

In 1967 Rosen published an anthology which reprinted a range of original papers on programming systems and languages [Rosen, 1967]. The structure of this book again illustrates the themes of classification and evaluation implicit in the earlier paper (which was reprinted in the book) while also widening its scope. A section titled simply “programming languages” contained papers on Fortran, Algol 60, Cobol and IBM’s ‘New Programming Language’, later known as PL/I. A separate section on “languages for processing lists and strings of symbols” contained papers on languages associated with the field of artificial intelligence, such as IPL-V and Lisp. The book also contained a number of papers on language processors and operating systems.

Another noteworthy publication in the 1960s was Jean Sammet's book "Programming Languages: History and Fundamentals", published in 1969 [Sammet, 1969]. Like Rosen, Sammet organized her presentation largely around a classification of application areas. Within each application area, a small number of languages were discussed in detail, with early developments and languages of restricted use being briefly described. Sammet adopted essentially the same classification as Rosen: Fortran and Algol 60 were listed as "languages for numerical scientific problems", Cobol as a language for "business data processing problems", and IPL-V, Lisp, Comit, Snobol and Trac as "string and list processing languages". PL/I was classified as a "multipurpose language", and Sammet also included sections describing "formal algebraic manipulation languages" and "specialized languages" for areas such as the control of machine tools and computer-aided design.

In 1971, at the conference of the International Federation for Information Processing (IFIP), Thomas Cheatham delivered a paper which surveyed "the recent evolution of programming languages" [Cheatham, 1971]. Cheatham wrote from a more academic perspective than Rosen and Sammet, and classified languages on technical grounds rather than by application area; he distinguished between "interactive" and "non-interactive" languages, and included categories of "special-purpose" and "extensible languages". In his account of the current situation, he listed a number of "popular" languages, including the then widely used Fortran and Cobol. However, in the category of popular languages Cheatham included Algol 60 and PL/I, despite admitting that they were much less widely used.

Why, then, were these less widely-used languages included? Cheatham identified certain languages as "important" — "Lisp ... takes its place with Algol-60 as being one of the most important programming languages ever developed" [Cheatham, 1971, p. 301] — and a major component of this importance was related to theoretical innovation rather than practical success. The importance of Algol 60, for Cheatham, came from the fact that it served as a model for many subsequent developments, and in particular led to "considerable and vigorous activity to develop formal models for syntax and semantic specification which has borne some very important fruit" [Cheatham, 1971, p. 299]. Similarly, "Lisp was probably the first programming language to have a formal semantic model" [Cheatham, 1971, p. 301]. This illustrates that by 1970 programming language theory had developed to such an extent that languages could be evaluated by something other than their practical or commercial success. For Cheatham, a key element in such an evaluation was the relationship a language bore to certain metalogical ideas.

In 1972, as part of the 25th anniversary of the ACM, the American professional body concerned with computing, both Rosen and Sammet were invited to contribute their thoughts on the current situation. Rosen stated that "[i]n 1972 and on into the foreseeable future, FORTRAN and COBOL are the languages in which most of the world's serious production programs are written" [Rosen, 1972, p. 591], although PL/I was also mentioned as an emerging "standard production language". PL/I was also mentioned because of the importance to language theory of the formal description that the IBM laboratory in Vienna had produced [Lucas and Walk, 1969]. Algol 60 and its successor, Algol 68, were described as primarily being of importance to the "theoretical development of programming language concepts" [Rosen, 1972, p. 593]. Describing more recent developments, Rosen adopted Cheatham's classification of 'extensible' and 'special purpose' languages, and highlighted the use of APL and Basic in interactive computing systems.

Sammet began by discussing some general questions about writing the history of programming languages, and in particular tried to describe the grounds for identifying certain languages as historically important. She concluded that "there are really two major reasons for a language to be considered significant: one is that it is economically practical and hence very useful, and the other that it is technically new" [Sammet, 1972, p. 603]; these reasons also seem to have informed Rosen's choice of subject material. Another feature of Sammet's paper was a "language history chart", which gave a graphical representation of many programming languages and the relationships between them: one language could be a subset or an extension of another, and in many cases a relationship of "influence" was shown, glossed as "sometimes the second language is 'like, or in the style of' the first" [Sammet, 1972, p. 606].

In 1976 Peter Wegner published a survey article describing "the first 25 years" of programming languages [Wegner, 1976]. This presented a more complex historical account in which three historical phases, each roughly corresponding to a decade, were identified. For Wegner, the 1950s represented a period of empirical investigation, in which important programming language concepts were discovered and described; in the 1960s a mathematical approach was taken to the elaboration and analysis of these concepts, and in the 1970s an engineering approach was taken to developing an effective software technology based on the earlier work. Within this periodization, Wegner described "some of the principal milestones of programming language development ... includ[ing] the development of specific programming languages, and the development of implementation techniques, concepts and theories" [Wegner, 1976, p. 1208]. The criteria according to which these milestones had been selected were not made explicit, however, but they did include familiar reference points such as Fortran, Algol 60 and Cobol.

The 1970s also saw a growing interest in the more general history of computing, and in June 1976 an International Research Conference on the History of Computing was held at Los Alamos [Metropolis et al., 1980]. This conference concentrated on developments before 1960, and the papers on programming languages largely described work carried out in and before the 1950s. A further conference, organized by the ACM in 1978, was dedicated to recording the history of individual programming languages [Wexelblat, 1981]. The criteria used for inclusion in the conference were that a language must have been in use in the period 1967–1977 and have had "considerable influence on the field of computing" in at least some of the following areas: "usage, influence on language design, overall impact on the environment, novelty (first of its kind), and uniqueness" [Sammet, 1981, p. xviii]. Two subsequent conferences, in 1993 and 2007, have been organized on similar lines [Bergin and Gibson, 1996, Ryder and Hailpern, 2007].

By the end of the 1970s, then, there was quite a sophisticated and long-established tradition of writing about the history of programming languages. At least two criteria were routinely used for assessing the significance of particular languages, namely the language's importance in practice or to theoretical development, and these were used to identify a set of 'landmark achievements' about which there was a considerable degree of consensus. There was also the beginnings of a more synoptic account of developments, apparent for example in the periodization suggested by Wegner.

All the historical work discussed so far was produced by people trained in computing and working in the field, either in industry or academia, but from about 1980 professional historians began to take an interest in the history of computing. One effect of this was an increased emphasis on the context surrounding technical developments. As Paul Ceruzzi put it in his 1980 thesis, "the emphasis will be on placing these descriptions [of early computing machines] into a larger context—of social, political, and historical themes as well as technical ones" [Ceruzzi, 1981, p. 2]. Among other things, this change contributed to a move of historical focus away from accounts of programming languages to a more general attempt to write the history of software.

This change of perspective has affected what is seen as significant in the past. For example, whereas Sammet recognized two distinct criteria for the significance of a programming language, historians of software tend to emphasize only the extent to which a language is used in industry. In the brief discussion of programming languages in their history of the computer, for example, Martin Campbell-Kelly and William Aspray discussed Fortran and Cobol, but made no mention of Algol 60, or indeed of any other programming language [Campbell-Kelly and Aspray, 1996].

In fact, programming languages as such have not been much discussed by historians of software, and when they have been it has often been in the context of a related topic. For example, Stuart Shapiro included significant discussion of programming languages in an article whose main focus was on different approaches taken to the task of software development [Shapiro, 1997], and Michael Mahoney touched on language issues in papers on the development of mathematical theories of computation [Mahoney, 1997]. The organisers of an International Conference on the History of Computing in 2000 devoted to "Software Issues" [Hashagen et al., 2002] proposed that the history of software be discussed under a number of general themes, including software as science and as engineering. Within these themes, however, programming languages were only incidentally mentioned. Later work, such as Campbell-Kelly's account of the software industry [Campbell-Kelly, 2003], has only reinforced this trend towards a more general, contextual approach.

A number of observations can be made on the basis of this brief survey of the literature of the history of programming languages. Firstly, there has been a move away from technical detail towards a kind of history which focuses almost exclusively on commercial and contextual issues. Campbell-Kelly recently reflected on this change in his own work, stating that he could not look back on his earlier, more technically-oriented work, "without a mild flush of embarrassment" [Campbell-Kelly, 2007, p. 40]. While recognizing the reasons that led historians to move away from the earlier, highly descriptive, technical accounts of programming languages, however, it is still possible to feel that something is missing from accounts which treat software history purely as business history, omitting any detailed consideration of the underlying technology. This point has recently been made in a slightly different context by Thomas Misa, who wrote that, "[a] contextual history—devoting close attention to specific details of the machines while situating them in their historical context—should be a vital ongoing tradition" [Misa, 2007, p. 53]. One of the aims of this thesis is to re-engage with the richness of the detailed technical history of programming languages, while simultaneously benefiting from the methodological insights of historians.

These insights are related to a second observation, that the history of programming languages has been written by two distinct groups of people, and that the judgements of these two groups on what is historically important are distinct. The first group can loosely be described as “insiders” [Mahoney, 1997], computing professionals with in most cases long experience and detailed technical knowledge of programming and programming languages. The second group, the ‘outsiders’, have typically been trained in a non-computing discipline, such as history or sociology. When judging a question such as the significance of a particular language, insiders tend to focus on aspects such as technical novelty and influence, whereas outsiders focus more on the importance of the language in an external, commercial context (though it should be noted that insiders such as Rosen and Sammet consistently applied both criteria in selecting languages for discussion). The distinction between these two groups has often been noted. In a discussion of the writing of the history of computing, for example, Jan Rune Holmevik distinguished two different types of history, one written largely by and for computer professionals and the other by historians with a wider range of interests, and argued for the legitimacy of each type within the context of its intended audience [Holmevik, 1994].

Insider history is sometimes said to be vulnerable to a number of methodological weaknesses, such as “Whiggism” and “internalism” [Holmevik, 1994], which together lead to the writing of a kind of history which represents the development of technology as an autonomous and teleological process, in which historical events follow purely technological laws of evolution to end up in what is essentially the current state of affairs. By contrast, outsider history emphasizes the importance of non-technological causes, such as economic and sociological factors, and presents technology as just one element of a larger historical manifold which evolves unpredictably in response to a wide range of events. The following sections examine the supposed fallacies of insider history in more detail, concluding with a description of the methodological approach taken in this thesis.

1.2 Whiggism

The Whig interpretation of history was characterized by the historian Herbert Butterfield as “studying the past with reference to the present”, and he identified a number of methodological errors in this approach [Butterfield, 1931]. Whig history misreads the past, seeing in it solely a reflection of contemporary concerns rather than making the positive effort of historical understanding necessary to grasp a situation that might in many ways differ from the present; it “over-dramatizes” the historical process, seeing it as the triumph of one set of ideas or actors over opposition and error rather than as a general ongoing transformation, the outcomes of which might in a dialectical fashion differ from anything anticipated by the historical actors; and it is over-concerned with questions of origin and causation, paying insufficient attention to the real complexity of historical transformation.

Whiggism in the history of computing has been specifically identified as a problem by a number of writers, including Holmevik and Anthony Hyman [Hyman, 1990]. A characteristic example from the history of programming languages is a paper by F. L. Bauer and H. Wössner which discusses the *Plankalkül* programming notation developed by Zuse in the mid-1940s [Bauer and Wössner, 1972]. The very title of the paper, which describes the *Plankalkül* as a "forerunner of today's programming languages", illustrates Butterfield's points, suggesting that historical events are to be regarded primarily in their relationship to the present, and that the principal motivation of the study is to identify origins and causes of contemporary techniques in the past. No attempt is made to place Zuse's work in its historical context, and it is seen as having value and interest primarily insofar as it anticipates later developments: "it is nevertheless surprising to what extent the *Plankalkül* already contains standard features of today's programming languages" [Bauer and Wössner, 1972, p. 678].

A further characteristic of this approach is the reinterpretation of concepts and terminology in the contemporary terms: "[Zuse] used *Angaben* for data and *Vorschrift* for algorithm. Not having at his disposition the word *Programm*, he called a program *Rechenplan*" [Bauer and Wössner, 1972, p. 678]. This approach obscures the evolution of technical concepts and the extent to which scientific terminology can change its meaning over time.

Finally, the paper nicely illustrates what Butterfield called the "Whig historian's quest for origins". Bauer and Wössner describe the *Plankalkül* as "a remarkable first beginning on the way to higher programming languages" [Bauer and Wössner, 1972, p. 678], despite acknowledging that it was never used in practice, and had a minimal influence on the development of later languages.

A slightly later and more sophisticated example of Whiggist tendencies is a paper by Donald Knuth and Luis Trabb Pardo, which played an important role in the historiography of programming languages by documenting and drawing attention to many of the early languages and notations that preceded the development of Fortran. Knuth and Trabb Pardo had a genuine interest in history, and unlike Bauer and Wössner they emphasized the need to "realize how long it took to develop the important concepts that we now regard as self-evident" [Knuth and Trabb Pardo, 1980, p. 198] and attempted to describe and illustrate each language as it would originally have been used. Nevertheless, they illustrated the languages under discussion by using them to code an artificial example program which required techniques that were not available in many of the older languages, and at the end of the paper the languages were classified according to the extent to which they supported a number of linguistic features salient at the time the paper was written.

Despite the methodological changes that have taken place in the history of computing since the 1970s, examples of Whiggism can still be found in historical writing on programming languages. In 1997, for example, Wolfgang Giloi published another account of the *Plankalkül* which, in terms very reminiscent of Bauer and Wössner, described it as “the first ‘non von Neumann’ programming language” [Giloi, 1997].

On the whole, it seems that Whiggish history is indeed open to criticism. The writing of history can not be completely divorced from the concerns of the present, as the selection of what is to be written about will necessarily be made from the standpoint of contemporary concerns and interests. Nevertheless, once this selection is made, historical accuracy is better served by making an attempt to understand the past for its own sake and in its own terms.

1.3 Internalism

A distinction has often been drawn between ‘internal’ and ‘external’ histories of science. In Kuhn’s words, internal history is “concerned with the substance of science as knowledge” whereas external history is “concerned with the activity of scientists as a social group within a larger culture” [Kuhn, 1968, p. 76]. The distinction is also applied to the kind of account that is given of the historical process, depending on whether internal or external causes are adduced to explain particular events. Internal history is typically written as an account of the autonomous development of scientific ideas, whereas external history deals with other factors, often economic, sociological or political, which are held to affect the development of the content of science.

At the heart of the distinction is a belief that the content of science can be clearly demarcated from the context in which it is developed. For example, an internal account of the development of programming languages might focus on the features provided by different languages, and the formal differences between them, and explain how innovations in one language influenced later designs. By contrast, an external account might examine how languages arose from the needs of computer users, and the way that the use of languages in industry affected their development. These different histories would typically appeal to different types of cause: the external account might consider the existing level of usage of Fortran and the increasing market share of IBM hardware, for example, while the internal account might discuss the technical advantages of certain language features.

An extreme account of the distinction between internal and external accounts was given by Imre Lakatos [Lakatos, 1971]. Lakatos proposed that philosophical accounts of scientific methodology which gave internal accounts of the "logic of scientific discovery", to use Popper's phrase, should be used as the basis for writing the history of science. A "rational reconstruction" of history would show the extent to which it could be viewed as conforming to the chosen methodology, and the non-rational residue would be given an external explanation. The ideal would be to give a rational, internal explanation of all the historical material: Lakatos thought of external factors as unscientific, only capable of having a negative effect on scientific progress by obstructing a rational, methodology-led development.

The priority that Lakatos and others gave to internal explanation was challenged by an increased interest in the sociology of knowledge. The "strong programme" put forward by David Bloor emphasized a positive role for social causes in the creation of scientific knowledge [Bloor, 1976], and proposed that a uniform causal explanation be given for all statements of scientific knowledge. The strong programme was highly influential in the subsequent development of science and technology studies, and its effects can be traced in the history of computing, as described above. As a result, internalism is sometimes characterized as an error in historical methodology comparable to Whiggism [Holmevik, 1994, for example]. Before considering this view, however, it will be useful to consider in more detail the typical characteristics of 'insider history'.

1.4 The insider perspective

Insider history of computing is overwhelmingly Whiggish: concerns of the present are routinely used as a guide for describing the past, and great emphasis is placed on identifying the origins of particular developments and the links whereby one event may have influenced another.

Furthermore, insider history is typically internal history, both in terms of the subject matter selected for examination and the explanatory model used. This explanatory model consists of two parts: the historical events that are selected for examination, and the relationships between them. From the insider perspective, historical events are thought of as discrete episodes, each marking a particular 'contribution' to the development of the discipline, such as the first publication of a new theorem, or a description of a new programming language. This tendency was noted by Kuhn, who wrote that a historian who believed in an cumulative account of science's development "must determine by what man and at what point in time each contemporary scientific fact, law and theory was discovered or invented" [Kuhn, 1962, p. 2]. A good example of this tendency in the history of programming languages is Wegner's article, cited above, which presented the history of programming languages as a series of "milestones".

Insider history is then typically concerned with tracing the migration of ideas, or intellectual content, between the discoveries or inventions so identified. Ceruzzi states this explicitly: "[a] central theme in the study of the history of science and technology is the transmission of ideas" [Ceruzzi, 1981, p. 170]. The process of transmission is often described in terms of 'influence': a key task is to trace the way in which one piece of work has influenced later workers in the field. For example, in his recent history of research on reasoning about programs, Cliff Jones stated that he would "trace the main line of development taking a key publication of ... Hoare as a pivotal point" [Jones, 2003, p. 26].

This basic picture is reinforced and motivated, as Kuhn noted, by a belief, which is usually left implicit, in the cumulative nature of scientific work. Historical episodes feature in insider history in so far as they have contributed to later work; influence is a process where a later worker will take up and further develop the ideas of a predecessor. The earlier work may be in certain ways developed, or 'its implications drawn out' but, by the very nature of the model, work which had no influence, or which is later deemed to have been erroneous, will tend not to feature in historical accounts. This in itself reinforces the Whiggish nature of insider history.

Another symptom of this view of history is the metaphorical use of the term 'pioneer' to describe those who carry out early work in a particular subject. This term has been widely used in the history of computing: for example, conferences organized by the ACM and IFIP have regularly included 'pioneer days'. This usage is taken from celebrations of early explorers and settlers of the American frontier, and encourages a perception of a new subject as an unexplored territory. The metaphor implies that the properties and features of the territory antedate its exploration, like an unexplored country, and that new ideas and the connections between them are waiting to be discovered by sufficiently gifted researchers. This sharply contrasts with later sociologically-inspired views of history which tend to emphasize the construction or invention of new developments rather than their discovery.

Clearly insider history, as described here, is susceptible to the fallacies of Whig history identified by Butterfield. The present state of development in a particular area is taken as the reference point for the history to be written, inescapably affecting the selection of historical incidents to be considered and their interpretation. The model can also be directly criticized, however, on the grounds of a lack of explanatory power.

One area in which this arises is with the question of anticipations: it frequently happens that it is possible to identify work which appears to be very much 'ahead of its time', but which appears to have had no influence. Later, the ideas are rediscovered, and at this point are taken up and developed further. The insider model has no way of explaining this phenomenon: as Jones says, referring to early papers by Goldstine and von Neumann and Turing, "[t]here is no compelling explanation of why more than a decade elapsed before the next landmark" [Jones, 2003, p. 29]. Despite this, a common theme in insider history is the search for 'forgotten pioneers': good examples of this are the papers on Zuse's *Plankalkül* cited earlier. Ironically, the Whiggish search for origins here seems to be in conflict with the explanatory model of influence.

A second explanatory shortcoming occurs when individuals appear not to have perceived some of the implications of their work, implications which from the perspective of the historian seem inescapable. Aiken is commonly criticized from this point of view, as Mark I did not in some respects conform to later ideas about computer design: for example, "Aiken does not seem to have recognized the general nature of logical control which is implied by his use of register 72" [Ceruzzi, 1981, p. 155]. Within the insider model, the only explanation that can be given of such cases is by appealing to the intellectual limitations of the individuals concerned. To extend the 'pioneer' analogy, it is as if an explorer could only be prevented from reaching the summit of an unclimbed mountain, for example, by some physical or logistical weakness: as in Lakatos's account, external causes are invoked to explain what would otherwise be a puzzling failure.

1.5 Methodology

In summary, then, insider history is predominantly Whiggish and internal, whereas following the recognition of the importance of sociological and related factors to scientific development, an alternative historiography has emerged which aims to avoid the fallacies of Whiggism and is largely external.

In this thesis, it is assumed that Whiggism is indeed a poor way to write history and to understand the past. It will be assumed that it is important to try to understand the past in its own terms, and in particular to avoid projecting the concepts and terminology of the present onto the past. Instead, an attempt will be made to depict the texture of innovation and recapture the meaning that events and ideas had for the people who were involved with them. Such a description should try to avoid selecting material on the grounds of its current relevance, and should not assume that particular events had the same significance as may now be attributed to them. Andrew Pickering has described this as the attempt to give a "*real-time* understanding of [scientific] practice" as opposed to the "retrospective" account typically provided by practising scientists [Pickering, 1995, p. 3]

Some writers have identified internalism as a historiographical fallacy comparable to Whiggism. Holmevik makes this criticism explicit [Holmevik, 1994], and it is implicit in much writing on the sociology of scientific knowledge. There are, however, strong reasons for supposing that internal history is a legitimate and important pursuit. Firstly, it reflects important properties of the scientific enterprise as it is perceived by its practitioners. Scientists are often inspired by the work of others, and certain pieces of work do give rise to large amounts of activity in new areas. Secondly, it is not obvious that all the fine detail of scientific work can be explained by purely external factors. For example, a convincing explanation for the development of programming languages during the 1950s could be based on the increased use of computers and the shortage of people with skills in machine code programming, but it is less plausible that the way in which the syntax of Algol 60 was presented, say, can be adequately explained by such economic and social factors.

The subject matter of this thesis is largely concerned with internal details of the development of programming languages and the influence of logic on that development. In the light of the foregoing discussion, the question arises of how this project can be carried out while avoiding the more problematic aspects of insider history. Part of the answer involves the avoidance of Whiggism, as discussed above, but in addition the following ideas drawn from research into the methodology of scientific activity have been found useful in the coming to an understanding of the historical phenomena.

The first of these is to recognize the importance of the notion of 'normal science' introduced by Kuhn [Kuhn, 1962]. For Kuhn, normal science is science which takes place in the context of a 'paradigm', or 'disciplinary framework' which among other things defines the problems that are to be addressed and the form that an acceptable solution will take. Normal science is contrasted in various ways with pre-paradigmatic science and the work that takes place at times of scientific revolution.

For the purposes of this thesis, a key feature of normal science is that it seems to possess some of the characteristics that the insider perspective attributes to science generally, notably the appearance of progressing through a steady accumulation of new results. Hence it can be argued that it is appropriate that an internal account be given of the history of normal science. In Kuhn's words, "[t]hat quite special, though still incomplete, insulation [of normal science from external factors] is the presumptive reason why the internal history of science, conceived as autonomous and self-contained, has seemed so nearly successful" [Kuhn, 1968, p. 81].

In this thesis, Lakatos's terminology of 'research programme' is used to describe a coherent tradition within which normal science progresses [Lakatos, 1970]. Although inspired by Kuhn's work, Lakatos's concept of research programme adds a number of features to the notion of normal science. Among these are an emphasis on the core propositions held by adherents to a research programme, the *hard core*, and the observation that within a particular field, a number of distinct research programmes can coexist, some progressing and others degenerating.

In the history of software, the term 'agenda' has been used by Michael Mahoney in describing the history of theoretical computer science. Although Mahoney cites neither Kuhn nor Lakatos, his notion has echoes of both: an agenda is defined as "what practitioners of the discipline agree ought to be done, a consensus concerning the problems of the field, their order of importance or priority, the means of solving them, and perhaps most importantly, what constitute solutions" [Mahoney, 1997, p. 619]. However, the term is also used in phrases such as "the agendas of semantics" [Mahoney, 2000, p. 31] to label diagrams which depict lines of influence between key workers and concepts in a field, in a style very reminiscent of the insider perspective.

From the point of view of historiography, the important thing about this family of ideas is that they provide a way of understanding how science can proceed independently of external factors, but without appealing exclusively to individual moments of inspiration, or some notion of the unfolding logic of a discipline. Paradigms, research programmes and agendas define 'what is to be done' in a particular field, providing direction and scope for creative individual work, and it is plausible that an 'internal' account can quite legitimately be written of the work carried out in such a context. The situation is different with work carried on outside a research programme, or before the formation of an initial paradigm in a particular area.

The second methodological guideline adopted in the thesis is to make explicit the development and change of meaning of technical concepts. Insider history tends to identify the current meaning of a term with the meaning or significance it may have had in the past, a Whiggish practice which can lead to significant misinterpretation of the past. The quotation given earlier from Bauer's paper on Zuse's *Plankalkül* exemplifies this tendency.

By contrast, it is often the case that a particular term will change its meaning over time, or that a concept will evolve from a simple initial form to a more complex set of ideas. In his study of Lakatos, Brendan Larvor has labelled approaches which describe such changes as 'dialectical': "Dialectical philosophy of mathematics studies the process by which mathematical argument improves mathematical concepts" [Larvor, 1998, p. 11]. Where appropriate, this thesis attempts to illustrate the way that practical experience has changed the technological concepts used to characterize that practice. It is a major weakness of the insider perspective that it is blind to the evolution of concepts in the light of experience.

A third methodological guideline relates specifically to the topic of the thesis, namely the use made of an existing discipline, mathematical logic, in guiding the development of a new subject area. Andrew Pickering has described this type of conceptual innovation as a process of *modelling* in which existing results are applied in a new domain, a process which he breaks down into three stages [Pickering, 1995, p. 115–7]. First, *bridging* is the choice of which existing work to take as a model in the exploration of a new domain. In Pickering's account, this choice is not determined, and researchers typically have alternative approaches available. Once a choice is made, however, a process of *transcription* follows, in which moves from the existing model are applied in the new domain. When transcription breaks down, because of resistances encountered in the application of ideas from the original domain in the new environment, it is followed by a process of *filling*, where aspects of the new system which do not correspond to anything in the model are completed, again in a fairly free way. Pickering's scheme is attractive for two reasons: it provides an analysis of what the rather vague term 'influence' might mean in at least some situations, and the notion of bridging can be seen as an important ingredient in some cases of paradigm formation. In the body of the thesis, the scheme will be applied to a number of historical situations; see for example Section 2.9 for an example of its applicability.

1.6 The argument

The historical event at the heart of the thesis is the emergence of high-level programming languages in the years around 1960. As discussed earlier, this period gave rise to a number of long-lived and influential languages, in particular Fortran, Algol 60, Cobol and Lisp, whose introduction was seen as highly significant by contemporary observers. It is argued that after 1960 the further investigation of programming languages was given coherence by a new research programme whose hard core was the identification of programming notations with the formal languages characterized by mathematical logic. The formation, development, significant achievements and potential decline of this research programme up to the mid 1970s are described in the body of the thesis.

Chapter 2 describes work in formal logic that was of particular importance to the later development of programming languages. A significant aspect of this was the articulation of a formal notion of computability in the first half of the 1930s. Turing's famous paper of 1936 is often seen as marking a transition between logic and computing; rather than viewing it as a 'milestone in computing', however, this chapter reads it in the context of related work in logic. The chapter also describes the concept of formal language that was developed by Carnap, Tarski and Morris in the 1930s and later drawn upon by workers in the field of programming languages.

Chapter 3 represents a slight detour from the main argument, discussing the development of computing machines in the period from the mid 1930s to 1950. Although an appreciation of this material is important for the understanding of later chapters, the main reason for its inclusion is to address the question about the influence of logic on 'the invention of the computer', about which there has been considerable debate. As with the relationship between logic and programming languages, it is argued that this is not the straightforward story of 'influence' that is sometimes claimed.

Chapter 4 describes the style of machine-level programming developed alongside the computers of the late 1940s and early 1950s. Although there was an awareness of a connection between logic and the activity of programming, there was little systematic connection between the two areas at this time and programming techniques were based fairly directly on characteristics of typical machine architectures.

Chapter 5 describes the gradual development in the 1950s of higher-level programming notations and the role played by logic in this development. Work in this decade shares many of the properties that Kuhn identified as characteristic of pre-paradigmatic science: workers in the field shared an ambition to 'make programming easier', but lacked a common understanding of how to achieve this goal. By 1960, however, the Algol 60 and Lisp languages had demonstrated two distinct ways in which logic could be applied in the development of programming languages.

Chapter 6 argues that Algol 60 played the role of a concrete paradigm, a technical achievement which served as the catalyst for the formation of a new research programme. Evidence for the existence of a coherent research programme is considered and the chapter describes that achievements of the new programme in the area of programming language theory and design.

The Algol research programme also had a significant impact on the practice of software development, and Chapter 7 considers the details of this process. The structured programming movement of the early 1970s is characterized in the light of this research programme.

Chapter 8 describes work from the early 1970s in the Algol research programme on the modular structure of programs and the unification of data and control logic. The attempts by the developers of the Smalltalk language to address similar questions are described, and in the light of subsequent developments it is suggested that Smalltalk represents the emergence of a competitor to the Algol research programme and marks a limit to the influence of logic on programming language development.

Finally, chapter 9 summarizes the argument and conclusions of the thesis, and offers suggestions for further work.

Chapter 2

Logic, computability and formal systems

This chapter surveys work in mathematical logic that was carried out in the 1930s and later drawn upon in the development of programming notations for automatic digital computers. Two developments are described, namely the evolution of a mathematical concept of computability and the articulation of a particular concept of formal language.

In the first half of the 1930s, the informal notion of effective computability was formalized in a variety of ways, and in 1936 different, but provably equivalent, accounts of it were published by Stephen Kleene, Alonzo Church, Emil Post and Alan Turing, a "confluence of ideas" that has been analysed by Robin Gandy [Gandy, 1988]. One reason for the importance of this work is that it is often seen as having had a material influence on the development of digital computers in the following ten years: the links between logic and the development of the computer are considered in detail in Chapter 3.

Later chapters are concerned with the influence of this logical work on the development of programming languages. This development made use not only of specific logical formalisms, but also of a metalinguistic account of those notations developed in the 1930s by Alfred Tarski, Rudolf Carnap and Charles Morris. Their work described a framework within which formal languages could be described and their properties discussed. The most important elements of this framework are described in Section 2.8.

2.1 Historical links between logic and computation

Before looking in detail at the work of the 1930s, it is useful to consider briefly the origins of the relationship between logic and computation. Logic is often defined as the study of valid patterns of reasoning in human thought, and if it is understood in this way the connection with computation is perhaps hard to see. Since the seventeenth century, however, philosophers and logicians have explicitly linked the two areas, and as a number of historical accounts have pointed out, a major theme in logical research has been to develop a calculus of reasoning so that deductions can be made or verified by algorithmic methods [Pratt, 1987, Davis, 2000].

An early connection was made by Hobbes, in whose view of language "general" or "universal" names signified collections, or "parcels", of objects, and "consequence" was a relation between names corresponding to the relation of inclusion between these collections. He explicitly compared reasoning with numerical computation, as follows:

> When a man *Reasoneth*, hee does nothing else but conceive a summe totall, from *Addition* of parcels; or conceive a Remainder, from *Substraction* of one summe from another: which (if it be done by Words,) is conceiving of the consequence from the names of all the parts, to the name of the whole; or from the names of the whole and one part, to the name of the other part. [Hobbes, 1651, chapter 5]

In the seventeenth century, this programme was developed in two ways. There were a number of attempts to develop a "real character", or a complete categorization of all the objects and concepts that could be referred to in discourse, together with symbols to represent them, and to base on this a "philosophical language" suitable for clear and unambiguous communication on any subject whatsoever [Wilkins, 1668, for example]. In this vein, Leibniz hoped to create a *characteristica universalis*, a universal language which would be adequate to represent the whole of human thought, by defining a set of elementary concepts from combinations of which all complex propositions could be expressed [Lewis, 1918].

Leibniz also planned to produce a *calculus ratiocinator*, an algebra which would make explicit the particular forms of 'reckoning' applied in reasoning. He made some progress with the calculus, formalizing, for example, the idempotency of logical addition, in which it differs from the numerical operation.

The development of logic in the direction of symbolic manipulation was continued by George Boole and others [Boole, 1854]. Boole's goal was to mathematize existing logic, particularly the syllogism, and the formulas of the resulting algebra of logic were not sufficiently expressive to capture all the features of sentences important to valid reasoning. In particular, the approach did not seem adequate to account for all the patterns of reasoning used in mathematics and so serve as a rigorous foundation for the subject.

Frege, by contrast, created a notation which did appear to be sufficiently expressive, but one for which the process of deduction did not possess the clarity and ease of use of algebra [Frege, 1879]. Deduction was represented by a formal system characterized by a number of axioms and rules of inference. Frege's system and those based on it, notably Russell and Whitehead's *Principia Mathematica* [Whitehead and Russell, 1910], shared with the algebra of logic the property that logical relationships could be checked by the manipulation of symbols, putting aside any thought of their meaning. Gödel among others highlighted this aspect of it:

> The development of mathematics toward greater precision has led, as is well known, to the formalization of large tracts of it, so that one can prove any theorem using nothing but a few mechanical rules. [Gödel, 1931]

In arithmetic, however, mechanical processes such as an algorithm for long division are guaranteed to lead in time to an answer to any problem, if correctly applied. In contrast, Frege's system and those based on it did not provide a guaranteed way of establishing whether a particular conclusion in fact followed from a set of premises. The decision problem or *Entscheidungsproblem*, given prominence by Hilbert and Ackermann [Hilbert and Ackermann, 1928], was the question whether an algorithmic process existed for establishing the relationship of logical consequence.

Although it did not directly address this question, Gödel's famous paper on the incompleteness of formalized theories of elementary arithmetic [Gödel, 1931] introduced a number of ideas and techniques that were widely used later, and is a convenient place to begin a more detailed consideration of the logical background.

2.2 Gödel's construction

Gödel's incompleteness proof is based on the idea of constructing, in the formal system of *Principia mathematica*, a self-referential sentence similar to those occurring in the semantic paradoxes. By analogy with the natural language sentences used in the 'liar' paradox, which assert their own falsity, Gödel constructed a formula which could be interpreted as asserting its own unprovability. Assuming the consistency of the system, it can be shown that neither this formula nor its negation is provable. This implies that the formula is true, however, and hence that the system is incomplete in the sense of containing true but unprovable formulas.

Informal presentations of the argument rely on the fact that natural languages are rich enough to serve as their own metalanguage: it is possible in English to refer to sentences of the English language, and the language also contains semantic predicates such as ‘is true’. Given these resources, the paradoxical sentences can easily be constructed. It is not obvious that a system designed for formalizing mathematics will necessarily be equally expressive, however, and a large part of Gödel’s paper is devoted to presenting the technical details of how to define the necessary syntactic and semantic properties in the formal system.

A formal system, for Gödel, is based on a set of primitive signs. The formulas of the system are finite sequences of these signs, and proofs are finite sequences of formulas. According to Gödel, “it is easy to state with complete precision *which* sequences of primitive signs are meaningful formulas and which are not” [Gödel, 1931, p. 147, emphasis in original], and the properties that distinguish proofs from non-proofs can similarly be specified. Gödel gave a description of a particular formal system P: this description is ‘precise’ but not formal, being stated in sentences of natural language. P consists of the logic of *Principia mathematica* combined with Peano’s axioms and its intended domain of interpretation is therefore the natural numbers.

Gödel then pointed out that for metamathematical purposes the exact choice of primitive signs is irrelevant, and proposed to use natural numbers as primitive signs in place of the conventional typographic symbols. By making use of the unique factorization theorem, Gödel showed that sequences of natural numbers could also be represented as numbers, and hence that it was possible to represent every primitive sign, formula and proof of P by a single natural number, its ‘Gödel number’. Gödel explicitly defined a function Φ which mapped linguistic elements to their Gödel numbers and picked out an “isomorphic image” [Gödel, 1931, p. 147] of P in the natural numbers. This encoding of the formulas of P as numbers, or *arithmetization*, is at the heart of Gödel’s approach.

Metamathematical discussion of P could now be carried out by talking about the isomorphic image $\Phi(P)$ rather than the formulas of P themselves. In other words, a metamathematical property R_P of formulas of P could be expressed as a property R_N of natural numbers such that R_P is true of certain formulas if and only if R_N is true of the corresponding Gödel numbers:

$$R_P(e_1, \ldots, e_n) \equiv R_N(\Phi(e_1), \ldots, \Phi(e_n))$$

It remained to show how the required metamathematical properties of P could be represented by formulas of P. Gödel did this in a slightly roundabout manner. First, he defined a class of number-theoretic functions and predicates which could be specified by so-called ‘recursive’ definitions; the precise nature of these definitions is discussed in more detail in the following section.

Gödel then gave a series of recursive definitions of a number of metamathematical properties of P, expressed in terms of the corresponding Gödel numbers in $\Phi(P)$. A number of these properties defined the syntax of the language P, thus demonstrating how the syntax of a formal system could be defined formally, or mathematically, rather than 'precisely' in a natural language. Further predicates gave number-theoretic definitions of the properties of being a meaningful formula and being a valid proof.

These recursive functions and predicates were defined in a metalanguage which went beyond the resources of the system P. As P was defined as a formal language for number theory, it is natural to ask whether they could have been defined in P itself. Gödel proved that this was in fact the case, and that in principle all the functions and predicates defining the metamathematical properties of P could have been directly defined by formulas of P itself.

In summary, then, Gödel's strategy was to code formulas of P as natural numbers, thus mapping the syntax of P into its own domain of interpretation. The metamathematical predicates required for his proof were then defined as number-theoretic predicates which were themselves expressible as formulas of P. These formulas could therefore be interpreted in two ways: firstly as statements about natural numbers, and secondly, thanks to the mapping Φ, as statements about formulas of P. By means of this second interpretation, P was enabled to act as its own metalanguage. The final requirement for the incompleteness proof was to construct a particular formula of P which, in this second interpretation, made reference to itself and asserted its own unprovability. The details of how Gödel achieved this are not of importance to this thesis, however, and will not be discussed further.

The techniques and arguments employed by Gödel were highly influential. In particular, Turing adapted Gödel's strategy in his definition of the universal machine, as discussed in detail in Section 2.7. The following section describes the further development of the notion of recursively defined functions. These formed the basis of one of the definitions of effective computability given in 1936, and were later of importance in the development of programming languages.

2.3 Recursive functions

The class of functions that Gödel called 'recursive' had been investigated before 1931. Definitions of functions over the natural numbers by 'simple recursion' were well-known: an example is the definition of addition by the definitions $a+0=a$ and $a+(b+1)=(a+b)+1$, for all a and b. The step-by-step evaluation of functions defined by simple recursion seemed to capture an important aspect of the notion of effective computability, and recursive definitions were widely discussed as an example of a finitistic approach to the definition of functions, acceptable to intuitionistic modes of thought.

In 1919 Skolem applied "the recursive mode of thought" to elementary arithmetic, with the aim of removing quantification over infinite domains from the system of *Principia mathematica*. He made extensive use of simple recursive definitions, basing his approach on "Kronecker's principle that a mathematical definition is a genuine definition if and only if it leads to the goal by means of a *finite* number of trials" [Skolem, 1923, p. 333, emphasis in original], thus highlighting the connection between recursive definition and the informal notion of effective computability.

In 1925 Hilbert categorized the "elementary" methods of constructing functions as substitution and recursion, and restated the connection between recursion and finiteness as follows: "[t]he method of search for the recursions required is in essence equivalent to that reflection by which one recognizes that the procedure used for the given definition is finitary" [Hilbert, 1926, p. 388].

Gödel's 1931 definition of recursive functions employed these two basic techniques. Substitution allowed a function ϕ to be defined from functions θ and $\chi_1 \ldots \chi_m$ by the equation

$$\phi(x_1, \ldots, x_n) \quad = \quad \theta(\chi_1(x_1, \ldots, x_n), \ldots, \chi_m(x_1, \ldots, x_n))$$

and a function ϕ could be "recursively defined in terms of" functions ψ and χ by the equations

$$\begin{aligned} \phi(0, x_2, \ldots, x_n) &= \psi(x_2, \ldots, x_n) \\ \phi(k+1, x_2, \ldots, x_n) &= \chi(k, \phi(k, x_2, \ldots, x_n), x_2, \ldots, x_n). \end{aligned}$$

A function ϕ was said to be "recursive" if it was a constant function, the successor function, or could be defined from other recursive functions using these two techniques [Gödel, 1931, p. 159].

It appeared, however, that the class of functions definable by these means did not exhaust those that appeared to be, in an informal sense, effectively calculable. Ackermann had proved that allowing higher-level recursions, involving "functionals" which could take functions as arguments, and definitions involving simultaneous recursion on more than one variable both allowed the definition of functions that could not be defined by substitution and simple recursion [Ackermann, 1928].

In 1934, Gödel gave a more general definition of recursive functions, based on a suggestion from Herbrand [Gödel, 1934]. This defined a "general recursive function" ϕ as being defined by a system of equations if from the equations exactly one equation of the form

$$\phi(k_1, \ldots, k_n) \quad = \quad m$$

could be derived, for natural numbers k_i and m, or in other words, a system of equations from which it followed that ϕ was in fact a function. Definitions involving only substitution and recursive definition were special cases for which the uniqueness of the function defined could be easily proved.

In a definitive paper of 1936, Kleene discussed Gödel's definition, introducing the now standard terminology of "primitive recursive" for functions defined using substitution and recursion only, and "recursive" for the wider class [Kleene, 1936a]. Kleene also gave the first 'formal' definition of recursive functions, in the sense of describing the sets of equations involved in a recursive definition as terms in a formal language. He adopted Gödel's technique of arithmetization and, like Gödel, defined a series of number-theoretic functions which characterized important syntactic properties of recursive definitions.

2.4 λ-definability

The λ-calculus was developed by Alonzo Church, and used in the first explicit attempt to give a formal characterization of the intuitive notion of effective calculability. It was inspired by work in which Schönfinkel had investigated the minimal set of primitive notions necessary for the formulation of logic, and in particular had attempted to remove the need for the use of variables in purely logical formulas [Schönfinkel, 1924]. Schönfinkel took as primitive the notion of a function, and generalized it firstly by allowing functions to use other functions as arguments and result values, and secondly by using this capability to reduce functions of several arguments to those of a single argument. Schönfinkel's work was further developed by Haskell Curry, who commented that the "*raison d'être* of the theory" was the fact that any expression involving variables $x_1, \ldots, x_n$ could be transformed into the form $Fx_1, \ldots x_n$ where F was a variable-free expression denoting a function of $x_1, \ldots, x_n$ [Curry, 1929].

Church first made use of this work in a paper on the foundation of logic. In Church's notation, the function of $\mathbf{x}$ defined by an expression $\mathbf{M}$ was represented by the notation $\lambda\mathbf{x}[\mathbf{M}]$, and the application of a function $\mathbf{F}$ to an argument $\mathbf{X}$ by the notation $\{\mathbf{F}\}(\mathbf{X})$ [Church, 1932]. These notations were related by the rule that function applications of the forms $\{\lambda\mathbf{x}[\mathbf{M}]\}(\mathbf{N})$ could be evaluated by substituting the argument $\mathbf{N}$ for the variable $\mathbf{x}$ in the expression $\mathbf{M}$, giving a result symbolized as $\mathrm{S}^{\mathbf{x}}_{\mathbf{N}}\mathbf{M}|$. This procedure could also be reversed, allowing a term to be rewritten as the application of a function to an argument.

It turned out that the attempt to base logic on these foundations gave rise to inconsistencies. However, following the discovery of a way of representing the natural numbers as λ-expressions, investigations by Church's students Kleene and Rosser revealed that an unexpectedly wide range of number-theoretic functions were λ-definable. In 1934, Church came to the opinion that the informal notion of effective calculability and the formal notion of λ-definability were equivalent, a belief dubbed as "Church's thesis" by Kleene [Rosser, 1984, p. 345].

In 1936 Church and Kleene published proofs that the λ-definable functions were precisely the recursive functions [Church, 1936, Kleene, 1936b], and Church proposed that this set of functions be identified as those which were effectively calculable. This paper gave a more detailed account of the formal properties of the notation, including an arithmetization which Church described as "the Gödel representation of a formula" [Church, 1936, p. 349]. This was used to demonstrate that syntactical operations on formulas were themselves recursive. Finally, Church answered the decision problem in the negative, by exhibiting an unsolvable problem.

Gödel was apparently unimpressed by the λ-calculus, and his definition of general recursive functions in 1934 has been described as an attempt, at Church's suggestion, to propose an alternative account of effective computability [Rosser, 1984]. For Church and his colleagues, however, the two formulations seemed intuitively acceptable, and their unexpected equivalence gave support to Church's thesis [Church, 1936, p. 346, footnote].

2.5 Direct approaches to defining effective computability

Both recursive functions and Church's λ-notation characterized effective computability in terms of formal systems in which the class of computable functions could be defined. However, the plausibility of this approach depends on the extent to which it is felt that the basic operations defined by the formal systems fall within the informal notion of effectiveness [Gandy, 1988, Soare, 1996]. In 1936, Emil Post and Alan Turing independently gave analyses of effective computability which, in Post's words, aimed at greater "psychological fidelity" [Post, 1936, Turing, 1936]. This work had a significant impact in gaining acceptance for Church's view that the informal notion of effective computability could be captured by a formal system.

Both Post and Turing described models which were based on taking seriously the metaphor that human beings perform certain intellectual tasks in a 'mechanical' manner. As discussed below, Turing made explicit reference to the behaviour of 'computers', the term then current to describe people carrying out complex calculations complex predefined plans. Jon Agar has described how similar 'mechanical' processes had been introduced in non-numerical areas, particularly in the British Civil Service, and has speculated that awareness of this was an additional factor leading to Turing's mechanical definition of computability [Agar, 2003]. Post and Turing abstracted two essential features from the familiar activity of human computation: an external medium on which the data involved in the computation could be recorded, and a representation of the instructions that the computer was following.

In Post's terminology, the model consisted of a "worker" operating in a "symbol space" by following a "set of directions", and the symbol space was a "two way infinite sequence of spaces or boxes" [Post, 1936, p. 103], each of which could be empty or contain a single symbol. The worker was assumed to be capable of performing a number of basic operations on the symbol space: moving to an adjacent box, placing or erasing a mark in a box, and detecting whether a box was marked or unmarked. The instructions followed by the worker were given as a numbered sequence of "directions". As well as start and stop instructions, the worker could be directed to perform a basic operation and then continue with a particular specified instruction, or to perform one of two alternative instructions depending on whether the currently occupied box was marked or unmarked.

Post only gave an informal description of sets of directions, and unlike Gödel, Kleene and Church, he did not define an arithmetization of his notation. Consequently he gave no formal proof of the equivalence of his formulation with the others, but he did anticipate that his account would "turn out to be logically equivalent to recursiveness in the sense of the Gödel-Church development" [Post, 1936, p. 105], as indeed proved to be the case.

Turing presented his model and notation in much more detail than Post, as described in the next section. Like Post, he made use of the notion of an external symbol space, which he described as a "tape" infinite in one direction only and divided into squares each capable of storing one of a range of distinct symbols. Rather than invoking the notion of a 'worker', however, Turing talked in terms of "machines" which embodied the agency necessary to execute a set of basic operations and be responsive to the contents of the tape. Turing went on to show that the operations carried out by a worker following a set of instructions could themselves be represented as a machine, the so-called "universal machine"; this development is discussed further in Section 2.7.

2.6 Turing's machine table notation

Turing's analysis was carried out by defining a class of abstract machines which are meant to embody the essential processes carried out by a human clerk or computer. Specific machines are described using a notation that Turing called "machine tables", and in the first half of the 1936 paper the machine table notation is developed into a powerful and sophisticated formalism for describing computations.

Despite the importance of Turing's work, the machine table notation has not received much attention from logicians and has gained a reputation for being obscure and confusing [Chaitin, 2001, p. 16, for example]. When it is discussed in detail, it is often with a view to identifying precursors or anticipations of features found in later programming languages [Knuth and Trabb Pardo, 1980, Copeland, 2004a]. Some of the resemblances between Turing's notation and programming languages are indeed striking, but focusing on these leads to a rather unhistorical interpretation of Turing's work. In this section the machine table notation will be considered in the context of the other notations for computability defined in the 1930s, and it will be argued that many of its features fit naturally into this context.

Turing machines

Imagine a human performing a 'mechanical' procedure, such as a calculation. Assuming that the procedure has been memorized, there will be no explicit written instructions being followed, but intermediate results may be recorded externally, perhaps on paper. A typical example of such a situation would be somebody carrying out an elementary calculation, such as a long multiplication. Turing described a class of abstract machines which were intended to simulate the behaviour of the human calculator in such cases. These machines have the following characteristics.

1. At any given moment, a machine is in exactly one of a finite number of states, known as *m-configurations*. These are intended to model the different 'states of mind' of a human computer, recording for example the stage reached in carrying out a computation.

2. The machine is supplied with a *tape*, representing the paper that a human worker would use to record the intermediate and final results of an ongoing computation. The tape consists of a sequence of *squares*, in each of which one of a finite number of symbols may be written.

3. At any given moment a machine has access to the contents of a subset of the squares on its tape, known as the *scanned* or *observed* squares. This reflects the fact that computations can be carried out which are so large that human computers cannot "immediately recognize" [Turing, 1936, p. 250] all the details of the work being done, and at different times will focus on particular aspects only.

Human memory has two distinct roles in computation, which were clearly distinguished by Turing. It is possible to carry out computations mentally, in which case the intermediate results of the computation are not written down, but simply remembered for a short period of time. The use of external aids such as paper becomes important as the size of the computations being undertaken increases. This might suggest that intermediate results could be represented either as *m*-configurations or as the contents of the machine's tape. However, Turing's distinction between *m*-configurations and the tape is based strictly on the differing functional roles of each component. The tape stores all the intermediate and final results of the computation, and in writing them all down Turing machines are more pedantic than a human computer might be. The *m*-configurations represent the computer's knowledge of which steps in the computation have been performed and what is to be done next, but not the results of those steps. If we imagine that the instructions specifying a computation are written down somewhere, the purpose of an *m*-configuration is simply to record which instruction is to be followed next.

In a Turing machine, computation proceeds by way of a sequence of discrete steps. At each step, the machine's behaviour is determined by its current *m*-configuration and the symbols in the currently scanned squares, together known as the machine's *configuration*. In a single step, the machine may perform one or more basic operations and possibly change its *m*-configuration. The basic operations are of two sorts: firstly, the symbols in the scanned squares may be changed, and secondly, the distribution of scanned squares on the tape may be changed. The new *m*-configuration and scanned symbols then determine the machine's behaviour in the next step of the computation.

A particular class of machines can be defined by specifying the structure of the tape, the set of symbols used by the machine, the distribution of scanned squares on the tape, and the repertoire of basic operations. The machines that Turing described in detail have a one-dimensional tape and are only capable of scanning one square at a time. The basic operations allow the symbol on the scanned square to be erased or altered, and only the squares adjacent to the scanned square may become the new scanned square (i.e. the machine can ‘move’ by only one square left or right at each step of a computation). The symbols used may vary from machine to machine, and are specified as required.

Machine tables

Turing described particular machines using a notation which he called ‘machine tables’. The simplest form of machine table consists of a set of rows, each defining the behaviour of a single step in a computation, or what the machine will do in a particular configuration. A row therefore consists of the following elements:

1. The *m*-configuration and the scanned symbol that define the configuration in question. In Turing’s paper, *m*-configurations were named by Gothic characters and symbols were shown literally.

2. The actions that the machine performs in this step of the computation. Turing used the abbreviations $P\alpha$ for the operation of writing the symbol α to the scanned square, E for the operation of erasing the symbol in the scanned square, and L and R for the operations of moving left and right, respectively.

3. The *m*-configuration that the machine is in at the end of a step in the computation, known as the *final m*-configuration.

Using these conventions, Turing gave the following example of a table describing a machine which prints the sequence ‘010101 . . .’ on alternate squares of the machine’s tape [Turing, 1936, p. 233].

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{b}$	None	$P0, R$	$\mathfrak{c}$
$\mathfrak{c}$	None	R	$\mathfrak{e}$
$\mathfrak{e}$	None	$P1, R$	$\mathfrak{k}$
$\mathfrak{k}$	None	R	$\mathfrak{b}$

This table is consistent with Turing’s description of the machines’ behaviour, which states that they can perform at most one write or erase operation and one move at each step in the computation. Turing immediately extended the notation, however, in two ways. Firstly, he allowed arbitrary sequences of basic operations to be specified in a single line of a table. In general, this reduces the number of *m*-configurations needed to describe a computation. Secondly, he introduced a notation, similar to the mathematical notation for 'definition by cases', for grouping together all the rows in a table that share the same *m*-configuration and selecting the required behaviour on the basis of the currently scanned symbol. Using these conventions a simpler table can be given for the machine defined above, using only one *m*-configuration [Turing, 1936, p. 234].

m-config.	*symbol*	*operations*	*final m-config.*
	None	$P0$	$\mathfrak{b}$
$\mathfrak{b}$	0	$R, R, P1$	$\mathfrak{b}$
	1	$R, R, P0$	$\mathfrak{b}$

Turing explained this form of machine table informally, saying that "for a configuration described in the first two columns the operations in the third column are carried out successively, and the machine then goes over into the *m*-configuration described in the last column. When the second column is left blank, it is understood that the behaviour of the third and fourth columns applies for any symbol and for no symbol" [Turing, 1936, p. 233]. It can be seen that the machine table notation is at least as expressive as the sequence of instructions used by Post to specify computations. The rows in a machine table are not ordered, but the sequencing of basic operations is determined by the explicit specification of a final *m*-configuration in each row, and the ability to select between alternatives is provided by allowing each instruction to discriminate on the value of the currently scanned symbol.

Variables and functions

Turing then went on to consider ways in which the task of writing tables for particular machines could be made easier. He observed that there are a number of basic processes, such as locating, copying, comparing and erasing symbols, which will form part of most machines and which may be carried out many times in a given computation. When defining significant numbers of recursive functions, Schönfinkel and Gödel had addressed this issue by defining simple functions which, by means of substitution, could be repeatedly used without redefinition in the definition of more complex functions, and Turing reinterpreted this technique in the context of his machine tables.

The definition of a function consists of an expression which defines in some way the transformation carried out by the function. In conventional functional notation, variables are used to represent those elements of the expression that can vary in different contexts of application. When a simple process represented by a machine table is repeated, the number of *m*-configurations and the basic operations carried out will remain the same, but the symbols involved and the *m*-configurations that specify what the machine should do next may differ from one occasion to another. Turing therefore introduced variables for symbols and *m*-configurations, and allowed *m*-configurations to be denoted not simply by names, but by expressions involving *m-configuration functions*, or *m-functions* [Turing, 1936, p. 236]. Tables containing these notational extensions were called *skeleton tables*.

The extended notation is best appreciated by means of an example. The table below defines a machine which will locate the first occurrence on the tape of a particular symbol, denoted by the variable α [Turing, 1936, p. 236]. If α does occur on the tape, the scanned square at the end of the computation will be the one containing the leftmost α and the final *m*-configuration will be that denoted by the variable $\mathfrak{C}$; if there are no αs on the tape, the final *m*-configuration will be that denoted by the variable $\mathfrak{B}$. In this example Turing introduces a convention whereby two occurrences of the special symbol 'ə' indicate the leftmost end of the used portion of the tape. In effect, computations take place on a tape which is unbounded to the right, and which contains at the start the symbols 'əə'.

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{f}(\mathfrak{C}, \mathfrak{B}, \alpha)$	ə	L	$\mathfrak{f}_1(\mathfrak{C}, \mathfrak{B}, \alpha)$
	not ə	L	$\mathfrak{f}(\mathfrak{C}, \mathfrak{B}, \alpha)$
$\mathfrak{f}_1(\mathfrak{C}, \mathfrak{B}, \alpha)$	α		$\mathfrak{C}$
	not α	R	$\mathfrak{f}_1(\mathfrak{C}, \mathfrak{B}, \alpha)$
	None	R	$\mathfrak{f}_2(\mathfrak{C}, \mathfrak{B}, \alpha)$
$\mathfrak{f}_2(\mathfrak{C}, \mathfrak{B}, \alpha)$	α		$\mathfrak{C}$
	not α	R	$\mathfrak{f}_1(\mathfrak{C}, \mathfrak{B}, \alpha)$
	None	R	$\mathfrak{B}$

This table defines three *m*-functions, $\mathfrak{f}$, $\mathfrak{f}_1$ and $\mathfrak{f}_2$. On different occasions of use, these expressions would denote different *m*-configurations, depending on the values supplied for the variables $\mathfrak{B}$, $\mathfrak{C}$ and α. The names of the *m*-functions were presumably chosen to emphasize the fact that they are parts of a self-contained table with a unified purpose. When the machine is in *m*-configuration $\mathfrak{f}(\mathfrak{C}, \mathfrak{B}, \alpha)$ it moves left until it reaches the beginning of the tape. It then goes into *m*-configuration $\mathfrak{f}_1(\mathfrak{C}, \mathfrak{B}, \alpha)$ and begins to move right, searching for an α. If an α is found the computation finishes and the machine goes to the 'success' *m*-configuration, $\mathfrak{C}$. If a blank square is found the machine goes into *m*-configuration $\mathfrak{f}_2(\mathfrak{C}, \mathfrak{B}, \alpha)$, and otherwise it moves on to the next square to the right. The behaviour of the machine in *m*-configuration $\mathfrak{f}_2(\mathfrak{C}, \mathfrak{B}, \alpha)$is the same as in *m*-configuration $\mathfrak{f}_1(\mathfrak{C}, \mathfrak{B}, \alpha)$, with one exception: the discovery of a blank square means that two blanks in a row have been found, and hence, according to a second convention adopted by Turing, that the end of the tape has been reached. In this case the machine goes to the 'failure' *m*-configuration, $\mathfrak{B}$.

Skeleton tables therefore introduce the familiar logical apparatus of variables and functions into the machine table notation. It should be noted that *m*-function expressions have a different significance if they appear in the first or last column of a table. In the first column they behave rather like expressions in Church's λ notation, binding the variables that appear in that row: this can be seen by noting that the variables in a row could be consistently renamed without changing the behaviour specified by the table. In the final column the *m*-functions are applied to determine the machine's next *m*-configuration.

Outside of skeleton tables, *m*-functions are applied by replacing the bound variables by the names of *m*-configurations and symbols, and the resulting terms, such as $\mathfrak{f}(\mathfrak{d}, \mathfrak{e}, x)$, denote *m*-configurations. Such applications enable a skeleton table, like a function, to be 'reused' whenever it is necessary to carry out the process that it defines. For example, given the following table fragment, a machine in the *m*-configuration $\mathfrak{c}$ will proceed to delete the first occurrence of the symbol x on the tape.

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{c}$			$\mathfrak{f}(\mathfrak{d}, \mathfrak{e}, x)$
$\mathfrak{d}$		E	$\mathfrak{e}$
$\mathfrak{e}$			...

From *m*-configuration $\mathfrak{c}$, the machine will immediately move to the *m*-configuration specified by the expression $\mathfrak{f}(\mathfrak{d}, \mathfrak{e}, x)$. The effect of this *m*-configuration is defined by the skeleton table above. With the obvious substitution for the bound variables, the machine will then search for the first occurrence of the symbol x on the tape. If this search is successful, the machine will move to *m*-configuration $\mathfrak{d}$, in which the currently scanned symbol, the x that has just been located, will be erased. After erasure, or in the case that no x was found on the tape, the machine will be in *m*-configuration $\mathfrak{e}$.

Turing defined the general effect of *m*-function application in the same way as function application is defined in the λ-calculus, "by repeated substitution [of *m*-configurations and symbols in place of variables] in the skeleton tables" [Turing, 1936, p. 236]. Applying this procedure to the table fragment, and replacing expressions like $\mathfrak{f}(\mathfrak{d}, \mathfrak{e}, x)$ with a simple *m*-configuration name such as $\mathfrak{f}$, yields the following expanded table in the original notation without variables and *m*-functions.

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{c}$			$\mathfrak{f}$
$\mathfrak{f}$	ə	L	$\mathfrak{f}_1$
	not ə	L	$\mathfrak{f}$
$\mathfrak{f}_1$	x		$\mathfrak{d}$
	not x	R	$\mathfrak{f}_1$
	None	R	$\mathfrak{f}_2$
$\mathfrak{f}_2$	x		$\mathfrak{d}$
	not x	R	$\mathfrak{f}_1$
	None	R	$\mathfrak{e}$
$\mathfrak{d}$		E	$\mathfrak{e}$
$\mathfrak{e}$			$\ldots$

In practice, of course, this substitution would remain implicit and use of the *m*-function $\mathfrak{f}$ in the original table would be understood as a way of invoking an operation to find a particular symbol. In this way, skeleton tables provide a mechanism for building complex tables out of simpler and independent components. Furthermore, *m*-functions allow the instructions for a particular task to be defined once and then used many times. There may be many occasions in a complex computation when specific symbols must be located: once the skeleton table is defined, this can be achieved simply by writing $\mathfrak{f}$, with suitable arguments, as the final *m*-configuration of some row in the table.

Thought of in this way, an analogy can be drawn between skeleton tables and the open subroutines or macro instructions found in later programming languages, as noted by Knuth and Trabb Pardo [Knuth and Trabb Pardo, 1980, p. 201] and more recently by Copeland [Copeland, 2004a, p. 12]. However, Turing's procedure in the introduction and use of skeleton tables is readily comprehensible in the light of contemporary work on effective computability. Gödel's formal definition of provability, for example, began with the definition of very simple arithmetical functions, and then reused these to define a series of increasingly complex functions [Gödel, 1931], and the general strategy, of defining complex functions in terms of simpler ones, was commonly used by writers on recursive functions and effective computability [Skolem, 1923, Kleene, 1935a, Kleene, 1935b, for example]. In this context, Turing's use of skeleton tables can be seen as a natural extension of the same strategy to the domain of machine tables.

Defining *m*-functions using substitution and recursion

In Gödel's original definition of recursive functions, new functions could be defined in terms of old ones by using the two techniques of substitution and recursive definition. Turing used both techniques in the machine table notation to enable the definition of new *m*-functions in terms of existing ones.

Substitution, in this context defined by Gödel as the "substitution of some of the preceding functions at the argument places of one of the preceding functions" [Gödel, 1931, p. 159, footnote], was provided by allowing *m*-function expressions to appear in the argument positions of the application of an *m*-function. Just as with recursive functions, this provides a powerful mechanism whereby new operations can be built up in terms of those already defined. For example, Turing gave the following definition of an operation $\mathfrak{e}$ to erase the first occurrence of symbol a on the tape [Turing, 1936, p. 237].

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{e}(\mathfrak{C}, \mathfrak{B}, \alpha)$			$\mathfrak{f}(\mathfrak{e}_1(\mathfrak{C}, \mathfrak{B}, \alpha), \mathfrak{B}, \alpha)$
$\mathfrak{e}_1(\mathfrak{C}, \mathfrak{B}, \alpha)$		E	$\mathfrak{C}$

This table defines an *m*-configuration $\mathfrak{e}(\mathfrak{C}, \mathfrak{B}, \alpha)$ which will erase the first occurrence of the symbol α on the tape and then move to *m*-configuration $\mathfrak{C}$. If no occurrence of α is found, the machine moves to *m*-configuration $\mathfrak{B}$. The machine first moves directly to an *m*-configuration defined by the skeleton table for the *m*-function $\mathfrak{f}$, which finds the first occurrence of α on the tape. If the symbol is found, the machine will moved to the *m*-configuration specified by the first parameter of $\mathfrak{f}$; this is a new *m*-configuration $\mathfrak{e}_1$ which will erases the symbol before moving to the 'success' *m*-configuration $\mathfrak{C}$.

Although the syntax used here for nested *m*-function applications is identical to the standard functional notation, it is worth noticing that the informal meaning of Turing's use of the notation differs from conventional usage. In a functional expression of the form $\phi(\psi(x))$, the function $\psi(x)$ is evaluated first, and its value used in the evaluation of ϕ. In Turing's expression $\mathfrak{f}(\mathfrak{e}_1(\mathfrak{C}, \mathfrak{B}, \alpha), \mathfrak{B}, \alpha)$, however, the effect is that the computation denoted by the *m*-configuration $\mathfrak{e}_1$ takes place *after* that denoted by $\mathfrak{f}$.

The *m*-configuration $\mathfrak{e}(\mathfrak{C}, \mathfrak{B}, \alpha)$ will erase the first occurrence of α on the tape, but if all occurrences of αs are to be deleted, this operation needs to repeated until none remain or, in other words, until it fails. Turing gave the following recursive definition of an *m*-configuration to achieve this [Turing, 1936, p. 237].

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{e}(\mathfrak{B}, \alpha)$			$\mathfrak{e}(\mathfrak{e}(\mathfrak{B}, \alpha), \mathfrak{B}, \alpha)$

The *m*-configuration $\mathfrak{e}(\mathfrak{B}, \alpha)$ will erase all occurrences of α from the tape and then go to state $\mathfrak{B}$. It is important to note here that there are two distinct *m*-functions in this table, both denoted by $\mathfrak{e}$, and distinguished only by the fact that one takes two and the other three parameters. $\mathfrak{e}(\mathfrak{B}, \alpha)$ first moves to the *m*-configuration $\mathfrak{e}(\mathfrak{C}, \mathfrak{B}, \alpha)$, defined in the previous table; this will delete the first occurrence of α. If this succeeds, the next *m*-configuration will again be $\mathfrak{e}(\mathfrak{B}, \alpha)$, and the process will repeat to delete the second and subsequent occurrences of α. Eventually, there will be no more αs on the tape and the deletion will be unsuccessful, in which case the machine will move to state $\mathfrak{B}$.

Free variables

The final syntactic feature of the machine table notation provides a way of passing the value of the currently scanned symbol to the final *m*-configuration without having to name it, by allowing free symbol variables to appear in the second column of the tables. In the table below, the *m*-configuration $\mathfrak{pe}$ prints the symbol β at the end of the tape and then goes to the *m*-configuration $\mathfrak{C}$. This operation is then used in an *m*-configuration $\mathfrak{c}_1$ which copies the currently scanned symbol at the end of the tape [Turing, 1936, p. 237].

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{pe}(\mathfrak{C}, \beta)$			$\mathfrak{f}(\mathfrak{pe}_1(\mathfrak{C}, \beta), \mathfrak{C}, ə)$
$\mathfrak{pe}_1(\mathfrak{C}, \beta)$	Any	R, R	$\mathfrak{pe}_1(\mathfrak{C}, \beta)$
	None	$P\beta$	$\mathfrak{C}$
$\mathfrak{c}_1(\mathfrak{C})$	β		$\mathfrak{pe}(\mathfrak{C}, \beta)$

In the first two lines, β is a parameter (or bound variable) and the value supplied when the row is called will be substituted in the remainder of the row. In the line defining $\mathfrak{c}_1$, β is free: the effect is that it will temporarily be bound to the scanned symbol, whatever that is, and that symbol will be supplied as a parameter to $\mathfrak{pe}$. Turing explained this as follows:

> The last line stands for the totality of lines obtainable from it by replacing β by any symbol which may occur on the tape of the machine concerned. [Turing, 1936, p. 238]

In other words, the line defining $\mathfrak{c}_1$ can be thought of as a shorthand for the lines

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{c}_1(\mathfrak{C})$	0		$\mathfrak{pe}(\mathfrak{C}, 0)$
$\mathfrak{c}_1(\mathfrak{C})$	1		$\mathfrak{pe}(\mathfrak{C}, 1)$
...			

where there is exactly one line for each symbol used by the machine.

At this point the question arises whether every skeleton table written using the abbreviations and conventions that Turing has introduced can be represented by a table in the unextended notation. Turing asserted this, but did not provide a proof. He considered the features of the extended language to be convenient abbreviations, stating that "[s]o long as the reader understands how to obtain the complete tables from the skeleton tables, there is no need to give any exact definitions in this connection" and "a table can always be put in this [simple] form by introducing more *m*-configurations" [Turing, 1936, p. 236, 239].

2.7 Universal machines

The *m*-configurations of a Turing machine represent the "states of mind" [Turing, 1936, p. 250] of a human computer, and encode information about the progress of a computation and the steps to be carried out next. Although humans can carry out familiar and simple algorithms mentally, more complicated processes require written instructions which are then followed in a step-by-step manner. Provided the computer or clerk can perform individual steps in the process and keep track of the next instruction to be carried out, this procedure is just as effective as memorizing the instructions.

The act of following instructions is itself a clerical task, however, and this raises the question of whether it is itself mechanizable. Turing called a machine which could follow the instructions expressed in a machine table *universal*, and answered this question in the affirmative by giving an explicit table for a universal machine. In the context of Turing's paper, this is "a single machine which can be used to compute any computable sequence" [Turing, 1936, p. 241]; more generally, it is a demonstration that the process of following explicitly given instructions is itself a mechanical, effective procedure. As Turing later put it, "we should consider the machine as doing something quite simple, namely carrying out orders given to it in a standard form which it is able to understand" [Turing, 1946, p. 21].

Turing began the construction of the universal machine by formalizing the simple form of machine table to which he claimed all tables could be reduced. This format is referred to as the *standard* *form* of a table. It is assumed that an enumeration $q_1, \ldots, q_R$ of the *m*-configurations used in the table is given, and also an enumeration $S_0, \ldots, S_m$ of the symbols which can appear on the tape. A machine table in standard form consists of a number of lines each of which has one of the following forms:

$q_i S_j S_k L q_m$
$q_i S_j S_k R q_m$
$q_i S_j S_k N q_m$

Here q_i and S_j denote the initial *m*-configuration and scanned symbol, and q_m and S_k the final state and symbol. L, R and N represent the three basic operations of moving one square to the left, one square to the right, or staying in the same position on the tape. Each line defines the behaviour of the machine when the current *m*-configuration is q_i and the scanned symbol is S_j. When the machine reaches this configuration in the course of a computation, the following events will take place. The symbol S_k will be written to the scanned square; if S_j and S_k are the same, the effect is that the symbol is unchanged, but this form of description allows this case to be subsumed into the case where a new symbol is written. The machine then moves one square to the left or right, or stays put, and the final *m*-configuration is q_m.

For an example of standard form, consider the first example table presented by Turing. It has four *m*-configurations and uses three tape symbols: let S_0 represent a blank square, S_1 the symbol 0 and S_2 the symbol 1. Assuming that the tape contains the symbol S_0 in every square at the beginning of the computation and starts in the *m*-configuration q_1, the following table defines a machine which prints the unending sequence $010101\ldots$ on alternate squares of the tape.

$q_1 S_0 S_1 R q_2$
$q_2 S_0 S_0 R q_3$
$q_3 S_0 S_2 R q_4$
$q_4 S_0 S_0 R q_1$

A universal machine must be able to examine the table of another machine. This can be achieved if it is defined how a machine table can be represented on the tape of the universal machine. For this purpose, Turing defined a further representation of machine tables, which he called their *standard descriptions*.

A standard description of a machine is an encoding of a table in standard form into the specific set of symbols used by the universal machine. Turing's universal machine uses the symbols A, C, D, L, N, R and ; to represent the standard form of tables. The *m*-configuration q_i is represented by the symbol D followed by i occurrences of A; the symbol S_j is represented by the symbol D followed by j occurrences of C; L, N and R represent themselves and the lines in the standard form table are separated by ;. The standard description for the table given above in standard form would therefore be:

$$DADDCRDAA; DAADDRDAAA; DAAADDCCRDAAAA; DAAAADDRDA$$

Standard descriptions are one-dimensional sequences of symbols, and therefore comprise an encoding scheme which enables machine tables to be represented on a tape, and hence as data that can be manipulated by other machines. Like Kleene and Church, Turing went one step further and produced an arithmetization of his notation, associating a unique natural number with each machine table. This representation was only used for theoretical purposes, however, and did not form part of the definition of the universal machine.

2.8 The concept of a formal language

We now turn to the second area of logical research to be considered, namely the development of a theory of a formal languages themselves. The similarity between the syntactic operations involved in the definition of a formal notation and the recursive, or 'inductive', definitions used in mathematics was familiar to logicians. For example, in an early paper Church commented on an informal explanation of the structure of the well-formed formulas of his system, saying that "[t]his is a definition by induction" [Church, 1932, p. 352]. Gödel made this idea precise by employing the technique of arithmetization and defining syntactic properties as recursive number-theoretic functions. As described above, a number of logicians subsequently provided an explicit arithmetization of their notations and commented on the theoretical role of the encoding. Kleene, for example, wrote that "[t]he operations on symbols which occur in the computation have a similarity to ordinary recursive operations on numbers" [Kleene, 1936a, p. 727], and Church referred to "the now familiar remark that, in view of the Gödel representation and the ideas associated with it, symbolic logic in general can be regarded, mathematically, as a branch of elementary number theory" [Church, 1936, p. 94, footnote 8].

The insight provided by the technique of arithmetization made possible the development of a mathematical theory of formal languages, a development associated particularly with the work of Tarski and Carnap [Tarski, 1933, Carnap, 1937, Carnap, 1939, Carnap, 1942]. This section briefly describes the major features of this account of formal languages, which served as the framework within which programming languages were subsequently studied.

Object language and metalanguage

An important preliminary distinction was drawn between the language under investigation, the *object language*, and the language in which the investigation is carried out, the *metalanguage*. This distinction originated in Hilbert's notion of metamathematics: for example, Gödel wrote of a certain formula that it "is merely a *metamathematical description* of the undecidable proposition" [Gödel, 1931, p. 149, footnote 13], and went on to explain how the proposition itself could be written down. In the technical parts of his 1931 paper, Gödel distinguished the formal language P from the metamathematical notation used to define recursive functions over the expressions of P by using distinct logical symbols for the two cases.

For Tarski, the distinction between object and metalanguage was motivated by the fact that not every language possessed "terms belonging to the theory of language" [Tarski, 1933, p. 167], and so in general it would not be possible to discuss the syntax, say, of a language in that language itself. Carnap stated explicitly that "we are concerned with two languages: in the first place the language which is the object of our investigation—we shall call this the **object-language**—and, secondly, with the language in which we speak *about* the syntactical forms of the object-language—we shall call this the **syntax-language**" [Carnap, 1937, p. 4].

This distinction raised the possibility of the need for an unending hierarchy of metalanguages. Arguing against this, Carnap emphasized that arithmetization provided a general technique whereby a language rich enough to contain the theory of the natural numbers could, without fear of contradiction, function as its own syntactic metalanguage [Carnap, 1937, p. 53].

Different metalinguistic resources were needed for different purposes. For the purposes of logical syntax, Carnap only needed a metalanguage which was capable of describing the syntax of the object language, hence his use of the more specific term 'syntax-language'. Tarski's semantic investigations, however, required the ability to describe both the syntactic form of object-language sentences and also their meaning. Tarski therefore demanded further that a metalanguage be expressive enough to contain a translation of each expression of the object language: "the fact that the metalanguage contains both an individual name and a translation of every expression . . . of the language studied will play a decisive part in the construction of the definition of truth" [Tarski, 1933,

p. 172].

Syntax

The first aspect of the metatheory of logic to be addressed in detail was that of syntax. Formal languages were originally characterized by the fact that their structure and properties could be discussed without any reference to the meaning of expressions in the language. Tarski wrote that formalized languages were those which could be described using "only those concepts which relate to the form and arrangement of the signs and compound expressions of the language" [Tarski, 1936b, p. 403], and Carnap stated that " [a] theory, a rule, a definition or the like is to be called *formal* when no reference is made in it either to the meaning of the symbols (for example, the words) or to the sense of the expressions (e.g. the sentences), but simply and solely to the kinds and order of the symbols from which the expressions are constructed" [Carnap, 1937, p. 1, italics in original].

Syntax was therefore understood to be the theory of the purely structural properties and relationships of the expressions of a language. Carnap thought of the syntactical description of a language as containing two aspects:

> The rules of the calculus determine, in the first place, the conditions under which an expression can be said to belong to a certain category of expressions; and, in the second place, under what conditions the transformation of one or more expressions into another or others may be allowed. ... The two different kinds of rules are those which we have previously called the rules of formation and transformation—namely the syntactical rules in the narrower sense ..., and the so-called logical laws of deduction.... [Carnap, 1937, p. 4]

The rules of formation described the structure of the expressions of a language, and defined which expressions constituted meaningful sentences or formulas. Tarski characterized the important aspects of the rules of formation by two properties:

> (α) for each of these languages a list or description is given in structural terms of all *signs with which the expressions of the language are formed*; (β) among all possible expressions which can be formed with these signs those called *sentences* are distinguished by means of purely structural properties. [Tarski, 1933, p. 166, italics in original]

Property (α) specified that the alphabet of the language must be given, in purely structural terms. The expressions in a language were all sequences, grammatical or not, of signs in the alphabet, and Tarski gave an axiomatization of the operation of concatenation by means of which these sequences are formed. The sentences of the language were those expressions which were "well-formed", and property (β) asserted that it must be possible to distinguish the well-formed expressions, or sentences, within the complete class of expressions purely in terms of their structural properties, or in other words, without referring to any interpretation of those expressions.

As the quotation above indicates, Carnap seems at this time to have thought of deduction as being an intrinsic part of a formal language. Tarski was more circumspect, writing that "formalized languages have hitherto been constructed exclusively for the purposes of studying the *deductive sciences*", but for him too the relationship of entailment between sentences was of particular interest. The study of proofs had made it apparent that much of the notion of entailment could be captured in formal terms, and so dealt with as part of logical syntax. Tarski summarized the way in which this was typically done in two further properties. Property (γ) stated that a set of sentences called *axioms* should be specified in purely structural terms, and property (δ) that a number of *rules of inference* should be specified by which sentences could be transformed into other sentences.

Semantics

Gödel's incompleteness result had shown that the syntactic notion of validity or provability did not in general coincide with the notion of truth. Tarski had subsequently given a 'semantic' definition of truth, so called because it was built upon a relationship of denotation, or designation, between the terms of a language and the objects and properties in a suitable domain of interpretation. Building upon this definition, Morris and Carnap defined semantics as the study of the "relations between the expressions of [a language] and their designata" [Carnap, 1939, p. 6].

Tarski's definition of truth was taken by Carnap as an exemplary semantic definition. Tarski required that "the sense of every expression is uniquely defined by its form" [Tarski, 1933, p. 165-6]. One important aspect of this requirement is *compositionality*: the meaning of a whole expression is given as a function of the meaning of its parts, and the way in which the meaning of a whole expression is arrived at depends solely on the way in which it is syntactically constructed. For example, Tarski's definition of satisfaction is based on the syntactic structure of sentences: for each clause defining how a sentence can be constructed from simpler sentences, there is a matching clause defining satisfaction of the resulting sentence in terms of the satisfaction of the simpler sentences [Tarski, 1933, p. 193].

The structure of the metatheory

Tarski's definition of truth established a distinction between purely syntactic accounts of formal languages and a semantic treatment. This distinction was applied and generalized by Charles Morris as part of the theory of signs. Morris based this theory on the process of *semiosis* which involved a three-way relationship between a "sign vehicle", a "designatum" and an "interpretant", the "effect on some interpreter in virtue of which the thing in question is a sign to that interpreter" [Morris, 1938, p. 3]. Considering the three terms in this relationship, Morris defined semantics as the study of "the relations of signs to the objects to which the signs are applicable" and pragmatics as the study of "the relation of signs to interpreters" [Morris, 1938, p. 6]. Noting that signs normally occur in the context of a system of related signs, "syntactics" was further defined as the study of the "relations of signs to one another in abstraction from the relation of signs to objects or interpreters" [Morris, 1938, p. 13].

Carnap restated Morris's categorization for the specific case of the analysis of language, distinguishing between "the action, state, and environment of a man who speaks or hears, say, the German word 'blau' ... the word 'blau' as an element of the German language ... [and] a certain property of things, viz., the color blue, to which this man ... intends to refer" [Carnap, 1939, p. 4]. Carnap suggested that all three aspects, which he called "pragmatics", "semantics" and "logical syntax", should be studied as part of a theory of language.

2.9 The relationship between Turing's work and logic

Turing's 1936 paper has often been described as foundational to the subsequent development of computing and computer science. This chapter has taken a different perspective, and has emphasized the close relationships between it and other work in mathematical logic. This section tries to do justice to the originality of Turing's work, not however by describing it as a 'precursor' or 'anticipation' of later work, but by applying to it Pickering's scheme for conceptual innovation, described in Chapter 1.

The first stage in Pickering's schema is *bridging*, the discovery of a way of using the concepts and results of one field to guide the development of some new area. Turing wanted to take seriously the idea of computation by machines as a basis for an analysis of computability, and his problem was how to bridge the gap between the existing discipline of mathematical logic and the more concrete world of machines. He achieved this by the introduction of the machine table notation, which provided textual equivalents of potentially physical machines. By treating machine tables as texts in a formal language, it became possible for Turing to apply the well-developed resources of formal logic to the study of machines.

Bridging is followed by a stage which Pickering describes as *transcription*, in which ideas and techniques from the existing domain are applied, in a more or less routine manner, to the new area. Illustrating this, Section 2.6 showed how the syntactic notions of variables, functions and recursion were applied to machine tables, and Section 2.7 how Gödel's technique of arithmetization was used in the definition of the universal machine. Because of the differences between machine tables and conventional languages of logic such as Gödel's language P, however, Turing's approach differs in detail from Gödel's.

The most significant difference stems from a basic semantic difference between the languages. The atomic formulas in conventional languages are formed by applying a predicate to one or more terms, which are in turn made up from variables and constants combined with function applications. Semantically, terms denote objects in some domain of interpretation, and atomic formulas make assertions which can be true or false. More complex formulas can be built up using truth-functional connectives and quantifiers, and these formulas also represent assertions and are evaluated for their truth value.

Turing's machine table language is quite different. It contains three kinds of terms, representing the symbols that can be written on the tape, the *m*-configurations, and the primitive actions that can be taken by a machine, and complex terms can be built up using *m*-functions. However, there are no predicates, and hence no atomic formulas and no way of expressing an assertion or a judgement in a machine table. At this point, the transcription of ideas from mathematical logic breaks down and reveals the need for what Pickering calls *filling*, or the creation of new material to fill out or complete the new theory.

The question is, how should the semantics of a machine table be understood? Turing wrote of his first example, "[t]he behaviour of the machine is described in the following table" [Turing, 1936, p. 233], and his informal annotations to subsequent tables take the form of a description of what the machine would do when in the appropriate *m*-configuration. This suggests an interpretation where machine tables and the lines comprising them are taken to be expressions denoting, or possibly making assertions about, machine behaviour. Apart from informal descriptions, however, Turing gives no characterization of machine behaviour separate from the machine tables themselves, and this interpretation is therefore left undeveloped.

Later in the paper, in the discussion of the universal machine, Turing makes use of an alternative interpretation. Tables are translated into standard descriptions in order to be written on the tape of the universal machine; this is a purely syntactical transformation, however, which we can assume leaves the semantics of the table unchanged. Turing then writes that "[t]he S.D. [standard description] consists of a number of instructions, separated by semi-colons" [Turing, 1936, p. 243]. A very similar interpretation is suggested by Post, who spoke in terms of a "set of directions" to be given which would determine the operations performed by a worker [Post, 1936].

The use of the word 'instruction' to describe lines in machine tables suggests an interpretation in which lines are treated not as denoting terms but as commands, as linguistic forms in the imperative, not the indicative, mood. If machine tables are understood in this way, however, the question arises of how to treat them semantically: commands are not naturally understood as making assertions, so the kind of interpretation used for indicative sentences does not seem to apply in this case.

Speaking informally, what we do with commands is to obey them, or carry them out, actually performing the actions that they specify: Post's notion of a worker obeying a set of directions captures this intuition. For the instructions contained in a machine table, we are interested in the computation that would be performed and the results obtained by the machine whose behaviour was described by the table. However, this is precisely what the universal machine does. Given a machine table, the universal machine will go through the steps involved in obeying the instructions in the table, and in so doing generate precisely the results that the original machine would produce. From this perspective, the universal machine defines a formal semantic account of the meaning of machine tables. This is not a semantic account of the denotational form assumed by Carnap and Morris, but one appropriate to the imperative nature of machine tables.

Given this, the relationship between Turing's work and Gödel's can be presented by means of the following structural analogy between their systems. Both begin by defining a formal language, in Gödel's case the language P and in Turing's case the machine table notation. The expressions of the language are then coded by mapping them into the domain of interpretation of the language. For Gödel, P is a formal language for number theory, so by means of arithmetization its formulas are encoded as natural numbers. Turing's notation describes the behaviour of machines computing with symbols on a tape, so Turing encodes machine tables as standard descriptions which can be written on a tape, thus making them accessible to other tables in the same way that an arithmetized formula of P is accessible to other formulas. Finally, the encoding is used to express metalinguistic properties of the object language in the object language itself. For Gödel this involved the definition of recursive functions, which are known to be expressible in P. For Turing, this must involve the definition of appropriate machine tables: the example he gave was the universal machine which, as argued above, defines a semantic account of machine tables in terms of what is involved in following the instructions they contain.

Morris wrote in 1938:

> [Logical syntax] has limited its investigation of syntactical structure to the type of sign combinations which are dominant in science, namely, those combinations which from a semantical point of view are called statements, or those combinations used in the transformation of such combinations. Thus on Carnap's usage commands are not sentences . . . [Morris, 1938, p. 16]

In contrast, this chapter has argued that Turing's work of 1936 can be understood as extending the domain of mathematical logic by introducing the machine table notation as a formal, textual representation of commands. Further, it has shown how Turing applied and generalized existing work in logic, particularly that of Gödel, to give, in the form of the universal machine, a formalization of the semantic notion of obeying a command.

Chapter 3

Logic and the invention of the computer

During the 1930s, the concept of computability was being investigated from both theoretical and practical points of view. As described in the previous chapter, mathematical logicians had succeeded in giving a precise logical characterization of the informal notion of effective computability, the culmination of a long investigation in how to make logical procedures effectively calculable. At the same time, Zuse and Aiken were beginning projects which would lead to the construction of large-scale automatic computing machines.

These investigations were largely independent of each other. In particular, it appears that Zuse and Aiken were principally motivated by the desire to avoid having to perform long calculations by hand, and were, at least initially, ignorant of theoretical developments in logic. For example, Zuse invented what he thought was a novel notation to describe certain features of the design of his machine, only to be later told that he had in effect rediscovered the propositional calculus [Zuse, 1993, p. 46].

In contrast, at least some logicians were aware of the importance of practical computation. By the 1930s, calculating machines and punched card machinery were extensively used in industry and commerce, and techniques for organizing large-scale calculations were widely known. Turing used the example of a human performing complex calculations to motivate the design of his abstract machines, and it has been suggested that his use of the machine concept in the definition of computability may have been partly motivated by his awareness of the processes of mechanization employed, for example, in the British Civil Service [Agar, 2003].

In the following decade computing machinery developed extremely rapidly. A major cause of this was the extensive computational requirements of the Second World War, not only in traditional areas of applied mathematics but also notably in cryptanalysis. By 1950, machines such as those of Zuse and Aiken were becoming obsolete, partly because of their limited computational capacity, but also because the design principles on which they were based had been superseded. Of particular significance is the adoption of the so-called *stored program* design: unlike the machines of Zuse and Aiken, which read their instructions from an external medium such as punched cards or paper tape, later machines stored their instructions internally, in the same medium that was used to store the data being operated on. The stored program concept was evolved by a group working at the University of Pennsylvania on a large electronic machine, the ENIAC, and was first described in a proposal describing its successor, the EDVAC [von Neumann, 1945].

During this period the logical and practical approaches to computation became increasingly entwined. Turing was extensively involved in the practical development of various machines in Britain, and in the United States the mathematician John von Neumann was from 1944 onwards centrally involved in the planning and construction of new machines, starting with the EDVAC. After 1950, it became a commonplace to describe computers as being instantiations of Turing's concept of a universal machine, and stored program computers are to this day described as being based on the 'von Neumann architecture'.

These observations raise the question of the extent to which theory and practice interacted in the development of computing technology. A widely accepted account sees the adoption of the stored program design as being the crucial innovation, and one in which theory played a crucial role. Michael Mahoney has expressed this view clearly:

> it is really only in von Neumann's collaboration with the ENIAC team that two quite separate historical strands come together: the effort to achieve high-speed, high-precision, automatic calculation and the effort to design a logic machine capable of significant reasoning. [Mahoney, 1988]

This image was reinforced by Eloina Peláez, who wrote that "[t]he development of the stored-program computer can be seen as the result of the coming together of two quite different traditions" [Peláez, 1999, p. 359]. In her account, the "two strands" had been separated by the increasing formalization of mathematics since the nineteenth century and were then forced back together by the practical demands of the war.

A number of writers have made the stronger assertion that the coming together of the two strands was a necessary precondition for the emergence of the computer in its modern form. For example, Stan Ulam, a mathematician who became an early computer user through his involvement with the

Manhattan project, wrote that “computer development became possible only by a confluence of at least two entirely different streams” [Ulam, 1980], and in his biography of Turing, Andrew Hodges describes Turing and von Neumann as “assembling the *necessary* ideas for the digital computer out of the conjunction of Hilbertian rationalism and Second World War technology” [Hodges, 1983, p. 556, emphasis added]. Forceful arguments in favour of this ‘confluence’ theory have also been made by the logician and computer scientist Martin Davis [Davis, 2000].

However, a widespread belief about the importance of logic to the practical development of the computer seems only to have emerged some time after the fact. In the mid-1950s, for example, the logician Hao Wang wrote that “Turing’s theory of computable functions antedated but has not much influenced the extensive actual construction of digital computers. These two aspects of theory and practice have been developed almost entirely independently of each other” [Wang, 1957, p. 63].

This chapter examines the interaction between theory and practice and the influence of logic in the development of the computer. A number of distinct claims have been made about this relationship. The first concerns the nature or essence of the computer, and asserts that the computer can best be characterized by its relationship with logic, as opposed, say, to its relationship with numerical analysis or electronic engineering. Davis put this position bluntly, stating that “a computing machine is really a logic machine” [Davis, 2000, p. xii].

A second claim concerns the causal role played by logic in the development of the computer, a role emphasized by Mahoney: “[a]s logic machines, the first stored-program computers . . . emerged as byproducts of theoretical inquiry into the nature and limits of logical thought” [Mahoney, 1989]. This idea was reinforced by Davis, who wrote as if the creation of the first computers was a relatively straightforward implementation of Turing’s abstract machine concept.

A third claim relates to the general-purpose nature of computers, their ability to be used not only for numerical computation, but for any task involving information processing. According to Davis, the general applicability of computers is attributable to Turing’s concept of a universal machine.

This chapter will examine these three claims in detail. It is convenient to start with the second claim, which also provides an opportunity to review the relevant historical material.

3.1 The origins of the stored program computer

Historians have traditionally located the origin of the computer in its modern form in work carried out at the Moore School of Engineering, part of the University of Pennsylvania, during the period 1943–1946 [Ceruzzi, 2001]. During this period, a team of electronic engineers and applied mathematicians, led by John Mauchly and Presper Eckert, designed and constructed an electronic calculator, the ENIAC [Goldstine and Goldstine, 1946]. This was not the first actual or proposed device to use electronics for automatic calculation, but the ENIAC project was on a far larger scale than its predecessors. It promised to remove a backlog in the calculations required in the development of new artillery weapons, and was supported financially by the Ballistics Research Laboratory (BRL) at the nearby Aberdeen testing grounds. The construction of the ENIAC demonstrated once and for all the feasibility of large-scale electronic computing devices.

The group soon recognized that there were several shortcomings in the design of the ENIAC, and during 1944 work started on a follow-up project [Stern, 1981]. Later that year, the ENIAC team came into contact with von Neumann, who joined the group as a part-time consultant. This appears to have been a highly fruitful collaboration, which led in 1945 to the writing of the *First draft of a report on the EDVAC* [von Neumann, 1945, hereafter ‘*Draft Report*’], an internal report describing features of the design of a proposed successor machine to the ENIAC. Although not intended for publication, this report was widely circulated, and is generally credited with defining for the first time the high-level design principles underlying virtually all computers subsequently built.

The ENIAC became operational at the beginning of 1946, and both it and the ‘von Neumann design’ were described in detail at a summer school held later that year at the Moore School [Campbell-Kelly and Williams, 1985]. Many of those attending this summer school were subsequently active in developing computers, including Maurice Wilkes from Cambridge whose EDSAC was very closely modelled on the machine described by von Neumann. The *Draft Report* therefore had an immediate and direct influence on subsequent computer developments.

Recent historical writing has been concerned to place these events in a wider context and, rather than describing a self-contained episode of technological innovation, has emphasized continuities within the wider history of computation. The history of pre-electronic calculating technology has been extensively described [Aspray, 1990a], and links between the office automation industry and the post-war computer emphasized [Agar, 2003]. However, even in this broader historiographical tradition, the events at the Moore School are seen as a watershed. In their history, entitled simply *Computer*, Campbell-Kelly and Aspray devoted a chapter entitled “Inventing the Computer” to the topic [Campbell-Kelly and Aspray, 1996]. Similarly, Ceruzzi’s *History of Modern Computing* dated the advent of the modern period to the completion of the ENIAC and the writing of the *Draft Report* in 1945 [Ceruzzi, 1998], and Ceruzzi later wrote that “the stored-program principle remains a valid focus for computing’s history” [Ceruzzi, 2001, p. 51].

The initial impetus for the Moore School work appears to have come from John Mauchly. Before the war, Mauchly was a professor of physics at a small college near Philadelphia. He was interested in meteorology, and in particular in the possibility of automating the very large calculations required in numerical meteorology. He explored the use of vacuum tubes to build electronic counters [Stern, 1981, p. 9], and in 1941 visited the University of Iowa and examined an electronic device intended to solve simultaneous equations developed by John Atanasoff. In the summer of 1941, Mauchly attended a training course in electronics at the Moore School, and subsequently joined the faculty in the autumn of 1941. While on the course he came into contact with Eckert, and succeeded in interesting him in the possible use of electronic technology for constructing very high speed calculating devices.

In 1942, Mauchly wrote a report entitled “The Use of High-Speed Vacuum Tube Devices for Calculating”, stressing the advantages to be gained from employing electronic technology to perform automatic calculation:

> There are many sorts of mathematical problems which require calculation by formulas which can readily be put in the form of iterative equations . . . a great gain in the speed of calculation can be obtained if the devices which are used employ electronic means for the performance of the calculation [Mauchly, 1942]

The report was submitted both to the Moore School and to the Army Ordnance Department, but little action was taken until 1943, when it came to the attention of Herman Goldstine. Goldstine was a mathematician with a background in ballistics who in 1942 was posted to the US Army Ordnance Department at the BRL. There had been liaison between the Moore School and the BRL since before the war.

A significant computational problem faced by the BRL was the timely production of firing tables for new artillery. The development of new weapons was proceeding at such a pace that the computational resources of the BRL could not keep up with the demand. Early in 1943, Goldstine came across Mauchly’s report, and became convinced that electronic technology could provide a solution to the BRL’s computational needs. A joint project was initiated in April 1943 between the Moore School and the BRL to develop the ENIAC, or Electronic Numerical Integrator And Computer.

The ENIAC was developed over the subsequent two years, being used to perform calculations for the Manhattan Project in the autumn of 1945, and was publicly demonstrated on February 14, 1946, when it was able to "compute the trajectory of a shell faster than the shell itself flies" [Burks, 1947, p. 756]. It was subsequently transferred to the Ballistics Research Laboratory and was extensively used until finally being decommissioned in 1955 [Fritz, 1994].

The ENIAC consisted of 40 distinct units. Of these, 20 were accumulators, each capable of storing a number and carrying out programmable operations of addition and subtraction on the stored number. Other units implemented more complex operations, such as multiplication or taking square roots, and the progress of a computation was controlled by a unit known as the 'master programmer'. It deviated significantly from other machines of the time, such as those of Zuse and Aiken, in the way in which instructions for a calculation were given to it. It did not seem feasible to use external paper tape or punched cards because the speed at which instructions would be read in would be so much slower than the electronic speed of computation that the advantages of computing electronically would be lost.

Instead, the ENIAC was manually reconfigured for each different problem it was applied to, a time-consuming and laborious process. This was recognized by its developers to be a problem, but it was felt that in the ENIAC's intended context of use the approach was tolerable, because it was assumed that the machine would be running the same program, to calculate firing tables, for long periods of time. This was noted in a progress report written at the end of 1943:

> No attempt has been made to make provision for setting up a problem automatically. This is for the sake of simplicity and because it is anticipated that the ENIAC will be used primarily for problems of a type in which one setup will be used many times before another problem is placed on the machine. [Anonymous, 1943]

The inconvenience of programming the ENIAC was soon recognized as being a significant limitation, however, and the desire to come up with a better method was one of the goals for subsequent development. In a document written in January 1944, Eckert described a device that was partly mechanical and partly electronic and that made use of magnetic storage devices [Eckert, 1944].

The device described consisted of a rotating shaft with a number of "discs or drums" mounted on it, of various types. So-called type (a) devices were to be capable of being magnetized and demagnetized quickly, thus providing "a method of storing, in some usable code, those characters or digits which must be used later or indicated". Type (b) devices would be engraved in some suitable way to "generate such pulses or other electronic signals as were required to time, control and initiate the operations required in the calculations". These descriptions suggest that different storage media were envisaged for data and program code, numbers being stored on the volatile type (a) discs with the type (b) discs playing the role of punched cards or paper tape in other machines, holding the program instructions. Eckert's proposal therefore clearly addressed the problem of reprogramming the ENIAC: the machine he described could be reprogrammed simply by changing the disc containing the program code. However, he went on to say:

> If multiple shaft systems are used, a great increase in the available facilities and for allowing automatic programming of the facilities and processes involved may be made . . . this programming may be of the temporary type set up on alloy discs or of the permanent type on etched discs. [Eckert, 1944]

The statement that "this programming may be of the temporary type", i.e. the type (a) discs, seems to imply that data and instructions could be stored in the same medium, but Eckert does not appears to view this as an intrinsic feature of his machine. It is hard to draw firm conclusions from such a short document, but it does not appear from this document that Eckert is thinking of a machine characterized by being based round a single, integrated store.

In 1944, the Moore School group was joined on a part-time basis by von Neumann. Although originally a pure mathematician, von Neumann became extensively involved in consulting activities to the US government concerned with various aspects of applied mathematics, an activity which during the war years occupied much of his time [Aspray, 1990b]. His consulting activities began in 1937 at the BRL, coincidentally enough, and after the outbreak of war quickly intensified. A significant involvement was with the Manhattan project at Los Alamos, where he advised on the shaping of explosions by the appropriate placement of explosive charges.

Many of these projects brought with them significant computational challenges, and von Neumann developed a serious interest in the current state of computational equipment. This interest was fostered by a visit to England in April 1943, when he visited the Nautical Almanac Office in Bath and helped to work out a program for an interpolation formula to be run on the punched card equipment being used there [Todd, 1974]. Following this visit, von Neumann wrote to Oswald Veblen that "I have also developed an obscene interest in computational techniques" [Aspray, 1990b, p. 27].

During 1943 and 1944, von Neumann carried out on behalf of the Manhattan project a survey of the existing technology for automatic computation. In January 1944, he contacted Warren Weaver, then head of the Applied Mathematics Panel of the Office of Scientific Research and Development, asking for information about the current situation. Weaver directed him to research groups at IBM and Harvard, Bell Labs and Columbia University, which von Neumann subsequently visited. None of these projects however seemed to be in a state enabling them to be of immediate use to Los Alamos. Von Neumann also gained first hand experience of the computational equipment currently in use at Los Alamos. In a letter of 1 August, 1944 to Robert Oppenheimer, von Neumann summarized his findings, and demonstrated a “deep and practical understanding of many of the important concepts of high-speed digital computation” [Aspray, 1990b, p. 33].

Curiously, it appears that despite this interest in automatic computation, von Neumann had not, before August 1944, either been told about or come across the ENIAC. Weaver had not mentioned the project, despite the fact that he undoubtedly knew of its existence. Various arguments have been put forward for this omission. In a book emphasizing the contributions made by Eckert and Mauchly, Nancy Stern has suggested that this was because the ENIAC project was held in low esteem by the scientific establishment, partly because neither Eckert and Mauchly had at this stage much of a scientific reputation [Stern, 1981]. Alternatively, it has been suggested that Weaver would not have known of any significant progress on the ENIAC, as he would have been unlikely to have read the first progress report, dated 31 December 1943, before responding to von Neumann’s enquiry in January 1944 [Aspray, 1990b, p. 35]

Whatever the reason, it appears that von Neumann did not know of the ENIAC project until he met Herman Goldstine, apparently by chance, and was told about Goldstine’s involvement in the development of an electronic calculating device [Goldstine, 1972, p. 182]. Aware of the limitations of the electromechanical technology he had been investigating, von Neumann was quick to appreciate the potential of the electronic speeds of computation promised by the ENIAC and its planned successor, and soon became involved with the ENIAC group as a consultant.

Von Neumann therefore brought to his work on the EDVAC proposal a detailed practical knowledge of current calculating technology, and a keen appreciation of the need in many areas of applied mathematics for greater computational capacity than was then available. This complemented the orientation of the ENIAC group towards the applications of automated calculation in areas such as ballistics and meteorology.

Progress reports on the EDVAC project give some insight into von Neumann’s contributions to the work. The first report, in March 1945, does not mention the stored program idea specifically, but does indicate what the group was expecting from the *Draft Report*:

> The problems of logical control have been analyzed by means of informal discussions among Dr. John von Neumann, . . . Dr. Mauchly, Mr. Eckert, Dr. Burks, Capt. Goldstine and others. . . . Points which have been considered during these discussions are flexibility of the use of EDVAC, storage capacity, computing speed, sorting speed, the coding of problems, and circuit design. . . . Dr. von Neumann plans to submit within the next few weeks a summary of these analyses of the logical control of the EDVAC together with examples showing how certain problems can be set up. [Eckert et al., 1945]

A second report, in September 1945, is more explicit about the historical background of the project, and claims that the stored program concept dates from Eckert's 1944 disclosure, though describing it in terms which go beyond what Eckert had originally written:

> . . . in January, 1944, a "magnetic calculating instrument" was disclosed. . . . An important feature of this device was that operating instructions and function tables would be stored in exactly the same sort of memory device as that used for numbers. . . . [Von Neumann] has contributed to many discussions on the logical controls of the EDVAC, has prepared certain instruction codes, and has tested these proposed systems by writing out the coded instructions for specific problems. Dr. von Neumann has also written a preliminary report in which most of the results of earlier discussions are summarized. [Anonymous, 1945]

The attribution of credit for the invention of the stored-program concept has proved to be very controversial. In retrospect, Turing's universal machine has been interpreted as embodying the notion, and some writers have therefore argued that credit ought ultimately to be given to Turing. The relationship between Turing and von Neumann and the extent to which von Neumann's contribution to the EDVAC was influenced by his knowledge of Turing's work are discussed further below.

In his autobiography, Zuse quoted diary entries made in 1937 and 1938 which appear to state, very briefly, the idea of holding both program and data in the same store [Zuse, 1993, p. 53]. However, Zuse did not build a machine based on these principles before 1945, and his work was in any case unknown to the EDVAC team.

Prior to 1945, other computer developers in the USA do not seem to have been aware of the stored program concept. Like Zuse, Aiken and George Stibitz's team at Bell Research Laboratories designed machines which were programmed by means of externally supplied programs. In 1940, Norbert Wiener described an automatic digital computing machine which would use electronic technology and contain data stored on a rewritable tape, in a manner very reminiscent of Turing's machines [Wiener, 1940]. Wiener's proposal was for a special purpose machine, however, and despite incorporating many of the features of the post-1945 machines, his proposal did not include the idea of the stored program.

Testimony from the members of the Moore School group themselves is mixed, and appears to be largely coloured by the subsequent split between von Neumann and Eckert and Mauchly. Goldstine and Burks, both with a mathematical and logical background, went to work with von Neumann at Princeton, and were clear that credit should be given to von Neumann. Eckert and Mauchly, on the other hand, contested this. Von Neumann himself appears never to have claimed credit for the idea, and in the EDVAC progress reports, as quoted above, he was consistently credited primarily for his work on logical control and coding.

The documentary evidence, summarized above, does not give an unequivocal answer. Drawing on Eckert's 1944 disclosure in particular, some writers have concluded that priority should be assigned to Eckert and Mauchly [Stern, 1980, Metropolis and Worlton, 1980]. This places a heavy weight on a rather thin text, however, and neglects the substantial differences between the disclosure and the later *Draft Report*. A reasonable compromise position, which seems to go as far as the evidence will allow, was given by Ceruzzi, who wrote that "Eckert and Mauchly had conceived of something like a stored-program principle by 1944, but ... it was von Neumann who clarified it and stated it in a form that gave it great force" [Ceruzzi, 1998, p. 22]. The relationship between logic and the stored program concept is discussed further in Section 3.4.

3.2 The early development of cybernetics

The previous section described the emergence of the stored-program computer against the background of research in the field of automated calculation, and found little evidence for the explicit involvement of logic in this process. One area in which an interest in logic and computers did come together in this period, however, was the emerging subject of cybernetics. This section briefly describes the origins of cybernetics, and in particular the involvement of Turing and von Neumann with the subject in the period before 1945.

Von Neumann first met Turing in Cambridge in 1935, and they later came into contact in Princeton, where Turing spent two years between 1936 and 1938 working with Alonzo Church [Hodges, 1983]. In his thesis, Aspray relates testimony from Stephen Rosser about this period, in which the interaction between Turing and von Neumann is described in the following terms:

> Even as early as his student days at Princeton, Turing argued vociferously that computing machines could be built which would adequately model any mental feature of the human brain. Von Neumann ... was attracted to Turing because of their common in-

> terest in mathematical logic. Turing's view on the computer and the brain was disputed by von Neumann, and the two discussed the issue on many occasions while Turing was completing his dissertation. This is purportedly what inspired von Neumann's interest in computing. Von Neumann and Turing separated when Turing returned to England, leaving both determined to build computers to test the possibility of mathematically modelling the human brain. [Aspray Jr., 1980, p. 147-8]

There are a number of anecdotal references testifying to von Neumann's continuing interest in Turing's work, and the high regard in which he held it. According to Stan Ulam, von Neumann spoke highly of Turing's work and "played with Turing machine-like mechanical descriptions of numbers" in the summer of 1938 [Aspray, 1990b, p. 178]. In a letter quoted by Brian Randell, Stan Frankel describes how in 1943 or 1944, while working at Los Alamos, von Neumann urged him to read Turing's 1936 paper. Frankel went on to say that an "essential role" played by von Neumann was "in making the world aware of these fundamental concepts introduced by Turing" [Randell, 1972, p. 10].

It is striking, however, that it appears to have been the analogy between Turing's machines and the brain that caught von Neumann's imagination. This analogy was also central to the work of Norbert Wiener who at the start of the war was investigating ways of improving the performance of anti-aircraft artillery. As Paul Edwards has described, this was of importance because of the increasing speed and complexity of modern technological warfare [Edwards, 1996]. Anti-aircraft batteries in particular were handicapped because human gunners were unable to track fast aircraft with sufficient accuracy. The solution adopted was to develop mechanisms which would automatically carry out some of the processing required, such that the resulting 'cyborg', a system consisting of both mechanical and human components, would achieve a level of performance beyond what each component was independently capable of.

In the course of this work, Wiener and his collaborators came to view the phenomenon of negative feedback as playing a crucial role. In 1942, in a paper summarizing their results, they argued that "a uniform behavioristic analysis is applicable to both machines and living organisms" [Rosenblueth et al., 1943, p. 22]. They outlined a hierarchical taxonomy of behaviour and argued that "[a]ll purposeful behavior may be considered to require negative feedback", a principle which was later commonly seen to encapsulate the central message of cybernetics [Wiener, 1948, Wisdom, 1951].

At about the same time, the psychologist Warren McCulloch and logician Walter Pitts described a model according to which aspects of the behaviour of a network of neurons, such as that found in the brain, could be captured in a logical calculus. McCulloch later described the work as having been directly inspired by Turing's paper on computability, claiming that they had viewed themselves as "treating the brain as a Turing machine" [McCulloch, 1948]. The paper concluded by claiming that neural nets equipped with a tape could "compute the same numbers as can a Turing machine", a result viewed as providing a "psychological justification of the Turing definition of computability and its equivalents" [McCulloch and Pitts, 1943, p. 129].

Von Neumann read this paper in 1943, apparently at the recommendation of Wiener and Bigelow [Aspray, 1990b, p. 180], and according to Bigelow, was "enormously impressed" with the work of McCulloch and Pitts [Aspray, 1990b, p. 313, note 23]. In 1948 he described the main result of the work as being the demonstration that behaviour which can be "defined at all logically, strictly, and unambiguously in a finite number of words can also be realized by . . . a formal neural network" [von Neumann, 1948, p. 412]. In other words, McCulloch and Pitts had linked the earlier work on computability with a plausible model of the brain, thus involving logic centrally in the emerging cybernetic framework.

Von Neumann immediately became involved in the area. An early result of this involvement was a conference organized by Wiener, von Neumann and Howard Aiken, held in Princeton in January, 1945. This meeting was described by Wiener as follows:

> The first day von Neumann spoke on computing machines, and I spoke on communication engineering. The second day Lorente de Nó and McCulloch joined forces for a very convincing presentation of the present status of the problem of the organization of the brain. In the end we were all convinced that the subject embracing the engineering and neurology aspects is essentially one, and we should go ahead with plans to embody these ideas in a permanent program of research [Wiener, 1945]

Plans to found a research institute subsequently came to nothing, however, and the most concrete outcome of the 1945 was a series of conferences held over the next few years under the auspices of the Macy foundation. These conferences were of great importance to the history of cybernetics but of less immediate relevance to the development of the computer, and so will not be discussed further here.

Because of constraints on travel during and immediately after the war, and the classified nature of his other work, Turing himself was only peripherally involved in these developments. However, he was visited by Wiener and McCulloch during the war, and spent the early months of 1944 in the US, visiting Claude Shannon at Bell Labs [Hodges, 1983].

3.3 Von Neumann's design for the EDVAC

By 1945, then, von Neumann was not only deeply involved in the field of automatic calculation, but was also playing a leading role in an informal group of scientists exploring analogies between computing machines and neuronal structures. This section illustrates how these two approaches were made explicit in the *Draft Report*, which presented not simply an electronic calculator, but rather a machine in which, to echo Wiener, "the engineering and neurology aspects were essentially one".

The report began by defining the purpose of the EDVAC, characterizing it as "a *very high speed automatic digital computing system*" [von Neumann, 1945, §1.1, emphasis in original]. The phrase "automatic computing system" is glossed as meaning "a device, which can carry out instructions to perform calculations of a considerable order of complexity—e.g. to solve a non-linear partial differential equation in 2 or 3 independent variables numerically" [von Neumann, 1945, §1.2]. Later, von Neumann wrote that "the device is primarily a computer" [von Neumann, 1945, §2.2]. The EDVAC was therefore designed to be a machine to automate mathematical calculation: the report nowhere suggests any wider uses for it.

Von Neumann next addressed the overall structure of the machine. The design is based on a small number of relatively high-level components, each identified with a single, clearly defined function. This contrasts strongly with the ENIAC, which was based upon a set of 20 identical accumulators, each capable of storing data, performing arithmetic operations, and controlling the sequencing of subsequent operations. In the EDVAC, by contrast, components would not be replicated, and each would have a single clearly defined functional role.

The first component that suggested itself was derived explicitly from the EDVAC's intended role as a calculator:

> Since the device is primarily a computer, it will have to perform the elementary operations of arithmetic most frequently. These are addition, subtraction, multiplication and division . . . It is therefore reasonable that it should contain specialized organs for just these operations. [von Neumann, 1945, §2.2]

In a discussion of an advanced electronic device, the use of the word 'organ' is striking. It introduces a metaphor that runs through the text of the draft report, that of the machine viewed as a body. As in the body, the 'organs' of the EDVAC are characterized primarily by the functions they perform. Observing that the list of operations that the machine should be able to perform directly is debatable, von Neumann concluded that “[a]t any rate, a *central arithmetical* part of the machine will probably have to exist, and this constitutes *the first specific part: CA*.” [von Neumann, 1945, §2.2, emphases in original]

The report then went on to consider how the course of a computation would be controlled:

> The logical control of the device, that is the proper sequencing of its operations, can be most efficiently carried out by a central control organ. If the device is to be *elastic*, that is as nearly as possible *all purpose*, then a distinction must be made between the specific instructions given for and defining a particular problem, and the general control organs which see to it that these instructions—no matter what they are—are carried out. The former must be stored in some way . . . the latter are represented by definite operating parts of the device. By the *central control* we mean this latter function only, and the organs which perform it form *the second specific part: CC*. [von Neumann, 1945, §2.3, emphases in original]

Again, this distinguished the EDVAC from the ENIAC, in which the instructions for particular problems were represented in the reconfigurable circuitry of the machine. Instead, the EDVAC design drew upon established practice in automatic computation. Babbage’s analytical engine had been designed to read the instructions for a computation from punched cards, and punched paper tape was used for this purpose by Zuse and Aiken.

In its portrayal of a machine which will be supplied with and will carry out instructions for different computations, this passage is evocative of Turing’s universal machine. Von Neumann does not draw attention to this analogy, however, and as the comparison with other contemporary machines makes clear, this was not a particularly innovative feature of the EDVAC design.

The report then continued by noting that “[a]ny device which is to carry out long and complicated sequences of operations . . . must have a considerable memory” [von Neumann, 1945, §2.4], using a term perhaps calculated to reinforce the machine/body metaphor. Among other requirements, it was noted that the instructions for the current calculation must be remembered as well as any intermediate results generated during the calculation, and this raised the question of whether different types of memory would be required:

> While it appeared that various parts of this memory have to perform functions which differ somewhat in their nature and considerably in their purpose, it is nevertheless tempting to treat the entire memory as one organ, and to have its parts as interchangeable as possible for the various functions enumerated above. . . . At any rate, the total *memory* constitutes *the third specific part of the device: M*. [von Neumann, 1945, §2.5, emphases in original]

This decision to have a single uniform memory is the innovation that makes the *Draft Report* the canonical source of the stored program idea. Again, although von Neumann did not comment on this, the design is reminiscent of Turing's universal machine, which used its single tape to store both the information required for, and that generated in the course of, a computation.

Von Neumann then refined the machine/body metaphor by making explicit reference to neurons in the central nervous system. The metaphor was used to motivate the introduction of the remaining components of the EDVAC design, the input and output devices I and O which transfer information from some external recording medium R to the internal parts (CA, CC and M) of the device.

> The three specific parts CA, CC (together C) and M correspond to the *associative* neurons in the human nervous system. It remains to discuss the equivalents of the *sensory* or *afferent* and the *motor* or *efferent* neurons. These are the *input* and *output* organs of the device . . . [von Neumann, 1945, §2.6, emphases in original]

This refinement aligned the internal parts CA, CC and M with the associative neurons, or in other words equated the central components of the EDVAC with the brain. This analogy was reinforced in the next section of the report where von Neumann turned to consideration of detailed structure of the three internal parts and the elements out of which they were built. Rather than moving straight to a description of this structure in terms of electronic components and circuits, however, he noted that computing devices were typically built out of elements which had two or more stable states, and could switch between states in response to various stimuli, commenting that "[i]t is worth mentioning, that the neurons of the higher animals are definitely elements in the above sense" [von Neumann, 1945, §4.2], thus reinforcing the connection between brains and computers.

To substantiate this claim, Von Neumann referred at this point, in the only technical reference in the *Draft Report*, to McCulloch and Pitts's abstract model of the neuron [McCulloch and Pitts, 1943]. Vacuum tubes were then presented as components which shared the properties of abstract neurons and were suitable for the construction of electronic computers. The details of the EDVAC's circuits in the remainder of the report are not presented in terms of tubes, however, as von Neumann wanted to separate issues of design from detailed considerations of electronics. Instead:

> The analogs of human neurons, discussed in 4.2-3 . . . seem to provide elements of just the kind postulated at the end of 6.1. We propose to use them accordingly for the purpose described there: As the constituent elements of the device, for the duration of the preliminary discussion. [von Neumann, 1945, §6.2]

In other words, the 'machine as brain' metaphor is now being presented as a substantial structural equivalence. It is being proposed that at an appropriate level of abstraction, an electronic computer can be described as being built out of the same sort of elements as the human brain.

Von Neumann later described this strategy as a form of axiomatization [von Neumann, 1948]. He described the problem of understanding the functioning of the brain as consisting of two parts: the first part would consider the physiological details of the neurons, the 'elements' of the brain, while the second would describe the overall organization of the elements, and the behaviour emerging from this organization. Linking the two parts is a abstract description of the elements, such as that given by McCulloch and Pitts. This should be framed in such a way as is convenient for building up the higher-level theory, while at the same time remaining faithful to the lower-level properties of the elements. Turing made a similar point, distinguishing between the roles of "mathematicians" and "engineers" in the design and use of automatic computers [Turing, 1946].

As this discussion has shown, then, the underlying cybernetic assumption of a fundamental analogy between natural organisms and machines was explicitly written into the first presentation of the architecture of the modern computer, and von Neumann was at pains to demonstrate that the new machines could be understood, at one level, as being artificial brains. The *Draft Report* was in this respect part of a much wider discourse in which advanced electronic machines, and computers in particular, were figured as 'giant brains'; this issue will be discussed further in Section 3.6.

The community of applied mathematicians who were the primary users of the early computers, and presumably very conscious of their limitations, were rather resistant to seeing computers as brains, however, and in von Neumann's later reports on computer design, coauthored with Burks and Goldstine, much less is made of the analogy [Goldstine and von Neumann, 1946, Burks et al., 1946]. It continued to play an important role in von Neumann's thinking, however, notably in the paper *The General and Logical Theory of Automata*, presented to an audience of cyberneticians in 1948 [von Neumann, 1948].

The role of logic in the *Draft Report*, then, is rather indirect. Rather than situating the EDVAC in an explicitly logical tradition, as Turing had his abstract machines, von Neumann presented the computer as simultaneously a calculator and an artificial brain. The connection between the stored-program computer and the Turing machine was left implicit, mediated by McCulloch and Pitts's work on the application of logic to the modelling of neuronal structures.

3.4 Logic and the stored program concept

Two conclusions can be drawn from this overview of the history of computers in the decade leading up to the writing of the *Draft Report*. Firstly, the report was grounded in a very practical programme of research into automatic computation, of which Mark I and the ENIAC are perhaps the two best-known examples. Secondly, the focus of this research was on the automation of numerical calculation. The seems to have provided the impetus which led several individuals to enter the field, and it was a tendency which was massively amplified by the military demands of the war. These computer was not, as Mahoney and Davis suggest, developed explicitly as a practical logic machine.

A weaker claim, however, would be that aspects of the design were adopted for reasons that were specifically logical, and it is the case that a number of aspects of the *Draft Report* have been characterized as being 'logical' in one way or another. For example, Aspray has described a sense in which von Neumann's influence could be described as involving logic:

> Von Neumann was interested in presenting a "logical" description of the stored-program computer rather than an engineering description; that is, his concern was the overall structure of a computing system, the abstract parts it comprises, the functions of each part, and how the parts interact to process information. [Aspray, 1990b, p. 40]

The abstract description of a computer's architecture as a set of functionally distinct subsystems is not original to the EDVAC, however. In particular, the idea of an architecture based around an extensive memory and a control or arithmetic unit operating on the contents of that memory appears to have occurred independently to several people. Babbage's Analytical Engine is perhaps the earliest example of such an architecture, containing a "store" and a "mill" which are functionally very similar to the EDVAC's memory and control, and in the 1930s Turing and Zuse independently came up with similar designs. Given this, it would be implausible to assert that this represented a specific influence of logic on the design of the EDVAC.

In the EDVAC progress reports, von Neumann's particular contributions are described as being in the area of logical control. The phrase 'logical control' is defined in the *Draft Report* as signifying "the proper sequencing of [the EDVAC's] operations" [von Neumann, 1945, §2.3], and referred to the control circuits that ensured that operations were carried out in the intended order and to the sequencing of operations required in particular problems. Elementary logic was certainly a useful tool in the design of such circuits, as Zuse and Claude Shannon had discovered [Shannon, 1938], but again, this does not point to a specific influence of logic on the design of the EDVAC.

Perhaps the strongest arguments in support of the influence of logic are based on the introduction in the *Draft Report* of the stored program principle. Before considering these arguments, however, it is necessary to distinguish three distinct concepts which this term is sometimes used to refer to. The first is the notion of universality, of a single machine which can be made to simulate the work of any other. A universal machine requires access to some representation of the machine it is simulating, but this representation does not need to be in the same medium as the data generated in the course of the simulation. Turing's arguments in 1936 would not have been affected if he had equipped the universal machine with a second tape to hold the coded table of the machine being simulated.

The stored program principle itself was defined above to be the decision to store the program and data of a computer in the same storage medium. Turing's universal machine does have this property, but this is not an essential part of its universality. This distinction is not always made clear: for example, Ceruzzi appears to conflate the two notions, writing that "the reason [for the importance of the computer] is the stored-program principle. A computer is not a single machine, but one of an infinite number of machines, depending on the software written for it" [Ceruzzi, 2001, p. 50].

The third concept is a consequence of the colocation of program code and data, namely the possibility for a program to manipulate its own code as if it were data, and hence to modify itself as it is being executed; this can be described as *self-modifying code*. Turing's universal machine makes no use of this possibility: the symbols in the squares on the tape which hold the table of the machine being simulated are left unchanged by the operations of the universal machine.

The earliest example of self-modifying code occurs in the *Draft Report*, which made use of a restricted form of self-modification known as *address modification*. This made it possible for a program to modify a instruction which retrieved a data value from a given location, say, so that the next time it was executed it would retrieve data from a different location. The use of address modification offered great advantages in the writing of programs which processed repeated sets of data such as vectors or matrices.

Different arguments for the influence of logic on the development of computers have been based on these concepts. For example, Davis argues that universality is what distinguishes the later machines from "earlier automatic calculators", writing that "[t]hese post-war machines were designed to be all-purpose universal devices capable of carrying out any symbolic process" [Davis, 2000, p. 185]. As a statement about the EDVAC, this assertion appears to be false: as discussed above, the

Draft Report makes it clear that the EDVAC was primarily designed as a numerical calculator. The relationship between universality and the computer is discussed further in Section 3.7.

Arguments related to the stored program principle are based on the decision in the *Draft Report* to equip the EDVAC with a single memory which would contain both the code of the program being executed and the data on which it was working. As noted above, this was a feature of Turing's universal machine, and it has been suggested that von Neumann's knowledge of Turing's work led directly to the incorporation of this feature in the design of the EDVAC.

Evidence for or against this position appears to be wholly circumstantial. On the one hand, von Neumann was aware of and admired Turing's work, and would certainly have been familiar with the design of the universal machine. On the other hand, there is no explicit mention of Turing in the *Draft Report* which, as discussed above, was concerned to link the new machine with the developing area of cybernetics rather than directly to logic.

One way to address the question is to ask why this particular design feature was adopted, when other aspects of Turing's design, such as the restriction on movement to adjoining tape positions, were not echoed in the EDVAC proposal. Explanations of this given by members of the EDVAC team did not stress the similarity of the two forms of stored information, numeric data and stored instructions, and give abstract or logical reasons for holding both in a single store; rather, the two forms were clearly distinguished and pragmatic reasons, based on engineering concerns, given for holding the two in a single store.

For example, in one of the Moore School lectures in the summer of 1946 Eckert explained aspects of the EDVAC's design [Eckert, 1946]. He identified a number of distinct uses for the memory of a computer, including the need to store data and instructions, and compared the characteristics of the memory required for these purposes. In particular, he noted that instructions must be available at high speed, so as not to hinder the progress of the computation. He then observes that different types of problem can have significantly different memory requirements, in terms of the relative amount of space required for these two purposes. Maximum flexibility and economy in construction could be obtained by combining both data and code in a single store, rather than providing separate memory components for each.

In another report dating from 1946 [Goldstine and von Neumann, 1946], Goldstine and von Neumann gave a similar account. They noted that the memory used to store instructions should provide the flexibility of media like paper tape, which could store an indefinitely large number of instructions and allow a machine to be easily reprogrammed, and also the ability to access these instructions at a high speed. They then noted that instructions on paper tape are already digitally encoded, and hence that there was no reason why they should not be stored in the same memory that is used for storing numerical data. The same point was also made in another report:

> Conceptually we have discussed above two different forms of memory: storage of numbers and storage of orders. If, however, the orders to the machine are reduced to a numerical code and if the machine can in some way distinguish a number from an order, the memory organ can be used to store both numbers and orders [Burks et al., 1946].

None of the writings originating from the Moore School group mention a specifically logical or theoretical rationale for the development of the stored program concept. It remains a possibility, of course, that the idea was suggested by von Neumann's knowledge of Turing's work, but even if that was so, its inclusion in the design was subsequently justified by practical, not theoretical, arguments; an influence from logic would not be sufficient, on its own, to explain this inclusion.

3.5 Turing and the ACE

The importance of the *Draft Report* lies in the concepts and approach it put forward, not in the specific details of the design presented. Later in 1945, when von Neumann was working on the code for a sorting and collating program, he assumed a slightly different machine structure and instruction set [Knuth, 1970], and the design for a machine at the Institute of Advanced Study, developed by von Neumann, Goldstine and Burks, differs from the EDVAC proposal in several significant ways [Burks et al., 1946], as did designs produced by others. Although important to the history of computer architecture, these alternative designs did not introduce anything new into the relationship between logic and the computer. The situation is slightly different, however, in the case of a design produced by Turing in 1946.

At the end of the war in 1945, Turing joined the National Physical Laboratory (NPL). He was given a copy of von Neumann's *Draft Report* and by the end of the year had produced a report outlining the design of a stored-program computer that he proposed the NPL should build, the Automatic Computing Engine, or ACE [Turing, 1946]. The ACE report is in many ways comparable in scope and ambition to von Neumann's *Draft Report*, and a comparison of the designs presented in the two reports is a good way of highlighting some of the characteristic features of each.

The influence of the *Draft Report* is apparent in the ACE report, and Turing recommends that they be read together. In many ways, the ACE is similar to the planned EDVAC. Both designs use mercury delay lines as the principal high-speed storage mechanism, and have a basic structure presented as a number of functionally distinct units, including a store, an arithmetic unit and a central control unit. Further, Turing uses von Neumann's abstract neuron-inspired notation for describing the logical circuits of the ACE, extending the notation in various ways for his own purposes. Nonetheless, the proposed ACE is in many ways quite different from the EDVAC, and it has been argued that these differences are not merely technical, but reflect a fundamental difference in the approaches of von Neumann and Turing [Carpenter and Doran, 1977, Peláez, 1999].

As discussed above, von Neumann in the *Draft Report* presented the EDVAC as fundamentally a calculator. Turing, by contrast, makes a very clear link between the ACE and his earlier analysis of computability as a formalization of certain more general practices:

> The class of problems capable of solution by the machine . . . are those problems which can be solved by human clerical labour, working to fixed rules, and without understanding . . . [Turing, 1946, p. 39]

He then goes on to list a number of possible applications of the machine, ranging from mathematical calculations to the solution of jigsaws and the playing of chess. In 1947, he was even more explicit and described the ACE as a "practical version" of the type of machine described in the 1936 paper [Turing, 1947, p. 107].

This difference in orientation is reflected in the design of the ACE in a number of ways, most noticeably in the way the machine is structured as a number of functional units. Turing initially describes the ACE as containing a memory, a logical control and a central arithmetic part in a manner virtually identical to von Neumann's description of the internal structure of the EDVAC [Turing, 1946, p. 21-22]. However, Turing's later treatment of the memory and the arithmetic unit is rather different from von Neumann's.

In both designs, it was proposed that the majority of the storage required would be provided by mercury delay lines. Delay line storage had the advantages of being cheap and relatively permanent, but the disadvantage of providing slow access to data because of the latency time involved as the data circulated round the delay line. More importantly, delay lines provided a passive storage medium, rather reminiscent of the tape in a Turing machine: data could be written to and retrieved from a delay line, but in order to add two numbers together, say, the numbers had first to be moved to a special location where the arithmetic circuits could gain access to them. Both designs therefore included provision for additional memory capability in order to get round this problem, but approached it in different ways.

In the EDVAC design, the arithmetic unit itself contained storage for three numbers, the two operands of the desired operation and the result. Instructions were provided to move data from the delay line storage into the arithmetic unit, and to move the result back to the delay lines. The arithmetic unit therefore functioned as a sort of ‘black box’: numbers were inserted into and the result extracted from it, but its internal workings were quite independent of the rest of the computer.

The design of the ACE is rather different. The ACE contained a number of “quick reference temporary storage units (TS)” [Turing, 1946, p. 22] in addition to the delay lines, but these were not associated with any particular functional unit of the computer. Rather, they were part of the memory, which was therefore divided between the delay line storage and the temporary storage. Operations were provided for moving data between the delay lines and the temporary storage.

Some of the TS locations were reserved for particular purposes. For example, Turing proposed that the arithmetic circuits should operate on the data found in TS 2 and TS 3 and store the results in TS 4 and TS 5. Similar conventions were proposed for some of the other TS units. Whereas the EDVAC could be described as having special purpose memory encapsulated within the arithmetic unit, the ACE by contrast did not have a specialized arithmetic unit, but rather a set of conventions governing the use of some locations in the general purpose memory. The ACE therefore maintained a strict distinction between memory and control reminiscent of the universal machine, whereas the EDVAC complicated this basic design with special-purpose units.

A second striking difference between the two reports concerns the way in which they viewed programs. Although its code supports looping programs and subroutines, the *Draft Report* conceived of a program as primarily a sequence of instructions invoking the basic arithmetic operations provided by the machine, in a way reminiscent of earlier machines such as Aiken’s Mark I. In contrast, Turing’s model of programming emphasized the idea that the basic operations required for the task being programmed should be defined as subroutines built from the ACE’s primitive instructions. This is the same procedure that he followed in the 1936 paper, in which machine tables to perform simple tasks, such as copying or erasing symbols on the tape, were first defined and then extensively reused. It was argued in Chapter 2 that this technique was derived from existing practice in the definition of recursive functions, and in the ACE report we can see this approach being carried forward into the sphere of practical computation. Turing recognized that this approach could be applied even to basic arithmetical operations: the arithmetic circuits in the ACE were therefore not viewed as fundamental components, as they were in the *Draft Report*, but rather as conveniences to increase the speed at which arithmetic could be carried out.

Thirdly, the two designs differed in the use made of the ability provided by the stored-program design to modify the code of a running program. In the *Draft Report*, instructions and data were clearly distinguished and only a limited form of instruction modification was allowed. Turing on the other hand allowed unrestricted operations to be performed on instructions, and referred more generally to the possibilities created by allowing the machine to write its own orders. As discussed above, this more free-wheeling approach would be enabled by the design of the universal machine, although in 1936 Turing made no mention of the possibility of instruction modification.

It appears plausible, then, that a number of features of Turing's design for the ACE were derived from his earlier theoretical work, and hence that the ACE could in a sense be described as more influenced by logic than the design of the *Draft Report*. In practice, however, the EDVAC design was vastly dominant. Like the EDVAC, the ACE was never implemented in precisely the form described in the initial report. The first machine completed at the NPL was the 'Pilot ACE', which was built on a smaller scale than Turing's proposed ACE and differed from it in a number of ways. The ACE itself was completed in the early 1950s, and the design principles it embodied were used in a small number of later machines. After the mid-1950s, however, the line of machines that directly made use of Turing's design died out.

This raises the question of why von Neumann's design, in some ways less logical than the ACE, proved so much more successful in practice. This question was addressed by Pelaéz, who downplayed 'internal' factors, such as the increase in complexity inherent in Turing's approach to programming, in favour of 'external' factors, and suggested that the primary reason for the greater success of the EDVAC design was its "instrumentality" [Peláez, 1999]. In emphasizing the provision of high-speed calculation von Neumann was addressing an immediate social need, and the EDVAC design was therefore picked up as a solution to a practical problem.

However, the ACE was as capable as the EDVAC of carrying out high-speed arithmetic, if not even faster, so this cannot be the whole story. Other relevant external factors include the wide circulation of the principles of the EDVAC design at the Moore School course in 1946, and the prestige lent to the whole project by von Neumann himself. A more internal consideration is that the *Draft Report* presented the EDVAC as a relatively straight-forward evolution from well-known machines such as the Mark I, in terms of application area, internal design and programming style. In contrast, the ACE was in many ways a more radical design put forward by a relatively unknown researcher, and the more 'logical' nature of its design does not appear to have been sufficient to ensure its widespread adoption. At the very least, this suggests that the connection between stored program computers and the universal machine was not widely appreciated in 1946: this point and its implications are considered in more depth in the following sections.

3.6 Giant brains

During the war, most research into computers was carried out in secret, and little information about the new machines was made publicly available. This situation changed rapidly after 1945, and it is possible to trace the reception and representation of computers in both the technical and more popular literature. Firstly, however, it had to be recognized that a significant development in computing technology had taken place. In January 1946, an article in the journal of the American Institute of Electrical Engineers discussed the "Impact of the War on Science", but made no mention of computing technology [Briggs, 1946]. Later that year, however, the journal *Mathematical Tables and other Aids to Computation* noted, in a review of a conference on 'Advanced Computation Techniques' held at MIT in October 1945, that:

> During the recent war there was a tremendous development of certain types of computing devices . . . these and other similar developments suggest that there will soon be available mechanical and electrical computing equipment which, in terms of speed and flexibility, will completely outdistance anything thought of before. [Archibald, 1946]

The new machines, in particular the ENIAC and Mark I, were widely reported in the press, and one prominent aspect of the coverage was the analogy drawn between high-speed calculators and the brain. During the war, various devices had been described as 'electronic brains', and the term was immediately applied to computing devices, as the following quotation from the psychologist Edwin Boring reveals:

> We have heard so much during the late war about electronic brains. The electronic computer on a range-finder figures the range and course and speed of a target, setting the fuses and aiming and firing the gun, all at a speed of which the human brain is incapable. There are now huge electronic mathematicians which will solve mathematical problems with a speed and accuracy and lack of fatigue that puts the mere headwork of the human mathematician out of the running. [Boring, 1946]

The press coverage of electronic computers in this period has been surveyed by Dianne Martin, who concludes that "during the critical early years of 1946 to 1948, the predominant characterization of the computer was as a mechanical or electronic brain or robot" [Martin, 1993, p. 130]. This phenomenon was not restricted to journalistic accounts, however. For example, Edmund Berkeley was deeply involved in the use and promotion of the early computers, and wrote one of the first books to provide a popular account of the new machines. He called the book "Giant Brains, or Machines that Think" [Berkeley, 1949].

Computer developers themselves often viewed such characterizations as inappropriately anthropomorphic. In a letter to the *Times*, Douglas Hartree opined that use of the term 'electronic brain' obscured the distinction between the thought and judgement involved in planning and setting up a computation and the labour of carrying it out and "ascribes to the machine capabilities that it does not possess" [Hartree, 1946a]. Mauchly, Turing and Aiken all gave newspaper interviews during 1946 and 1947 in which they were at pains to point out the limitations of the new machines [Martin, 1993, p. 129].

Such arguments were often supported by an appeal to a principle first enunciated by Babbage's collaborator Ada Lovelace: in Hartree's words, "[t]hese machines can only do precisely what they are instructed to do by the operators who set them up" [Hartree, 1946a]. Along with the related question of whether machines could think, this generated a substantial public discussion in the following years.

In Section 3.2, it was shown that the cybernetic conception of the computer, which was explicitly drawn upon by von Neumann and Turing, depended on the belief that, considered in the abstract as information processing machines, a strong identification could be made between the brain and the electronic computer. The description of computers as 'giant brains' can therefore be viewed, not as irresponsible anthropomorphism, but rather as a faithful representation of the cybernetic point of view.

A striking feature of the situation at this time is that it was the early machines, such as Mark I and ENIAC, which were described as revolutionary. Machines based on the stored program design did not become at all widely available until the early 1950s. At the point at which they entered public discourse, then, computers were not represented or understood as logic machines. Rather, the way in which they were described reflects the dual heritage of the machines that von Neumann emphasized in the *Draft Report*: as scientific devices for carrying rapid and autonomous calculations, and also as models or analogues of the brain.

3.7 Universal machines

Davis's third claim about the influence of logic on the development of computers draws attention to an important difference between modern computers and earlier calculating devices, namely that computers are intended to be, and are used as, universal computing devices rather than as purely numerical calculators [Davis, 1988, Davis, 2000]. The argument plays slightly on an ambiguity in the word 'universal', which can refer specifically to Turing's concept of a universal machine while at the same time suggesting less formally the very wide range of applications that computers are used for. This section will examine how stored-program computers came to be understood as 'universal', in Turing's sense, and the following section will consider the argument that this led to the use of computers in applications more general than calculation.

In Turing's 1936 paper, the word 'universal' is applied to a specific machine U which is able to simulate the behaviour of any other machine, given a suitable representation of the table of the machine to be simulated. U is only universal relative to the class of machines described in the paper, however. Presented with a description of a configuration of the ENIAC, say, it would be unable to simulate the resulting computation: for this purpose, a different universal machine would have to be defined.

A machine such as the EDVAC can also be described as universal in this sense. We can imagine specialized machines which have the same memory and repertoire of basic operations as the EDVAC, but whose control units are configured, like that of the ENIAC, to perform only the basic operations required by one particular computation. An EDVAC program serves as a representation of such a machine, in the same way that a machine table serves as a representation of a single Turing machine. The EDVAC itself, whose control unit is wired up in such a way as to interpret the program and reproduce the coded sequence of basic operations, is therefore acting in a manner precisely analogous to Turing's machine U.

As pointed out above, a computer does not have to incorporate a stored program in order to be universal in this sense. The argument of the previous paragraph could be applied to machines such as Zuse's Z3 or Mark I, and leads to the conclusion that they can also be described as universal, despite reading their programs from external storage devices.

In both the examples above, the machine being described as universal belongs to the same class of machines as those being simulated: U is itself a Turing machine, for example. It would be perfectly possible for a machine to be universal relative to the machines of a different class, however: for example, the ENIAC could be wired up to interpret standard descriptions of Turing machines, and in fact something similar to this was done in 1948 when it was reconfigured to operate as a stored program computer [Rope, 2007]. A condition of this being possible is that the machine doing the interpretation must be able to simulate the memory structure and basic operations of the machines being simulated, thus creating a 'virtual machine' whose behaviour it will then emulate. The notion of a virtual machine has found a number of applications, notably in the semantics of programming languages, as will be discussed in Chapter 5.

This consideration leads to another sense in which a machine can be described as 'universal', namely that it is capable of simulating the behaviour of any other machine whatsoever. It does not follow automatically that a machine which is universal in the first, technical, sense has this property. Rather, this is a consequence of an argument that the machine can perform, within the limits of finiteness, all the computations that can be performed by Turing machines, and hence, by the Church-Turing theses, all effectively computable processes. Early discussions of electronic computers tended not to distinguish these two senses of 'universal', nor to demonstrate the 'Turing-completeness' of the machines under discussion.

The characterization of stored-program computers as universal provides one way in which the claim that computers are 'really' logic machines can be understood. It is striking, however, that this characterization was not immediately obvious, and it was not until the early 1950s that it was common for computers to be described as universal machines. As Jon Agar has written, "the good historical question to ask is not 'Are stored-program computers universal Turing machines?' but 'Why have electronic stored-program computers been cast as universal, as general-purpose machines?' " [Agar, 2003, p. 7]. This remainder of this section will describe the process by which this took place, and suggest an answer to Agar's question.

Turing was quite clear about the connection between his earlier theoretical work and the practical post-war computer developments, and on a number of occasions he explicitly compared the ACE with the universal machine. In a report written in 1948, for example, he gave a classification of "logical" and "practical" computing machines, considering in some detail the question to what extent a finite machine such as the ACE could be considered to be universal [Turing, 1948].

Turing evidently imparted this understanding to his close collaborators. In 1946, a semi-popular account of the ACE project explicitly linked the construction of automatic computing machines with *On Computable Numbers*:

> Although this Harvard machine [the Mark I] is an independent and original development, the possibility of the construction of such machines, and, indeed, more elaborate ones, had already been foreseen in this country. Dr. A. M. Turing, a fellow of King's College, Cambridge, had written in 1936 a severely mathematical paper in which he had discussed the properties of such machines in connection with certain problems of mathematical logic, without considering practical problems of construction. [Department of Scientific and Industrial Research, 1946]

This report makes no mention of the universal property, however. In a review article written in 1948, Harry Huskey, who had worked at the NPL for a year during 1947/8, described a machine resembling the new machines but with an infinite memory as "absolutely general in the sense that it could be made to imitate any other computing machine merely by giving it the appropriate instructions" [Huskey, 1948, p. 976], citing *On Computable Numbers* in support of this claim. Confusingly, however, he later refers to computers as providing a "universal model" for a large class of physical experiments, as opposed to specific models such as wind tunnels. This would appear to refer to the distinction between digital and analogue calculation, rather than the more technical notion of universality.

Another long-term collaborator of Turing, Max Newman, made the connection explicit in 1948 in a discussion on computing machines held at the Royal Society:

> [a] universal machine is a single machine which, when provided with suitable instructions, will perform any calculation that could be done by a specially constructed machine . . . subject to this limitation of size, the machines now being made in America and in this country will be 'universal'—if they work at all; that is, they will do every kind of job that can be done by special machines [Newman, 1949, p. 271-2]

However, despite these statements, the connection between the new computers and the universal machine was not appreciated more widely. At the same discussion at which Newman made the statement quoted above, Maurice Wilkes described the EDSAC, a machine then under construction at Cambridge. He made no mention whatsoever of Turing's work, focusing instead on the expected influence of the EDSAC on scientific research [Wilkes, 1949].

A more detailed presentation can be found in the book *Calculating Instruments and Machines* published by Douglas Hartree in 1949 [Hartree, 1949]. In 1946 Hartree had travelled to the USA and made practical use of the ENIAC [Hartree, 1946b]. His book was based on a series of lectures given at the University of Illinois in 1948 and, as the title suggests, Hartree was primarily interested in the mathematical applications of computers.

Hartree referred to the computer designs of both von Neumann and Turing, and his presentation of the ideas underlying computers derived from them in a number of ways. For example, when introducing digital computing machines, he first considered their functional design very much in the style of von Neumann, even drawing the same analogy between the structure of computers and that of living organisms. He then motivated the particular design of the computer by referring to Turing's analogy with the procedures carried out by human computers [Hartree, 1949, p. 56-7]. Later, when giving a more detailed description of the structure of computers, he used the neuron-inspired notation of computing elements "introduced, in this context, by von Neumann and extended by Turing" [Hartree, 1949, p. 97].

Hartree did not, however, refer to Turing's 1936 paper, and appears not to have had a very clear notion of the concept of the universal machine. He described the problem caused by the need to set programs up manually on the ENIAC, and went on to suggest that this would be replaced by "a means by which the machine can set up for itself the connections required for the sequence of computing operations" [Hartree, 1949, p. 94]. Perhaps this phraseology was an attempt to make the concept accessible to a non-specialist audience, but it is striking that it does not make the point that a universal machine removes the need to alter any connections at all from one calculation to another.

Later, in the context of a discussion of whether machines work with decimal or binary numerals, Hartree commented that, with the exception of the UNIVAC, the proposed computers "work in the scale of two, though the A.C.E. is intended as a universal machine and will be able to be programmed to work in scale of ten—or any other scale—and this may also be the case for the others" [Hartree, 1949, p. 97]. This rather contorted sentence suggests on the one hand that Hartree was aware of Turing's characterization of the ACE as universal, but on the other that its significance was lost on him. Given Hartree's first hand experience of electronic computing and intellectual standing, this is strong evidence that the characterization of computers as universal machines was not at all obvious or straight-forward.

A different perspective was offered by Claude Shannon in an article discussing work carried out in 1948 on "the problem of constructing a computing routine or 'program' for a modern general purpose computer which will enable it to play chess" [Shannon, 1950, p. 256]. Shannon did not define what he means by "general purpose", however, and immediately introduced a contrast between such computers and machines which would carry out specific non-numerical tasks, stating that "[m]achines of this [latter] general type are an extension over the ordinary use of numerical computers in various ways". Later in the paper, when discussing the need to "represent chess as numbers and operations on numbers, and to reduce the strategy decided upon to a sequence of computer orders", Shannon concluded that "[i]deally, we would like to design a special computer for chess containing, in place of the arithmetic organ, a 'chess organ' specifically designed to perform the simple chess calculations" [Shannon, 1950, p. 265].

It is not easy to extract a single consistent view on universality from Shannon's paper. On the one hand, the computer is described as 'general-purpose' and the paper demonstrates the feasibility of programming such a machine to play chess. On the other hand, Shannon stated that "the rather Procrustean tactics of forcing chess into an arithmetic computer are dictated by economic considerations" [Shannon, 1950, p. 265] and made clear that his preference would be to develop special purpose machines. The paper makes no reference to Turing's work, and it seems clear that Shannon views the machines being designed in 1948 as numerical calculators, not as universal machines.

In September 1950, both Shannon and Turing attended a Symposium on Information Theory, organized by the Ministry of Supply in London. In a historical presentation, Colin Cherry made the following observation, suggesting that Shannon was not alone in his view that special-purpose machines would ideally be developed for various purposes:

> Just as arithmetic has led to the design of computing machines, so we may perhaps infer that symbolic logic may lead to the evolution of "reasoning-machines" and the mechanization of thought processes. [Cherry, 1950]

At the same symposium, Turing made a comment in discussion in which he distinguished special purpose machines for playing chess from the task of programming a computer to perform the same task [Turing, 1950b], but in 1950 his most significant contribution was in a paper discussing the relationship between machine thought and intelligence. In discussing the question "Can machines think?", Turing proposed to limit the discussion to electronic computers, and to motivate this included a section on "The Universality of Digital Computers". He concluded:

> This special property of digital computers, that they can mimic any discrete state machine, is described by saying that they are *universal* machines. The existence of machines with this property has the important consequence that, considerations of speed apart, it is unnecessary to design various new machines to do various computing processes. They can all be done with one digital computer, suitably programmed for each

> case. It will be seen that as a consequence of this all digital computers are in a sense equivalent. [Turing, 1950a]

This paper appears to have been widely read, and very quickly changed the way in which computers were described. In August 1951, Wilkes wrote an article for the *Spectator* on the question "Can Machines Think?" in which he referred to Turing's paper, classified "modern automatic-calculating machines" as universal, and wrote that:

> Provided that the basic operations form a logically complete set, a universal machine can be programmed to do anything which could be done by a specially built machine. The tendency nowadays is, therefore, to ask whether a universal machine could be programmed to perform a particular function, rather than to ask whether it would be possible to design a special machine for the purpose. The universal machines which have been built so far have been designed for performing arithmetical calculations rather than the logical operations which would be involved if they were to simulate human behaviour. This is not, however, a matter of fundamental importance. [Wilkes, 1951b]

Although Wilkes is here clearly influenced by Turing, he appears, like Shannon, to be envisaging different classes of machines specialized for different tasks, while simultaneously recognizing the universality of particular machines within each class. The required specialization is in the set of basic operations that the machine provides. As Wilkes put it later in 1951, "[a] machine primarily intended for experiments on 'thinking' would not differ in any fundamental way from an automatic calculating machine. The choice of basic order code would, perhaps, be somewhat different, since the emphasis would be on logical rather than arithmetical operations" [Wilkes, 1951a, p. 88]. Turing, however, is quite explicit that a single machine could be used for all purposes: as pointed out above, this point of view is implicit in his design for the ACE.

Over the next few years, Turing's view gained ground. In a 1952 article about chess programs, D. G. Prinz wrote of:

> 'electronic brains' or, to give them their proper name, universal high speed electronic digital computers. The emphasis here is on the 'universal' . . . The problem is no longer 'making a machine to play chess' but rather 'making a machine play chess' [Prinz, 1952, p. 261]

In the same year, Tony Oettinger spent a year with Wilkes in Cambridge working on programs which simulated learning. One of these simulated the ability of a machine to go shopping, but rather than suggesting that the EDSAC be supplemented with a 'shopping organ', Oettinger was happy to represent shops and products by integers and to write a purely numerical simulation. In documenting this work, he cited Turing's 1950 paper and wrote that universal machines "have the important property of being able, when provided with a suitable *programme*, to mimic arbitrary machines in a very general class" [Oettinger, 1952, p. 1243].

By 1953 Wilkes himself had adopted the more general view: "machines of this kind are sometimes known as *universal* machines. Given a suitable program a universal machine can do anything which could be done by a specially built machine" [Wilkes, 1953a, p. 1232]. Shannon, however, retained an interest in special purpose machines, developing a physical machine to solve simple mazes rather than writing an equivalent program. In a survey paper written in 1953, he stated that "[m]ost digital computers, provided they have access to an unlimited memory of some sort, are equivalent to universal Turing machines and can, in principle, imitate any other computing machine and compute any computable number" [Shannon, 1953, p. 1236], but significant parts of the paper are devoted to a consideration of 'machines' for various purposes, not programmes. More generally, this preference for special purpose machines has been noted as a feature of the cybernetics community [Pickering, 2002].

Turing's paper of 1950 was therefore a turning point in the characterization of the computer as a universal machine. Before its publication, this link was only made by Turing and his close associates, and other writers, even those intimately connected with computers and familiar with the relevant literature, did not make the connection, or think it important. Following 1950, however, Turing's paper was widely cited, and his characterization accepted and put into circulation.

3.8 General purpose machines

The third claim to be considered in this chapter is most clearly stated by Davis, who states that the fact that the computer is now thought of and used as a general-purpose machine rather than, say, a specialized calculator is attributable to Turing's characterization of it as a universal machine. However, automated computation was applied in a wide variety of areas, both before and after 1945.

As discussed above, modern digital computers emerged from the two fields of automatic computation and cybernetics. The majority of the early computers were built specifically for performing numerical calculations; the best known exception is perhaps the Whirlwind, developed at MIT as a flight simulator [Redmond and Smith, 1980]. Cybernetics suggested a wider range of applications: Wiener had originally been inspired by the problems posed by automated support for anti-aircraft guns, and the cybernetics-inspired analogy between the computer and the brain naturally suggested that a wide range of mental tasks could be performed by computer.

A third influence on the application of computers came from the data processing industry. Even before the first electronic computers were completed, punched card equipment was adapted or developed to provide a greater capability for automatic computation. Such machines continued in use well in to the 1950s, when electronic machines were still scarce and expensive resources. The possibility of carrying out commercial applications on computers was encouraged by these developments, and the company started by Eckert and Mauchly had this as its focus.

Turing himself had a very clear idea of the range of applications that computers could be used for, and in a lecture in 1947 gave as an example the possibility of computers being used to solve jigsaw puzzles [Turing, 1947]. As Davis comments, it is possible that Turing's outlook here was coloured by his computing experience during the war which, unlike von Neumann's, was not primarily concerned with numerical calculation. The details of this work remained classified, but it is striking that Turing's design for the ACE made many fewer assumptions about the intended use than the EDVAC design, and in described a computer which could have been more easily used for non-numerical applications [Turing, 1946].

3.9 Conclusions

This chapter has focused on a particular episode in the development of modern computers, namely the articulation of the so-called 'stored program principle' in 1945. This episode has been given great prominence by historians of computing, and the involvement of von Neumann makes it a plausible place to look for a logical influence on computer design. However, it should be stressed that this episode represents a moment of *closure* as much as a moment of invention, a point when the efforts of many people over the preceding decade to design machines capable of large-scale automatic calculation reached a widely accepted conclusion. The *Draft Report* was a concrete paradigm which, as the response to it at the Moore School course showed, enabled workers in the field to agree on the basis of the design of computers and focus in a concentrated and collaborative way on their implementation.

Outside the world of computer builders, however, the stored program principle attracted little immediate attention. In the scientific literature before 1950, the new machines and those under development were treated together, and characterized firstly by their ability to perform computation automatically, leading to the discussion about 'giant brains', and secondly by the high speed obtainable with electronic technology. The stored program property was seen as a technical feature required by the use of electronics, and slightly later as one that made programming easier in some respects, not as the defining property of a new technology as it later became.

The details of the history of the development of the computer lend little support to the claim of Mahoney and Davis that the computer was developed as a byproduct or application of theoretical work in mathematical logic. Instead, the majority of the early work was inspired by the desire to automate numerical calculation. The interaction between von Neumann and the ENIAC group raises the possibility that logical concerns played a part in the design of the *Draft Report*, and while this cannot be ruled out, it is striking that in the immediately following period arguments for the design were based on practical concerns of engineering rather than logic. It seems quite plausible that something like the stored program design would have emerged even without von Neumann's involvement with the ENIAC group.

Similarly, the claim that the general-purpose nature of the computer stems from Turing's universal machine concept seems to overstate the role of logic. Automatic computation using punched card machinery was widespread between the wars, and the emergence of the computer from a background in automatic calculation, cybernetics and data processing made it inevitable that a range of applications would be considered for the new machines.

In both areas, of the design and application of computers, the influence of logic seems to have been indirect, mediated by the ideas of cybernetics and in particular the idea that the electronic stored-program computer could be understood not merely as an electronic calculator, but as a device essentially analogous with the brain. Von Neumann wrote this analogy explicitly into the first description of the new computer, in the *Draft Report*. Although more constrained by security restrictions, Turing seems to have inspired many of his co-workers at Bletchley with a similar vision of the meaning of the computer and the scope of its potential application. The success of this strategy can be seen in popular representations of the new technology which was very widely described as being an "electronic brain".

Finally, it was argued that in the 1950s, stored-program computers became widely characterized as universal machines, a development that seems to be largely attributable to the writings of Turing himself. To this extent, then, Davis's comment that "computers are logic machines" can be supported, but with the important proviso that this does not describe a fact about the nature or origins of the computer, but rather the way in which scientific culture came to think of the new machines.

Again, we can note the importance of cybernetics: Turing’s 1950 paper was not specifically logical or technical, but rather a philosophical contribution to the discussion of the cybernetic question of whether machines could think.

Chapter 4

Machine-level programming and logic

The task of programming the new machines, or ‘coding’, was understood to be that of specifying the sequence of operations that a machine would carry out in the course of a computation. The available operations were defined by the machine’s ‘order code’, a list of the instructions out of which programs could be constructed. Many different order codes were possible, however, and it was only through practical experience that the features of a successful code could be identified, as the von Neumann and his collaborators realized:

> It is easy to see by formal-logical methods that there exist codes which are *in abstracto* adequate to control and cause the execution of any sequence of operations which are individually available in the machine and which are, in their entirety, conceivable by the problem planner. The really decisive considerations from the present point of view, in selecting a code, are of a more practical nature: simplicity of the equipment demanded by the code, and the clarity of its application to the actually important problems together with the speed of its handling of those problems. [Burks et al., 1946, p. 100]

By 1950, broad agreement had been reached about the basic features that a successful and usable code should provide. A book published by Wilkes and his colleagues in Cambridge described the programming system devised for the EDSAC, but its authors pointed out that “for the main part [the methods] may readily be translated into other order codes” [Wilkes et al., 1951, preface]. This book was widely read, and contributed to the further spread of this model of programming.

The first half of this chapter describes the evolution of this model; the historical development of the key features of the early machine codes is described, emphasizing the experimentation and consideration of alternatives that preceded the acceptance of a ‘standard model’. The second half of the chapter considers the more logical and philosophical aspects of this programming style.

4.1 Sequencing

In his proposal for an automatic calculating machine, written in 1937, Howard Aiken observed that the design of existing calculating machinery made it easy to carry out a small number of operations repeatedly on the elements of large data sets, typically held as decks of punched cards. In many scientific applications, however, Aiken believed that the opposite procedure was required, namely the ability to carry out an extended sequence of operations on individual numbers [Aiken, 1937]. This requirement strongly influenced the design of the first large-scale, automatic digital calculators. Aiken and Grace Hopper wrote of the completed Mark I that:

> The development of numerical analysis . . . [has] reduced, in effect, the processes of mathematical analysis to selected sequences of the five fundamental operations of arithmetic: addition, subtraction, multiplication, division, and reference to tables of previously computed results. The automatic sequence controlled calculator was designed to carry out any selected sequence of these operations under completely automatic control. [Aiken and Hopper, 1946, p. 386]

and Arthur Burks described the ENIAC in similar terms:

> the ENIAC can solve any problem which can be reduced to numerical computation, i.e. to a finite sequence (of reasonable length) consisting of additions, subtractions, multiplications, divisions, square-rootings, and the looking up of function values. [Burks, 1947, p. 756]

Specifying the required sequence of operations was therefore a basic aspect of coding problems for these machines. Prior to 1945, however, the majority of calculating machines and installations had units which were capable of operating in parallel, and hence they could carry out more than one operation simultaneously. This introduced a conflict between the need to describe a computation as a sequence of operations, and the desire to make the most efficient use possible of the available machinery.

For example, Mark I contained a number of storage registers, or counters, each of which stored a number and allowed other numbers to be added to it. A program was viewed as a simple sequence of instructions in a standard form, each specifying that a number be copied from one register to another, along with some operation that might be performed on the number, such as taking its complement to enable subtraction rather than addition to be performed. This sequence of instructions was read from a paper tape by a sequence mechanism which was incapable of skipping instructions or going backwards in the sequence.

As well as the storage registers Mark I possessed a number of specialized units for carrying out other operations, such as a unit which performed multiplication and division. These specialized units were controlled by multiple instructions: for example, performing a multiplication required two instructions to load the multiplier and multiplicand into the multiplying unit, followed by a third instruction to retrieve the result. As Aiken commented, "no longer does each line of coding correspond to a single operation of the machine" [Aiken and Hopper, 1946, p. 449].

Once started, multiplication was carried out by the dedicated unit quite independently of the main sequence mechanism. In general, this would take much longer than a simple operation to copy a number from one register to another, and until the multiplication was complete the main sequence unit would be idle. This was seen as a waste of computing resource, and the technique was adopted of 'interposing' unrelated instructions between the instructions specifying a multiplication, thus allowing the main body of the machine to perform useful work while waiting for the multiplication unit to finish.

Thus, despite Aiken's emphasis on sequence control, a program for Mark I could not be read as a straightforward sequence of the operations carried out by the machine, and the parallelism in its architecture was reflected to some extent in the way it was coded. Although increasing the efficiency of machine usage, however, the technique of interposing instructions created problems in writing and maintaining programs, as Richard Bloch, an early Mark I programmer, noted:

> Although I tried to annotate my coding sheets thoroughly, it was at times almost impossible for an operator running a program to decipher exactly what was going on. Aside from the fact that the logical flow of the program was at times terribly difficult to follow, the compaction of code made the task of analysing and tracking down the cause of a sudden machine stoppage doubly difficult. [Bloch, 1999, p. 87]

Hardware parallelism was also a feature of the ENIAC. The machine was built around 20 accumulators which, like Mark I's storage registers, both stored a number and carried out simple operations on it. It also possessed several separate units for carrying out specialized tasks, including multiplication. The ENIAC was not programmed by means of instructions read from a tape, however, but was physically reconfigured for each different problem. Individual instructions could be placed on accumulators, and transfer of information or carrying out a multiplication were enabled by connecting units together in the appropriate way. The sequencing of operations when the machine was running was controlled by special 'program pulses' that circulated round the machine. Depending on the configuration, any number of distinct operations could be carried out in parallel, and the setup for a particular problem could be described in a two-dimensional diagram [Goldstine and Goldstine, 1946].

Like Bloch, however, the ENIAC team felt that the advantages of parallelism were outweighed by the complications it introduced into the programming, as Eckert explained in a lecture in 1946:

> In thinking out the various operations of the machine, if they can be thought out in a purely serial fashion, it is not necessary to worry about any irrelevant timing between the various steps. For example, if two steps A and B are being done together, A and B start at the same time but do not necessarily end at the same time since a different length of time may be required to do each step. ... The human brain does not think in several parallel channels at the same time: it usually thinks these things out step by step. Therefore, in all ways, it is found exceedingly desirable to build the machine so that only single steps are performed at any time. The ENIAC is usually used in this way. [Eckert, 1946, p. 114]

As Eckert went on to note, the relay machine developed by the Bell Telephone Laboratories was programmed in a purely sequential manner [Alt, 1948]. Sequential, step-by-step processing emerged in the *Draft Report* as a fundamental design principle, there motivated by a desire to minimize the amount of physical equipment used:

> The device should be as simple as possible, that is, contain as few elements as possible. This can be achieved by never performing two operations simultaneously, if this would cause a significant increase in the number of elements required. The result will be that the device will work more reliably ... It is also worth emphasizing that up to now all thinking about high speed digital computing devices has tended in the opposite direction: Towards acceleration by telescoping processes at the price of multiplying the number of elements required. [von Neumann, 1945, §5.6-7]

This principle was applied at all levels of the design. Numbers were no longer stored in separate units with some processing capability, but in a passive memory. A single arithmetic unit performed all calculations, so the possibility of parallel execution of operations was removed. Further, the individual digits of the operands to an operation were handled sequentially, one at a time. As a consequence of this, in the proposed code for the EDVAC there is a strict correspondence between instructions and operations carried out.

Mitchell Marcus and Atsushi Akera have suggested that the emphasis on sequential processing was motivated by the desire to increase reliability by using as little physical equipment as possible [Marcus and Akera, 1996, p. 23]. As shown by the quotation above, some support for this view can be found in the *Draft Report*. However, many computer designs after the EDVAC reintroduced parallel processing in some areas: for example, for the machine built at the Institute of

Advanced Studies, von Neumann and his collaborators proposed handling the digits of a number in parallel [Burks et al., 1946]. This suggests that reliability was not the only issue. In 1947, without mentioning reliability, Mauchly articulated a view which balanced the desire for efficiency with the need to simplify programming, making it clear that parallel processing was acceptable as long as program structure remained strictly sequential:

> the machine should be kept serial as far as the operator is concerned. That is, no two instructions which the operator gives the machine are to be carried out at the same time. Any particular instruction which the operator gives, however, may involve the simultaneous operation of numerous parts. [Mauchly, 1947, p. 204–5]

The model that finally emerged, then, viewed a program as a sequence of instructions each specifying a single operation which was carried to completion before the next instruction was obeyed. This approach was natural in machines whose design followed the *Draft Report* in having a passive store and a single arithmetic unit: this virtually ruled out the possibility of two operations being carried out simultaneously. Although reliability was initially a consideration, this model was subsequently justified by reference to its role in simplifying the task of programming.

Two main approaches were adopted to the problem of specifying the sequence of instructions in a program. On machines which read instructions from an external tape, such as Mark I, the sequence of instructions was simply defined by their order on the tape. The *Draft Report* copied this approach, storing instructions in contiguous locations in memory: when an operation was complete, the next instruction was automatically read from the following memory location. This approach introduced an inefficiency on the delay-line storage that was commonly used at the time, however, as there was no guarantee that the next instruction would be immediately available when it was required. To avoid these delays, an alternative approach to coding included in every instruction the address of the next instruction to be executed. As this could be anywhere in memory, by careful planning it was possible to avoid delays by ensuring that the required instruction was available just as it was needed by the program, an approach known as ‘optimum coding’. A few machines adopted this approach [Bloch et al., 1948], but as random access memory technology became available the practical advantages of optimum coding became less crucial and sequential placement of program instructions in memory became the norm.

Ceruzzi has discussed the transition from “an architecture that processed data in parallel to one that processed data serially” [Ceruzzi, 1997]. This section has examined this transition from the point of view of programming, and shown that it took some time for the notion of instruction sequencing to reach a stable form. The concept that did emerge was influenced by experience in programming the new machines as much as by considerations of machine architecture.

4.2 Transfers of control

It quickly became apparent that programs for automatic calculators could not be a simple list of the desired sequence of operations. In 1947, Mauchly described the problem and its solution as follows:

> Calculations can be performed at high speed only if instructions are supplied at high speed. Thus many instructions must be made quickly accessible. The total number of operations for which instructions must be provided will usually be exceedingly large . . . However, such an instruction sequence is never a random sequence, and can usually be synthesized from subsequences which frequently recur. [Mauchly, 1947, p. 204]

All the automatic calculators shared this model, according to which a computation was built up from a number of distinct subsequences of instructions. Normally, one sequence was thought of as defining the structure of the entire computation: there needed therefore to be some way to execute the other sequences when necessary and to cause a given sequence to be repeated as often as required. These requirements were met in different ways by different machines.

For example, Mark I's sequence control unit read instructions from paper tape. Computations were normally split across multiple tapes, each containing a particular instruction sequence, but there was no mechanism for automatically transferring from one tape to another. Instead, the programmer had to leave detailed instructions for the operators specifying what tapes should be loaded on to the machine and when, among other pieces of information. Aiken and Hopper described a simple program for evaluating a polynomial, which consisted of a "starting tape", which would read initial data from cards, and a "main control tape", which would compute the value of the polynomial for particular data values. The operator was instructed to restart the calculator after the starting tape had completed, and then to run the main control tape "until the card for $F(9.99)$ has been punched, then press stop key" [Aiken and Hopper, 1946, p. 528]. The repetition of the instructions on the control tape in this process was achieved by making the tape "endless" [Aiken, 1946, p. 156]: in practice this was done by simply gluing the ends of the tape together, so forming a loop.

On the ENIAC, the high-level structure of a program was expressed physically in the machine's hardware in a unit known as the "master programmer". This consisted of devices known as "steppers", which allowed a sequence of up to six distinct subsequences to be defined, each of which could be repeated a specified number of times. By using more than one stepper, programs could be constructed in which the subsequences themselves had a similar internal structure. The master programmer contained a total of 10 steppers, thus allowing for the definition of highly complex program structures [Goldstine and Goldstine, 1946]. By 1946, Aiken recognized the need to supply Mark I with multiple sequence mechanisms, and in 1947 a “subsequence mechanism” was added, which allowed the machine to be configured with more than one instruction tape, and provided the ability to switch between them automatically [Bloch, 1947].

In the early machines, then, the logical structure of a program was expressed physically in some aspect of the machines’ setup. In the subsequence mechanisms employed by Mark I and the Bell Labs relay machine, for example, transfer of control was effected by an instruction which made explicit reference to the tape reader containing the next subsequence to be executed. With the adoption of the stored program design, in which a complete program was stored in a single, uniform memory, a different approach to the question of the transfer of control became necessary. The code defined in the *Draft Report* made use of the fact that instructions could be referred to by the address of the storage location holding them. A generalized transfer instruction was provided which had the effect of transferring control to the instruction at a specified address. Eckert explained the distinction as follows:

> The only big difference between this control on a relay machine and the control in the EDVAC is that the control words in the EDVAC are read from its internal memory, and that some of the operations may send the control from one point in the memory to another. In other words, the main routine tape in a relay machine may indicate that the operations on a certain sub-routine tape are to be done, while in the EDVAC there may be a symbol in the memory which instructs the control to go to another place in the memory and do what is indicated there. [Eckert, 1946, p. 116]

The EDVAC included in its code an order ζ which had the effect of connecting the control organ to a specified memory location from which program execution would continue [von Neumann, 1945, §15]. This location could either be a previously executed instruction, or the beginning of a distinct sequence, so this single order supported both the execution of a new subsequences and the repetition of the current sequence. (Machines in which each order specified the address of its successor in effect made unconstrained transfer of control the default mechanism, and their codes did not need a specific transfer instruction.)

A significant aspect of transfer orders is that they allow the flow of control within a program to be specified without making reference to particular features of the machine on which the program is running. The stored program design therefore made possible a more abstract understanding of program structure, where an entire program, including all necessary subsequences of operations, is thought of as a single sequence of labelled instructions. This in turn made it possible to think of a program separately from the machine it will run on. The generalized transfer instruction therefore marks a significant step towards a complete logic of control, a notation independent of any particular hardware configuration and capable of expressing both the elementary operations and also the order in which they will be performed.

Two other points can be noted about the transfer instruction. Firstly, although it appeared in conjunction with the stored program design, it does not require it: there is no reason in principle why relay machines should not have labelled each instruction on a tape, in the same way that certain forms of data, such as function values, were already labelled. The stored program design made it easier to implement, however, as all instructions were stored in a memory location and the address of the current instruction was stored in the control unit.

Secondly, although more flexible than, and capable of reproducing the effect of, any particular machine design, the use of transfer instructions could make it harder to perceive the structure of a program. The use of subsequences and repetition of instruction sequences were not made explicit in the programming notation: by replacing both with a more general, lower level instruction, programs became harder to read and understand.

4.3 Condition testing

The simple transfer instruction was soon found to be insufficiently expressive to define all common computational patterns. In many cases, the future course of a computation will depend on the results obtained so far: a commonly cited example was where it was necessary to carry out a sequence of instructions until the results fell within a certain tolerance, the precise number of iterations needed to achieve this not being known in advance.

A variety of approaches were adopted to provide this capability. In Mark I, for example, counter 72 was known as the ‘automatic check counter’, and an instruction code was provided which would halt the computation if the last result calculated in that counter was less than zero. This was typically used to test whether a computed quantity fell within desired limits. After halting, the overall computation could be restarted manually by the operator, and the conditions for doing this were stated in the operating instructions provided with the program [Aiken and Hopper, 1946].

As originally designed, the ENIAC did not include any mechanism for testing the values currently held in the accumulators. This capability was provided, apparently at a late stage in the development, by adding a ‘direct input line’ to the master programmer. A signal on this line caused computation to continue with the next defined subsequence, regardless of whether the current sequence had been repeated the specified number of times. By connecting the numerical output from an accumulator to the direct input line, it became possible to use numerical data to trigger the master programmer and thus affect the course of the computation [Marcus and Akera, 1996].

By 1945, then, experience had demonstrated the utility of instructions which would enable the course of a computation to depend on the result of a test, typically of the sign or magnitude of a number, applied to some data value. Von Neumann summarized the situation as follows:

> A further necessary operation is connected with the need to be able to sense the sign of a number, or the order relation between two numbers, and to choose accordingly between two (suitably given) alternative courses of action [von Neumann, 1945, §11.3]

The code defined in the *Draft Report* provided this capability indirectly, however, relying on the fact that in the stored program design instructions and their addresses can be treated as numeric data, and so, in principle at least, examined and modified just as numbers can. A basic operation, s, of the arithmetic unit was defined which would take four numbers, x, y, u and v as input; if $x \geq y$ the operation would have as result u, and if $x < y$ the result would be v. Von Neumann argued that “the ability to choose the first or the second one of two numbers u, v depending on such a relation, is quite adequate to mediate the choice between any two alternative courses of action” [von Neumann, 1945, §11.3]. This was achieved by supplying the addresses of two instructions as u and v: once the desired address had been selected by the s instruction, it could be copied into the address field of a ζ transfer instruction. When the modified ζ instruction was executed, control would be transferred to whichever address had been selected, thus allowing the behaviour of the program to vary according to the outcome of a purely numeric test.

In the ACE report, Turing adopted a similar indirect approach to conditional tests. Rather than providing a specific operation to choose between two numbers, however, he suggested that the destination address could be calculated using existing numerical instructions before being copied into an unconditional transfer instruction [Turing, 1946].

By the middle of 1946, however, codes had been proposed which did not require programmers to construct alternative transfer instructions explicitly, but instead included a single instruction to carry out a conditional transfer. Eckert and Mauchly's 'Code A' contained instructions which would jump to a specified instruction depending on the result of a comparison between two other numbers [Eckert, 1946], and the code used by von Neumann's group at Princeton had similar instructions which effected a transfer depending on the sign of the number stored in the accumulator [Burks et al., 1946]. Conditional transfer instructions of these or similar types were found in all subsequent codes.

It is a striking fact that the utility of coding a conditional pattern of control such as "transfer to instruction 53 if the value of the number at address 256 is negative" as a single instruction appears not to have been obvious, the Bell Labs machine being the only one of the early automatic calculators to provide such an instruction [Alt, 1948, p. 72–3]. Carpenter and Doran have commented that "[i]t is strange that conditional branching was a stumbling block to both von Neumann and Turing, especially since the program for an abstract Turing machine is just one large decision table" [Carpenter and Doran, 1977, p. 271]. From a Whiggish perspective, this presents a problem requiring an explanation which does not seem to be immediately forthcoming. However, it is better seen as a piece of evidence of the potential difficulty of making innovations that later come to seem self-evident and of the extended process of exploration and negotiation that often accompanies conceptual innovation.

4.4 Instruction modification

The stored program design raises the possibility of manipulating instructions programmatically, a capability exploited by both von Neumann and Turing to provide conditional jumps, as discussed above. As with other programming concepts, however, the idea of modifying the instructions making up a program as it progressed underwent considerable evolution before reaching a definitive form.

In the *Draft Report*, a rather complex chain of design decisions led to the proposed machine possessing only a partial ability to treat program orders as data. Von Neumann first considered the desired memory capacity of the machine, and concluded that 32 "memory units", or binary digits, would be sufficient to store a real number to an appropriate degree of precision. He then wrote that "[t]he fact that a number requires 32 memory units, makes it advisable to subdivide the entire memory in this way: First, obviously into *units*, second into groups of 32 units, to be called *minor cycles* ... It will therefore be necessary to formulate the standard orders in such a manner than each one should also occupy precisely one minor cycle, i.e. 32 units" [von Neumann, 1945, §12.2].

No theoretical principle was invoked to justify treating orders as data. Rather, a pragmatic decision was taken to constrain orders to be the same size as numbers, in order to make the engineering of the memory as simple as possible. Underlining the distinction between the two forms of data, von Neumann went on to write that "[m]inor cycles fall into two classes: *Standard numbers* and *orders*. These two categories should be distinguished from each other by their respective first units i.e. by the value of i_0. We agree accordingly that $i_0 = 0$ is to designate a standard number, and $i_0 = 1$ an order" [von Neumann, 1945, §15.1]. Far from being treated in the same way, orders and numbers were clearly demarcated and treated separately.

Nevertheless, there were two cases where this demarcation broke down. Firstly, when considering the orders needed to transfer numbers from memory into the arithmetic unit, von Neumann decided that "[i]t is simplest to consider a minor cycle containing a standard number . . . as such an order per se" [von Neumann, 1945, §15.3]. In other words, in certain contexts a number would be interpreted as if it expressed an implicit order. Secondly, when a number was transferred from the arithmetic unit back to memory, the way in which this transfer was effected was to depend on whether the minor cycle it was being transferred to held a number or an order. In the first case, the entire minor cycle would be overwritten with the new data, but in the second case only those parts of the order which held the address of the minor cycle being operated on would be modified [von Neumann, 1945, §15.6]. This facility for 'address modification' was the only way provided by the code to modify the orders making up a program, and was used among other things to provide conditional jumps, as discussed above.

Turing defined the ACE's memory in essentially the same way as von Neumann, as "minor cycles" of 32 binary digits grouped into "major cycles", and wrote that "[s]uch a storage will be appropriate for carrying a single real number as a binary decimal or for carrying a single instruction" [Turing, 1946, p. 24-5]. When discussing the way in which numbers would be encoded in the store he further stated that a minor cycle might contain some information which would "distinguish between minor cycles which contain numbers and those which contain orders or other information" [Turing, 1946, p. 25].

However, the ACE report assumes that programs will have an unrestricted ability to modify their own orders, using the same operations as are used on numbers. For example, when describing how to perform a conditional jump, Turing did not rely on a specific facility for address modification, but instead suggested performing arithmetical calculations directly on the minor cycles containing the instructions. He gave the following example where it is required to carry out instruction 33 or 50 depending on whether a certain digit D is 0 or 1:

> One form the calculation can take is to pretend that the instructions were really numbers and calculate
>
> $$D \times \text{Instruction } 50 + (1 - D) \times \text{Instruction } 33.$$
>
> The result may then be stored away, let us say in a box which is permanently labelled 'Instruction 1'. We are then given an order . . . saying that instruction 1 is to be followed, and the result is that we carry out instruction 33 or 50 according to the value of D. [Turing, 1946, p. 35]

In the ACE proposal, then, Turing made use of an unrestricted ability to manipulate instructions as numbers, and for a program's instructions to be constructed and modified by the program itself as it runs. This approach was also taken by several speakers in the Moore School lectures in mid-1946. Mauchly mentioned the requirement to store instructions and numerical data in the same device and the pragmatic reasons for doing so, but then stated that:

> A much more fundamental reason for this requirement is that the instructions themselves can then be operated on by the use of other instructions. It should be possible to carry out such operations upon instructions by the use of the same instructions as would be utilized when operating upon numbers [Mauchly, 1946, p. 455].

Calvin Mooers made this point even more bluntly, stating that the modification of orders should be "a simple arithmetic operation between numbers and orders" [Mooers, 1946, p. 470]. The actual codes described by Mauchly and Mooers did not differentiate numbers and data in the way that von Neumann's EDVAC code did, but despite the generality of the statements above, made only rather limited use of operation modification in copying bits from one word to another, to set up subroutine parameters, and incrementing address fields in operations. Whereas Eckert and Mauchly's Code A included a specific operation for doing this [Eckert, 1946, p. 122], Mooers used straightforward numerical addition, thus simplifying his code slightly.

Von Neumann and his collaborators gradually came to adopt a more relaxed approach than that of the *Draft Report*. In the design of the Institute of Advanced Studies computer they distinguished "two different forms of memory: storage of numbers and storage of orders" [Burks et al., 1946, p. 98] before observing that orders, suitably coded, could be stored in the same memory as numbers. Orders and numbers were no longer formally distinguished, but specific orders were defined to rewrite the address field in an order. Functionally, the code defined was very similar to that in the

Draft Report. In 1947, however, von Neumann and Goldstine stated that control could “modify any part of the coded sequence as it goes along” [Goldstine and von Neumann, 1947, p. 153].

Despite these statements of principle, however, the importance of instruction modification and the range of its application was often limited to the modification of addresses in individual instructions [Bloch et al., 1948, p. 293], [Bowden, 1953, p. 29]. A good example of the power of unrestricted modification is the “Initial Orders” written by David Wheeler to load programs into the EDSAC when it was started up [Wheeler, 1950]. This program included such techniques as the use of “ambiguous words”, which at different times were treated as numbers or instructions, and repeatedly formatted a “transfer order” which would carry out quite different tasks on different occasions of use.

In summary, then, the first order code for a stored program machine, that of von Neumann’s *Draft Report*, made an explicit distinction between numbers and orders, and only permitted a limited form of modification of orders for specific purposes. Gradually, codes evolved which permitted unrestricted manipulation of instructions as numerical data, but except in a few cases, this facility was usually made use of only to modify the address contained in an order.

4.5 Subroutines

It was universally recognized that certain computational routines were of general utility, and that the programming task would be simplified if such routines could be reused rather than being repeatedly coded. From the Mark I onwards, programs were typically viewed as containing a ‘master routine’ which invoked a range of subroutines which were not necessarily specific to the problem being solved, and computing installations aimed at having a ‘library’ of subroutines which could easily be applied to new problems.

As discussed in Section 4.2, subroutines, as reusable sequences of instructions, were physically distinct program tapes on Mark I and the Bell Labs machine. One consequence of this was that every time a subroutine was called, exactly the same instructions were executed. On stored program machines, however, subroutine instructions were stored in the same memory as the master routine, and the ability on such machines to modify program instructions led to a much more flexible use of subroutines.

Subroutines were not mentioned in the *Draft Report*. By contrast, they were central to the approach to programming described by Turing in the ACE report:

> We also wish to be able to arrange for the splitting up of operations into subsidiary operations. This should be done in such a way that once we have written down how an operation is to be done we can use it as a subsidiary to any other operation. [Turing, 1946, p. 34]

This approach requires the ability to transfer control to the beginning of a subroutine and to return to the calling routine on completion of the subroutine. The former task can be accomplished by a straightforward transfer instruction, but the latter is more complex because control will have to return to different places at different times. Turing's solution was as follows:

> When we wish to start on a subsidiary operation we need only make a note of where we left off the major operation and then apply the first instruction of the subsidiary. When the subsidiary is over we look up the note and continue with the major operation. [Turing, 1946, p. 35]

The notes of the return addresses were to be "buried" in storage, and a record kept of the most recent one. On completion of a subsidiary routine, the most recent note would be "disinterred" and control returned to that point. It is characteristic of Turing's approach to programming that both these operations were themselves to be performed by subsidiary routines, known as BURY and UNBURY.

A second problem with the use of subroutines in stored program machines was that in general a subroutine would be located at different places in the memory on different occasions of use. However, subroutines typically make reference to addresses internal to the subroutine: the commonest occasion for this is when control transfers from one location to another inside the subroutine, something that would be necessary in all but the simplest cases. The problem then is how to reconcile the need to provide a fixed address in the transfer instruction with the fact that that address will vary in different programs, depending on where the subroutine is located in memory.

Turing's solution to this problem was to propose a two stage process of program assembly. Instructions were to be written on cards in a "popular", or relatively human-readable, form and identified by "group name" and "detail figure", or line number within the group. Transfer instructions would refer to their destination by group name and detail figure. When a program was being constructed, all the cards required would be collated, and sorted by group name and detail figure. The instructions would then be renumbered sequentially, and the popular group name and detail figure references would be replaced by the actual binary addresses used in the program. Turing recognized that "[i]t would be theoretically possible to do this rearrangement of orders within the machine" [Turing, 1946, p. 38], but did not propose to do this in the first instance.

Goldstine and von Neumann considered the use of subroutines in detail in a report circulated in 1948 [Goldstine and von Neumann, 1948]. They described the changes that would have to be made to a subroutine when it was being used as a constituent of a new problem, and classified them into those that would be made before the subroutine was used in a particular problem, and those that would have to be made while the program was running.

The first type of change was that already identified by Turing, namely that a subroutine would typically appear at different locations in memory on different occasions of use, and that references to addresses internal to the subroutine would need to be modified before the subroutine could be successfully used. Unlike Turing, Goldstine and von Neumann considered how this could be done automatically. They proposed a procedure for subroutine reuse which involved loading the various instruction sequences into the machine, and running a special “preparatory routine” which would make the required changes to the code before the complete program was executed.

The second type of change was due to the fact that a subroutine would in general be called more than once during the execution of a program. As well as the problem of returning control to the correct place on completion of the subroutine, Goldstine and von Neumann noted that subroutines need to be supplied with parameters, or data which can vary from one call to the next. Unlike the first type of change, which was handled by the preparatory routine, the changes required by parameters and return locations can only be dealt with when a program is running. Goldstine and von Neumann do not describe in detail how this could be done, but it is clear that they assume that some form of instruction modification while the program is running will suffice.

It is worth noting that this approach is in general less flexible than Turing’s proposal to store return addresses separately, which permits recursive calls to subroutines. This difference is perhaps accounted for by a different philosophy of program design. Whereas Turing, as noted above, viewed the use of subroutines as ubiquitous, Goldstine and von Neumann considered subroutines which performed significant amounts of computation, and seemed to have in mind a hierarchical structure in which the main routine would call subroutines, but references between subroutines would be rare.

In 1949, once the EDSAC was operational, a detailed scheme for handling all these aspects of subroutines was worked out by David Wheeler. Rather than reading the complete program into memory and then modifying it, as proposed by Goldstine and von Neumann, Wheeler wrote a set of “initial orders” which were loaded into the EDSAC when it was started, and which read a program from paper tape and placed it in memory before executing it [Wheeler, 1950]. Rather than modifying a complete program, in the style of Goldstine and von Neumann's preparatory routine, however, the initial orders interpreted a coded version of the program read from the tape and constructed the complete program in memory. Wheeler also invented coding techniques for modifying the return addresses in subroutines and allowing parameterized data to be used in subroutines. These were later described in the textbook issued by the Cambridge group, and became highly influential [Wilkes et al., 1951].

The adaptation of the familiar idea of a subroutine for use on the new stored program computers, then, can be characterized by two main features. Firstly, it turned out that subroutines could not be reused without a stage of processing prior to execution, where the required form of a complete program including subroutines was constructed in some way. Secondly, in the complete program thus constructed, the existing capabilities provided by transfer instructions and instruction modification were sufficient to make use of subroutines. In other words, in machine code, subroutines were not marked syntactically in any way, and apart from certain conventional patterns of usage, were not distinguished in any way from other code.

4.6 Machine code and program structures

Between 1945 and 1950, then, a widely accepted 'standard model' of order codes for stored program computers emerged. This standard model had three main aspects. Firstly, each code defined a number of basic instructions. The commonest of these controlled the transfer of data from one location in the computer to another and the various arithmetic operations that could be carried out. From the programmer's point of view, the important properties of basic instructions were that only one could be executed at any time, and that they were atomic, in the sense that the execution of a basic instruction could not be interrupted by any other instruction in the program.

Secondly, control instructions defined the order in which the basic instructions were carried out. Some codes assumed that instructions would be executed in the sequence that they were found in memory, and provided an unconditional transfer instruction to allow variations from this sequence. An alternative approach was for each instruction to specify explicitly the location of its successor. In addition, conditional transfer instructions were provided to allow the sequence or orders executed to depend on the current state of the computation.

Finally, instructions could be modified programmatically in the course of a computation. There were a number of standard situations in which this was known to be necessary, but rather than provide special instructions for these situations, most codes simply allowed instructions to be treated as numeric data, and placed no restrictions on the manipulations that could be performed on them.

Various extensions to this standard model had been proposed. For example, at the Moore School course in 1946, Mauchly presented a code which included “index counting instructions” to make the control of loops easier [Campbell-Kelly and Williams, 1985, p. 452], and Mooers described a modification to the von Neumann design using a device called a “sentinel” and a code which included “stop order tags” to facilitate the detection of boundary conditions in certain applications [Mooers, 1946]. Neither of these innovations were widely, if at all, adopted; for example, the very influential EDSAC code was essentially that of the standard model outlined above [Wilkes et al., 1951].

A striking feature of the standard model was that the facilities it provided for controlling the flow of a program did not coincide with the ways in which people thought about computational structure. For example, in the ACE report Turing stated that instruction modification and branching were together sufficient to carry out all required computations [Turing, 1946, p. 35]. In a lecture given to the London Mathematical Society in 1947, however, he described a number of “tactical situations that are met with in programming” [Turing, 1947, p. 117]. These described the way that a programmer thought about the overall structure of the computation that is being coded, and their use predated the stored program computer and even automatic computation.

A fundamental computational structure was the subprogram or subroutine, a set of instructions that could be written once and then executed whenever required by the demands of the computation. As described above, all automatic machines incorporated some method for structuring a computation out of a number of subroutines. The standard model of machine code contained no explicit representation of subroutines, however: instead, the required behaviour had to be implemented using the more primitive notions of transfer of control and instruction modification.

Another key computational structure is the ability to repeat instructions as often as required. Turing describes this situation as being “like an aeroplane circling over an aerodrome, and asking permission to land after each circle” [Turing, 1947, p. 118]. This situation can easily be coded using a conditional transfer, but this same instruction can be used in quite different situations, such as choosing between alternative courses of action, where no loop is involved. As this illustrates, there was no simple correspondence between the high-level computational structures in terms of which computations were planned, and the low-level instructions provided by standard machine codes. Programming textbooks explained how to implement the high-level structures using machine code [Wilkes et al., 1951], but this meant that it was not easy to grasp the structure and design of a program simply by inspecting the code.

4.7 Machine code and logic

Turing and von Neumann both commented on the relationship between the new activity of coding for automatic computers and the existing discipline of formal logic. Speaking to the London Mathematical Society in 1947, Turing stated that:

> I expect that digital computing machines will eventually stimulate a considerable interest in symbolic logic and mathematical philosophy. The language in which one communicates with these machines, i.e. the language of instruction tables, forms a sort of symbolic logic. [Turing, 1947, p. 122]

and Goldstine and von Neumann made a similar point:

> Since coding is not a static process of translation, but rather the technique 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. [Goldstine and von Neumann, 1947, p. 154]

However, Turing, von Neumann and Goldstine did not spell out exactly what the force of this comparison was. As discussed in Chapter 2, one of the achievements of logic had been to demonstrate how important aspects of mathematical language could be captured by formal, or 'mechanical' rules. A possible link between order codes and logic, then, derives from the fact that the former were defined in such a way as to be readable by machines, and so by definition 'mechanical'. Machine code programs and the instructions they contain bear little resemblance to the sentences of propositional and predicate logic, however, so it is worth exploring in a bit more detail what was understood by the analogy.

The terms 'logic' and 'logical' were used in discussions of computers in a number of senses, without necessarily implying a connection with mathematical logic. For example, it was common to distinguish the 'logical' from the 'physical' design of a machine, the distinction being that the logical design made no reference to specific circuits or electronic devices [Bloch et al., 1948]. From this, however, it is only a short step to a consideration of the notation in which the logical description of a machine can be expressed, a transition exemplified in the following quotation:

> Babbage invented a new algebra with which to describe the movements of the interconnected parts of the machine—to evaluate their *logic* to use the modern phrase [Bowden, 1953, p. 17].

This emphasis on the activity of the machine echoes the earlier emphasis that von Neumann had placed on the sequencing of operations. The *Draft Report* defined "[t]he logical control of the device" to be "the proper sequencing of its operations" [von Neumann, 1945, p. 2], and Goldstine and von Neumann later wrote of an example program that "[t]his extension will bring in a simple induction, and thus the first complication of a logical nature" [Goldstine and von Neumann, 1947, p. 113]. An explicit distinction was made between the 'arithmetical' or 'mathematical' operations of a computer and its 'logical' operations: Goldstine and von Neumann describe "arithmetical operations and transfers of numbers" as being the "properly mathematical (as distinguished from the logical) operations of the machine" [Goldstine and von Neumann, 1947, p. 115], and Edmund Berkeley included among the logical operations those of detecting a relation of inequality between two numbers, and providing for conditional branching and the automatic detection of the end of a calculation [Berkeley, 1950].

The analogy between order codes and logic, then, appears to have been based on an understanding of machine code as a formal language for defining the sequence of basic operations to be carried out by a machine. The view of logic as the study of formal languages was well established, having been put forward in works such as Carnap's *Logical Syntax of Language* [Carnap, 1937], but nevertheless formal languages of machine processes are different in many ways from the traditional logical calculi of deduction, and the question arises of why it seemed natural at this time to widen the denotation of the term 'logic'.

On possible explanation is that something like the following analogy was being appealed to. Just as deductive calculi provided rules of inference describing a formal relationship, that of entailment, between sentences, so the 'logical' aspects of machine code described a particular formal relationship, the order of execution, holding between the basic instructions of a program. Support for this interpretation is provided by the terminology used by Konrad Zuse. His programming notation, the Plankalkül, was named by analogy with the predicate calculus, or *Prädikatenkalkül* in German. Zuse is quoted as stating that his aim was "to provide a purely formal description for any computational procedure" [Giloi, 1997, p. 18], implying that the influence of logic was not to be found in the details of any particular calculation, but rather in the properties that are common to all, namely the ways in which computations can be organized.

An alternative interpretation of the naming of the Plankalkül has been offered by Bauer, who states that the "*Plankalkül* is an instrument for reasoning about programs – quite a modern point of view" [Bauer, 2000, p. 278]. This comment does not seem to be valid if interpreted as meaning that Zuse was concerned with proving or validating properties of his programs: unlike Goldstine and von Neumann or Turing [Turing, 1949], Zuse never tried to formalize properties of the data being used in a computation, for example. Zuse was very interested in logic, both at the level of computer design and also as an application—for example, one of his example programs was to check the well-formedness of a formula in propositional logic—but his programming notation does not seem to have been specifically related to more traditional logical notions of proof and reasoning.

The remainder of this chapter will consider in more detail the ways in which, drawing on the analogy with logic, the metalogical categories that had been developed for formal logic were applied to machine codes.

4.8 Syntax

Early automatic computers were thought of primarily as numerical calculators, and the store was correspondingly understood as a repository for numbers. With the advent of stored program machines, however, instructions were also placed in the store. This was often described as a process of coding the instructions as numbers, but this does not make explicit the fact that numbers also had to be coded before they could be stored. A variety of coding schemes had been used, even on the early relay computers [Booth, 1949]. A more accurate view of the store was as a neutral medium in which different types of information could be represented and whose "[w]ords may be interpreted as numerical information or as instructions" [Huskey, 1951].

The details of these coding schemes fall into the category of syntax, defined by Carnap as concerned only with the kind and order of symbols used in the expressions of a language, the symbols in this case being the individual digits held in the store. Accounts of specific machines typically explained how numbers were coded, and gave a description of the machine's order code in the form of a table listing the basic machine operations, accompanied by a more or less detailed account of how an instruction to the machine to perform one of these operations would be coded.

The structure of a typical order code was extremely simple. Individual orders contained a number of fields: one field specified the operation to be carried out, and other fields contained the addresses of one or more locations in the store. In addition, some codes contained digits used to verify the data stored in a word or for other internal purposes. In some cases the coded form of an instruction did not correspond exactly to the word size of the machine and some parts of the word would be left unused. Alternatively, on some machines it was possible to store more than one instruction in a single word.

Order codes were often envisaged as existing in a variety of symbolic representations. For example, in the *Draft Report* von Neumann distinguished between "short symbols" used for discussing code and setting up problems for the device, and "code symbols", which were the strings of binary digits holding instructions in the machine [von Neumann, 1945, §15.6]. Turing distinguished the "machine form" of the code both from the "permanent form", used for example to store subroutines for reuse, and also from a more readable "popular form" used when instructions were to be listed [Turing, 1946, §13]. The input tapes used in the EDSAC programming system represented addresses in decimal notation, not the binary form used inside the machine, and used a single-character mnemonic representation of basic operations [Wheeler, 1950]. The various 'popular' forms represented only the functional details of codes, ignoring for example the presence of check digits in instruction words or the details of placing multiple instructions in one word.

Machine codes had little, if any, syntactic structure above the level of the individual instruction. The sequence of instructions making up a program was usually shown by listing actual or illustrative memory locations and showing the instruction stored at each. These memory locations, however, were those denoted by the addresses appearing in individual instructions: the overall program structure could therefore only be grasped by referring to an aspect of the meaning of the code, and not through purely syntactic means. It was impossible, in other words, to understand what a program did by simple inspection of the instructions making it up: it was also necessary to know where in memory these instructions were stored.

There was very little theoretical analysis of the syntax of machine code, a more pressing concern being the best choice of basic operations for a code. The most widely discussed syntactic issue concerned the number of address fields contained in a single instruction. Codes which contained three addresses allowed a single order to express an instruction like "add the numbers stored in locations x and y and store the result in location z". In a code which provided only a single address field, this would require three instructions: "add the number stored in location x into the accumulator; add the number stored in location y into the accumulator; transfer the number stored in the accumulator to location z". A further variant, to support optimum coding, allowed the address of the next instruction to be stored explicitly in each instruction, leading to two and four address codes.

There appeared to be no clear advantage, in terms of overall code size or execution time, between one and three address codes, and both schemes were widely adopted. A theoretical result to this effect was published by Calvin Elgot in 1954; this is of interest as being an early application of formal language theory to computer programs [Elgot, 1954]. Elgot's proof involved the definition of a formal language intended to represent the relevant differences between the two forms of code, but did not, however, give a formal syntactical description of a complete or realistic machine code.

The most notable application of logical syntax to computers at this time was made by George Patterson, who explicitly drew on Carnap's work to outline a general theory of "syntactical machines" [Patterson, 1949]. He described a class of machines, described as "linguistic transducers", which accepted input data and transformed it into output data. By viewing this data as symbolic expressions, Patterson hoped to use Carnap's approach to develop a logical theory of such machines. Analogue computers were ruled out from this treatment, but the class of syntactical machines was wider than just digital "calculating machines", and Patterson listed a number of other machines, including cryptographic machines and switching systems, to which his approach could be applied.

Patterson listed a number of problems whose solution he felt could be aided by a unified syntactical approach. These included problems in machine analysis and synthesis, as well as the design of suitable order codes for machines and the coding of specific problems. The applications described in the paper were concerned with formalizing and verifying properties of basic electronic circuits for computer arithmetic, however, and Patterson described no applications of his ideas to the formalization of machine code or the construction of programs.

An interesting terminological difference between Patterson and Carnap marks the shift to the application of the ideas of logical syntax to a type of formal language very different from conventional logic. For Carnap, transformation rules are intended to capture aspects of the consequence relation between complete sentences. Patterson does not explicitly describe a class of sentences however, nor discuss relationships between sentences. Instead, he is concerned with a class of "numerical expressions", or numerals in a specified base, and he gives a recursive definition of a "quasi successor" relation between numerical expressions. This definition is subsequently referred to as a "transformation rule", a usage which clearly marks a break with logic's concern with truth and consequence. A more subtle difference is that for Carnap transformation rules provided a way of capturing non-recursive relationships such as logical consequence. In Patterson's usage, however, the term is used as a synonym for a recursive definition, suggesting that in the context of computers, the only transformations of interest are those that are computable, or definable by recursive functions.

4.9 Semantics

Perhaps the most obvious interpretation of the meaning of a program is that a program denotes the computation that it performs, or more concretely, the sequence of operations performed by a computing machine when running the program. In 1947, Goldstine and von Neumann elaborated a sophisticated account of this notion of program semantics, together with a notation of flow diagrams which expressed such meanings and could be used as a way of developing a program.

They began by pointing out the complexity of the relationship between the instructions in a program and the mathematical operations performed when it was executed. This was due to certain features of the order code, such as transfers of control which caused variations in the written sequence of instructions, and address modification which meant that the sequence of instructions itself could be expected to vary as the computation proceeded.

These observations highlighted two significant differences between the tasks of giving a semantic account of predicate logic and of programs. In logic, semantics can be characterized as a mapping from a stable, syntactic expression to some domain of meanings. Because of the possibility of a program modifying its own instructions, however, the situation with programs is more complex. Executing a program can cause the program text itself to change, and the meaning, in the sense of the operations subsequently carried out, depends on the modified text, and hence only indirectly on the original program text. In other words, "coding is . . . the technique of providing a dynamic background [i.e. the changing instructions] to control the automatic evolution of a meaning [i.e. the operations automatically carried out by the computer]" [Goldstine and von Neumann, 1947, p. 154].

Secondly, semantics for logic are typically compositional: the meaning of an expression can be derived in a systematic way from a knowledge of the meanings of its constituent subexpressions and the way they are put together. Machine code programs could not be understood in this way, however. Even in the simplest case of two adjacent orders it could not be concluded that the operations they denote will be performed in sequence: even leaving aside the possibility of instruction modification, a transfer from elsewhere in the program might cause the second to be executed independently of the first.

Faced with the relative obscurity of the relationship between program code and the operations executed by the program, Goldstine and von Neumann developed a diagrammatic formalism to help in the development of programs. The method they proposed was "to plan first the course of the process and the relationship of its successive stages to their changing codes, and to extract from this the original coded sequence as a secondary operation" [Goldstine and von Neumann, 1947, p. 84]. The flow diagram notation they developed was therefore intended to provide a graphical representation of the behaviour of the running program, the required sequence of operations, a "schematic of the course of C [the control] through that sequence". In effect, the flow diagrams were a technique for expressing the semantics of a program, and Goldstine and von Neumann proposed a development method which would derive from this a sequence of orders which when executed would result in the required operations being carried out.

Flow diagrams as proposed by Goldstine and von Neumann were directed graphs in which the nodes represented groups of operations that were always executed in the default, sequential order; the arcs represented transfers between these blocks of operations. A node with two arcs leading from it represented a block with two possible continuations, and hence a conditional jump, and loops were represented by cycles in the graph. Because of the possibility of the dynamic modification of instructions, however, the structure of the graph might change as the program ran, and in an attempt to deal with this, Goldstine and von Neumann introduced so-called "variable remote connections" into the flow diagrams. These were intended to show that particular arcs should be considered to link different pairs of nodes at different times.

On top of this basic expression of structure, flow diagrams contained a lot of information about the properties of the data being manipulated by the program. A strict distinction was maintained between the mathematical expression of the problem being coded, described in terms of *variables*, and the actual data being manipulated, which was referred to by reference to *storage locations*. The operations required, and the numerical values stored at each point of the program's execution, were described mathematically. A distinction was drawn between free and bound variables, the terminology being explicitly linked with that of "formal logics" [Goldstine and von Neumann, 1947, p. 91]; free variables were those whose value could be set from outside the routine being considered, and bound variables those which were considered 'private' to the routine. Unlike in logic, however, where variables become bound by being used in a quantification, there was no syntactic means of distinguishing free from bound variables in the flow diagram notation, the distinction being purely contextual.

Flow diagrams used *assertion boxes* to state properties that were expected to hold at various times during program execution. This formed a bridge between programming and traditional logical notations: an assertion could be any logical formula making use of the program variables. These assertions could be understood by treating the mapping between program variables and the corresponding stored values at the moment when the assertion came into effect as giving an interpretation of the variables. The use of assertions was also adopted by Turing in 1949 as a method for checking correctness of programs [Turing, 1949], but then faded from view until reemerging in the mid-1960s. A third difference between programs and conventional logic can be noted at this point, namely that whereas the meaning of a predicate logic formula is given with respect to a single interpretation, or mapping from variables to objects, the interpretation given to the variables in a program changes as the program executes.

The flow diagram notation developed by Goldstine and von Neumann can therefore be viewed in part as an attempt to assimilate machine code programming to the theory and practice of conventional logic. Flow diagrams are an attempt to create a formal representation of program semantics, conventional logic in the form of assertions is built in to the notation and to the methodology of program construction based upon it, and analogies are drawn with logic even in relatively minor details of terminology, such as the distinction between free and bound variables.

In later comments on this work, Arthur Burks described the use of "bound variables" in loops as being related to the bounded quantifiers introduced by Gödel [Aspray and Burks, 1987]. This observation derives from the fact that in many programs, variables are used to control loops by counting the number of iterations that the program has made through the loop. In a similar way, the variables in bounded quantifiers index all the integers in the range of the quantifier. The use of variables to control loops was a universally adopted programming technique, however, and clearly related to the variables used in interactive schemes for manual computation. It does not seem likely that this feature of programs needs to be explained by reference to formal logic.

A striking feature of the flow diagram notation is that it constrained the freedom theoretically available in programming for stored program machines. Flow diagrams were well adapted to express high-level computational structures such as loops and conditional branching, but only permitted the expression of a limited form of instruction modification, by means of the variable remote connections. It is not clear that an arbitrarily complicated program could be perspicuously depicted in a flowchart. Furthermore, flow diagrams seem most suitable for describing the flow of control within a single routine, and do not appear to have been used to show the high-level structure of a program as a set of subroutines, or the calling relationship between subroutines.

The flow diagram notation was widely adopted, but usually in a simpler form than that proposed by Goldstine and von Neumann. For example, in 1949 Renwick used flow diagrams to explain an example program for the EDSAC [Renwick, 1949]. However, the diagrams showed only operation boxes and alternative boxes, and the connections between them: no distinction was made between program variables and storage locations, and assertions were not used. With the exception of Turing's paper [Turing, 1949], these more 'logical' aspects of the notation were not on the whole taken up.

4.10 Programs as metalinguistic expressions

As noted above, Arthur Burks later speculated on the relationship between Goldstine and von Neumann's work and some of Gödel's logical ideas, writing that "I think it likely that, in his programming work, von Neumann was guided by his knowledge of Gödel's work, at least intuitively" [Aspray and Burks, 1987, p. 384-5]. In particular, Burks saw in the ability of stored data to refer either to a number or an instruction "an instance of the metalanguage versus object language distinction" [Aspray and Burks, 1987, p. 385]. The situation is slightly more complicated than this, however.

As discussed in Chapter 2, Turing adopted Gödel's strategy of arithmetization to encode machine tables as data which could be stored on the tape of the universal machine, and the same strategy is adopted by stored program computers. Rather than being an instance of the *distinction* between object and metalanguage, however, this means that a stored program can simultaneously be viewed as belonging to the object language, when it is manipulating numerical data for example, and the metalanguage, when it is modifying its own instructions.

The possibility that a language could express its own syntax by means of arithmetization initially appeared paradoxical, and one result of Carnap's work was to show that in the case of conventional logic this gave rise to no problems [Carnap, 1937]. A program which is capable of modifying its own instructions, however, seems to take a step beyond what is possible in logic, and to further blur the distinction between syntax and semantics. For Carnap, syntax was concerned only with the classification and ordering of symbols in expressions, uncontaminated by any considering of the meaning of the expressions. If the meaning of a program is considered to be the operations it carries out, however, then instruction modification represents an infection of the syntactic domain by semantics: execution of the program is able to change the syntactic representation of the program itself.

For many people in the late 1940s and early 1950s, the possibility of self-modifying programs was an extremely significant feature of the stored program design. This was so for both practical and theoretical reasons. The ability to change instruction addresses made the coding of iterative programs much easier and more flexible than it had been on machines such as Mark I, but self-modification was also invoked, for example by Turing, as having the potential to explain higher cognitive functions such as the ability to learn.

This chapter has described the different approaches adopted to instruction modification in early order codes, and it is striking that von Neumann consistently adopted a conservative attitude towards it. This conservatism can be understood as an effect of applying the metalogical structure created for conventional logic to the new ‘logics’ of computer codes, and in particular as an attempt to keep separate the domains of syntax and semantics. Further evidence in support of this line of thought will emerge in subsequent chapters, which describe the simultaneous emergence of programming language theory modelled on metalogic and the elimination of the ability to write self-modifying programs.

Leaving aside the issue of instruction modification, there were other programs which could more straightforwardly be described as metalinguistic, as Burks suggested. In logic, a metalanguage provides the capability to express the syntax and possibly also the semantics of another language. Programs do not make statements, and so cannot declaratively represent syntax in the way logical metalanguages do, but it would appear reasonable to describe a program which can manipulate syntactic representations of other programs as metalinguistic.

Examples of such metalinguistic programs appeared early on, the best documented early example perhaps being the Initial Orders for the EDSAC, a program which translated input programs expressed as a mixture of letters and decimal numbers into a purely binary form [Wheeler, 1950]. This approach was quickly recognized as providing great benefits in programming productivity: a number of systems defined ‘interpretative codes’ for various purposes such as floating point arithmetic, for example. These enabled the programmer to write code in one notation which would be translated by an interpretative program into the machine's native code. This approach was generically known as 'automatic programming', and throughout the 1950s many such codes were produced, gradually evolving into what became known as programming 'languages'. This development is discussed in detail in the next chapter.

4.11 Conclusions

The *Draft Report* is widely recognized to have marked a turning point in the development of computer hardware. It represents a moment of closure, where a number of elements, largely present in earlier work, were for the first time put together in a form that became a definitive model for most if not all subsequent developments.

The role of the report in the development of programming techniques is less dramatic, however. This chapter has described the gradual evolution of the basic concepts of machine code programming from the late 1930s to about 1950, and it is apparent that there is a much greater continuity between the way in which Mark I and the EDSAC were programmed, say, than there is in their hardware. The nearest analogue to the *Draft Report* in the field of programming is perhaps the textbook written by the EDSAC group [Wilkes et al., 1951]. Like the *Draft Report*, this summarized in a particularly clear form the principles on which contemporary programming was based, and served as a model for much later work.

Although there was at this period little theoretical reflection on order codes, a connection was made between these codes and formal logic, particularly by Turing and von Neumann. Awareness of this connection inspired a certain amount of rather unsystematic research into the application of metalogical ideas to machine codes. The most significant attempt was that of Goldstine and von Neumann to give a semantic account of programs using flowcharts. The immediate influence of this work, however, seems to have been rather limited.

Chapter 5

Programming notations as formal languages

It quickly became apparent that the task of creating machine code programs was one that most humans would find very taxing, and techniques for simplifying and automating parts of this process were soon developed. Symbolic abbreviations for operation codes were frequently defined and some programming systems, such as the EDSAC, provided special programs to translate the symbolic form into the internal machine representation automatically.

These developments automated certain aspects of the production of machine code, but still required programmers to define the sequences of basic operations making up the program. A further stage of automation was envisaged in which this task, described as 'programming' to distinguish it from the more mechanical activity of 'coding', would itself be performed by machine. Stanley Gill expressed the goal as follows: "One might say that an ideal programming scheme would allow one merely to state the problem to be solved . . . existing systems . . . still require the user to specify a series of steps to be performed by some conceptual computer" [Gill, 1959].

This problem was first addressed in connection with mathematical formulas. A number of systems were developed which allowed programmers to include in programs formulas written in some approximation to standard mathematical notation; these formulas would then be automatically translated into a sequence of instructions to perform the desired calculations. The most ambitious and successful of these systems was Fortran, which first became available in 1957 for the IBM 704 machine [IBM, 1956].

By the end of the 1950s many automated programming systems existed, most designed for and implemented on a particular type of computer. A number of groups had discovered the benefits of sharing programs, but the existence of many different programming notations made this difficult. This situation gave rise to a number of initiatives aimed at developing a universal programming notation; the best-known and most influential of such developments was the Algol 60 language [Naur et al., 1960].

This chapter examines these technical developments and the parallel evolution in theoretical accounts of programming notations. At the beginning of this development, programming notations were understood relative to a machine, whether real or imaginary; at the end, they were thought of free-standing notations, or ‘languages’, which could be studied independently of any machine. Both natural languages and formal languages were taken as models for programming languages. Whereas natural languages inspired developments in notations intended for use in data processing applications, formal logic was taken as a model for programming languages intended for mathematical and theoretical uses, such as Algol and Lisp.

5.1 Automatic coding

At the beginning of the 1950s, the term ‘coding’ was used to refer to the process of translating the instructions of a program into the coded form used inside the machine. In some cases this was carried out entirely by hand, but following the example of the EDSAC [Wheeler, 1950], many installations devised a set of ‘initial orders’ which would translate instructions from a more human-friendly form into machine code.

In simple cases, this translation required little more than a correspondence between symbols and codes, but the use of subroutines made the task more complex. If subroutines are to be reused freely, it must be possible to place their instructions at different locations in the store in different programs. However, this means, for example, that instructions in the subroutine that refer to specific storage locations or jumps to another instruction within the subroutine, will differ from one occasion of use to another. The EDSAC’s initial orders automated the process of calculating the required addresses, so that subroutines were correctly translated depending on their location in any given program. A further refinement was provided by an ‘assembly subroutine’ which calculated the location of each subroutine in a program, so that the programmer would not have to decide where in the store each subroutine should be placed.

These and related techniques, such as ‘floating addresses’ [Wilkes, 1953b], were based on manipulating a program before it was run. From an input tape consisting of a master routine and a number of subroutines, a complete translated machine code program would be produced, and then executed. An alternative approach made use of so-called ‘interpretive routines’. When an interpretive subroutine was called, the processing to be carried out was specified by a number of ‘pseudo-orders’, or instructions that in fact did not belong to the machine’s order code. The job of an interpretive routine was to read these pseudo-orders and ensure that appropriate code was executed in response to each one. In contrast with the approaches described above, the pseudo-orders were not translated in advance into machine code; instead, the interpretation process was carried out during program execution.

For example, the EDSAC subroutine library contained interpretive routines to facilitate calculations with complex and floating-point numbers. The codes interpreted by these routines bore a very strong relationship to the basic machine code, being in the same format and even using the same code letters to refer to analogous operations in most cases [Wilkes et al., 1951]. Presumably this was intended to make the use of the subroutines as natural as possible to programmers, as well as allowing input of the interpreted codes without having to change the initial orders [Campbell-Kelly, 1980, p. 29]. More generally, interpretive routines raised the possibility of designing codes that were adapted for specific purposes, and which would therefore diverge further from the underlying machine code. Wilkes and his collaborators give an example where the instructions in the interpreted code were so small that two orders could be placed in a single machine word [Wilkes et al., 1951, p. 162–164].

In the EDSAC system, the interpretive routines were intended to be called as part of a larger program, and the interpreted codes therefore formed only part of the complete program. In effect, a single program could be written using an extended code, where the basic order code was supplemented by the pseudo-orders handled by one or more interpretive routines. An alternative approach was to enable an entire program to be written in a single interpreted code. The earliest such system to have been implemented appears to have been the so-called ‘Short Code’, which first ran on the UNIVAC in 1950 [Schmitt, 1988, p. 11].

Short Code evolved from a proposal made by John Mauchly in 1949 to develop a “special code chosen to simplify the work of the human programmer and throw much of the tedious detail of coding onto the computer” [Mauchly, 1949]. Mauchly’s argument in favour of an interpreted code was primarily economic: he identified a class of “small” problems where the cost of programming far outweighed the cost, in terms of computer time, of running the program. He anticipated that the use of an interpreted code would make programming easier, and therefore significantly reduce the overall cost of such programs.

A disadvantage with interpreted codes was that the translation to machine code was performed as the program was running, and therefore increased the time taken to run programs. An alternative approach was to perform the translation as a separate step before running the program. Wilkes described this approach as follows: “[t]he programmer writes down ‘orders’, here called *synthetic* orders, which the control circuits of the machine are incapable of executing. The necessary expansion into sequences of ordinary machine orders . . . takes place once for all in advance of the execution of the programme” [Wilkes, 1952]. Wilkes gave an example of synthetic orders designed for the EDSAC, but stated that the technique had not yet been used to any significant extent.

Related developments, associated particularly with Grace Hopper, were being carried out on the UNIVAC. Again motivated by a growing awareness of the cost of programming, Hopper hoped that “[t]he programmer may return to being a mathematician” [Hopper, 1952, p. 244]. This was to be achieved by what Hopper called “compiling routines”, which were “designed to select and arrange subroutines according to information supplied by the mathematician or by the computer” [Hopper, 1952, p. 248]. A series of such routines were developed and ran on the UNIVAC from 1952 onwards.

From 1950 on, then, a wide variety of schemes and tools were developed to implement various approaches to automatic coding. Interpreted and compiled schemes shared the property that programs were not written directly in the code of the target machine, but in a *pseudo-code*. It was widely hoped that this would make programming easier and less time-consuming, although potentially decreasing the run-time efficiency of the machine. This trade-off was widely seen as having overall economic value.

5.2 Virtual machines: the semantics of pseudo-codes

Machine codes were, naturally, understood as being notations for expressing, or specifying, the behaviour of particular computing machines. Pseudo-codes however broke the correspondence between instructions and machine operations, so a more complex understanding of the meaning of pseudo-code programs was required.

Syntactically, pseudo-codes were often very similar to actual machine codes: “[n]o example of a programme of interpretive orders need be given since it would look just like an ordinary programme" [Wilkes, 1952]. In these cases, pseudo-codes were treated as extensions of machine codes: "the [interpretive] sub-routine executes the 'orders' in the list in a similar fashion to the way that the machine obeys ordinary orders" [Wheeler, 1952]. Even simple subroutines could be understood in this way: "[b]y deciding to place a closed subroutine in the store, the programmer effectively extends the order code of the machine so as to cover the operation performed by the subroutine" [Wilkes, 1952].

Unlike the EDSAC interpretive codes, Mauchly's Short Code was syntactically quite distinct from machine code. Nevertheless, it was initially understood in a similar way, as a more powerful code than that understood directly by the machine. Mauchly wrote that a computer "may be made to interpret and execute instructions given in the simple code" [Mauchly, 1949], and in a similar vein Wilkes and his colleagues stated that "the use of interpretive routines effectively extends the order code of the machine by increasing the complexity of the operations which may be performed in response to a single 'order"' [Wilkes et al., 1951, p. 35].

An alternative interpretation was available, however, in which an interpretive routine was understood as an extension or modification not to an order code, but to the underlying machine itself. Turing put this as follows "[a]n interpretive routine is one which enables the computer to be converted into a machine which uses a different instruction code from that originally designed for the computer" [Turing, 1951, p. 192]. These two interpretations were aligned by Earl Isaac: "[t]he use of subroutines permits the coder to think in terms of functions that are complex combinations of the elementary arithmetic and logical operations of the machine. This is in effect a different structure than that permitted by the basic machine" [Isaac, 1952].

The second interpretation gained some of its force from the desire to simulate on one machine the hardware of other, more powerful machines. For example, the floating-point interpretive routine designed by Brooker and Wheeler simulates in the EDSAC's memory a floating point accumulator and the so-called 'B tube' developed in Manchester [Williams, 1951, p. 176], and permits recursive subroutine calls, even though these capabilities were not provided by the EDSAC's hardware [Brooker and Wheeler, 1953]. John Backus may have been thinking of this case when he later wrote that "[t]he purpose of the early systems was to provide synthetic machines which had floating-point operations and often index registers (B-tubes), since the real machines did not" [Backus, 1958, p. 234]. Backus himself had designed an interpretive scheme for the IBM 701, introducing it in terms of the "synthetic" machine that it simulated: "[t]he IBM 701 Speedcoding System is a set of instructions which causes the 701 to behave like a three-address floating point calculator. Let us call this the Speedcoding calculator" [Backus, 1954].

Compilers as well as interpretive routines were understood as creating synthetic machines. Hopper later described the compiling routines by saying that "the compiler . . . effectively converted the UNIVAC from a single-address, fixed-decimal computer into a three-address, floating-decimal computer" [Hopper, 1959, p. 167]. The introduction of later pseudo-codes was commonly explained or motivated by an appeal to the notion of a synthetic machine. For example, Laning and Zierler, whose system is described in more detail below, wrote that "[t]he effect of our program is to create a computer within a computer . . ." [Laning Jr. and Zierler, 1954, p. 1], and in a later description of programming the DEUCE, the descendant machine of Turing's ACE, Robinson wrote that "it is constructive to look upon [three interpretive schemes] as three alternative machines" [Robinson, 1960, p. 115].

Pseudo-codes, then, came to be understood in the same way as machine codes, namely as being the instruction codes of particular computing machines. The machine corresponding to a given pseudo-code would usually not have been built, however, but would be simulated on an existing machine. When a pseudo-code program was run, the job of the interpreter or compiler was to ensure that the same results were produced that would be obtained if the 'synthetic', or virtual, machine assumed by the writer of the pseudo-code had been operational and the pseudo-code program run directly on it.

As noted above, Turing was an early advocate of this point of view, and the idea of an interpretive routine enabling one machine to simulate another later became associated with the universal machine concept:

> the founder of [the field of automatic programming] was the late A. M. Turing, who . . . first enunciated the fundamental theorem upon which all studies of automatic programming are based . . . it states that any computing machine which has the minimum proper number of instructions can simulate any other computing machine, however large the instruction repertoire of the latter. All forms of automatic programming are merely embodiments of this rather simple theorem [Booth, 1960]

It is debatable whether this 'theorem' was in fact stated by Turing, however. Turing demonstrated the existence of a universal machine within a certain class of machines which shared the same physical structure. He only argued informally, however, that the machines he had defined were capable of simulating all forms of computational machinery, commenting for example that a two-dimensional grid of data values could be represented on a one-dimensional tape.

A number of the linguistic features of pseudo-codes of this type are worth noting. Firstly, the idea of self-modifying code lessened in importance, or at least became something that the programmer no longer had to worry about explicitly. For example, Backus listed as one of the features of Speedcoding that it provided "automatic address modification" [Backus, 1954] handled by the interpreter and not by the programmer.

Secondly, the idea of pseudo-codes extending machine code introduced a rather vague notion of subroutines existing at different levels. Sometimes this was simply a question of whether one routine called others, or was itself written purely in machine code, but more ambitious proposals were also put forward. For example, Hopper described the basic compiling routines as being of 'Type A', and described more sophisticated routines which appeared to be capable of writing new subroutines: "the mathematician ... sends the information defining the function itself to the UNIVAC. Under the control of a 'compiling routine of type B' ... the UNIVAC delivers the information necessary to program the computation of the function and its derivatives" [Hopper, 1952, p. 244]. Higher levels were also envisaged: "[t]ype B routines at present include linear operators. ... It can scarcely be denied that type C and D routines will be found to exist adding higher levels of operation" [Hopper, 1952, p. 249].

Thirdly, the job carried out by interpretive routines and compilers was frequently described as one of translation. For example, in 1951, Jack Good asked whether anybody had "studied the possibility of programme-translating programmes, i.e. given machines *A* and *B*, to produce a programme for machine *A* which will translate programmes for machine *B* into programmes for machine *A*" [Good, 1951]. Translation is usually conceived as a meaning-preserving relationship between expressions in distinct languages. Applying the metaphor of translation to interpretive routines encouraged people to think of programming codes as languages in their own right. In 1952, a group at Manchester described their work on developing a code for a new machine in precisely these terms: "[w]e are ... developing a scheme which will enable us to test the new programmes on the old machine and this will be done by means of an interpretative [sic] scheme which translates the new routine from the new code back into the code of the existing machine" [Bennett et al., 1952], and in the same year Earl Isaac offered the general opinion that "[c]oding for digital computers is a process of translating from one language to another" [Isaac, 1952].

Towards the end of the 1950s, Gill summarized the way in which pseudo-codes were understood, explicitly bringing together the notions of translation and virtual machines: “[t]he net effect may be looked on either as a translation of the original program language into that required by the machine, or as a way of making the machine imitate another machine which recognizes the original language directly” [Gill, 1959, p. 111].

5.3 Formula translation

In some ways, programming in a pseudo-code was a similar experience to machine code programming. Problems had to be broken down into small steps which could be expressed as individual instructions in the code being used, whether or not it was the code of a real machine. This low-level coding soon became seen as a routine and rather unrewarding task, but one for which there was an increasing demand as applications of computers became more widespread. One strategy to address this situation was to make programming more interesting and accessible by using programming notations that were more related to the problems that users were trying to solve.

In the 1950s, this approach was applied with considerable success to the specific task of evaluating mathematical formulas. A basic step in many calculations is to use values already calculated to compute the value of a new variable. Such steps can be formalized as equations of the form $x = F(y, z, \ldots)$, where x is the variable to be computed and F is a formula expressed in terms of known values. Many such formulas can be interpreted as expressing algorithms, specifying what arithmetical operations have to be carried out and in what order. Rather than translating this algorithm into code by hand, it seemed that it should be possible to have the computer itself generate the coded instructions. In addition to the perceived economic benefits of using interpreted codes, this raised the possibility of allowing mathematicians to program computers directly using a familiar notation, thus reducing the demand for skilled coders.

Automatic formula translation seemed more challenging technically than the interpretation of pseudo-codes. Earl Isaac broke it down into two steps, the “translation of grammar” producing a sequence of instructions coding the operations required to perform the calculation, and the “translation of words” generating machine code, for example by replacing variable names by machine addresses. He noted that the translation of grammar appeared to be the harder problem and one on which little progress had been made [Isaac, 1952].

Developers of formula translation systems in the early 1950s made different decisions about the ‘grammar’ of formulas. Some assumptions were widely shared, for example that formulas should resemble standard mathematical notation as far as possible, using the four basic arithmetical operations and, where necessary, parentheses to control the order of evaluation, and that it should be possible to include standard functions, such as trigonometric and exponential functions, in formulas. Systems varied greatly in detail, however, partly as a result of the different ambitions and goals of their authors, and partly because of technical difficulties uncovered while writing a program to translate formulas.

For example, a program written in the UNIVAC’s Short Code consisted of a number of statements, such as $X = Y + ZW$ [Schmitt, 1988]. The interpreter associated numerical values with variables, and a statement allowed these values to be used to calculate a new value. Initially, expressions could use the only the four basic arithmetical operators and parentheses, but other operators and functions were soon added. Equations were transliterated by hand and presented to the interpreter in coded form; the interpreter would then scan the expression, replacing variables by their current values and calling the appropriate subroutine whenever an operator or function was encountered. Multiplication was expressed by the juxtaposition of variables, presumably with the intention of making the code resemble conventional notation; in general, however, the behaviour of the interpreter had to be taken into account when writing expressions, to ensure that the expected value would be computed.

A later system, developed by Laning and Zierler at MIT, allowed a more natural use of mathematical notation, interpreting equations as complex as $z = 1 - zx^2/y(y - 1)$ correctly, although the system was limited to four levels of nested parentheses [Laning Jr. and Zierler, 1954]. A wide range of functions were predefined as subroutines and subscripted variables could be used. This system went beyond the evaluation of single expressions, and certain systems of differential equations could be solved automatically. For example, the system $dy_1/dt = y_2 + 1$, $dy_2/dt = -y_1$ could be solved by writing the following two equations in a program:

```
D y|¹ = y|² + 1
D y|² = – y|¹
```

The different formula translation systems proposed in the 1950s, therefore, varied considerably in what grammatical forms they interpreted. At one extreme, the autocode for the Pegasus computer permitted only one operator to be written in each formula. It was argued that this made the code very easy to learn [Felton, 1960]. Other systems, like that of Laning and Zierler, went beyond the evaluation of functional expressions in various ways. For example, formula translation systems were based on the fact that mathematical formulas encode algorithms which can be translated into machine code. A number of authors pointed out, however, that an 'implicit' formula such as $y - 2 = 3x$ encodes an algorithm for working out the value of y just as clearly as the equivalent 'explicit' formula $y = 3x + 2$. There seemed to be no reason in principle why translators could not be written to generate machine code from implicit formulas, and techniques for doing this were proposed [Cleave, 1960].

By contrast, later programming languages show much less variety, supporting little if anything more than functional expressions written using arithmetical operators, parentheses and calls to subroutines. This 'standard form' first appeared in the Fortran language: the next section will describe this, and offer an explanation for the subsequent success of this form.

5.4 Fortran and the definition of expressions

Fortran's definition of expressions can be characterized by two features. Firstly, the language only supported 'explicit' equations:

> A FORTRAN arithmetic formula resembles very closely a conventional arithmetic formula; it consists of the variable to be computed, followed by an = sign, followed by an arithmetic *expression*. For example, the arithmetic formula
>
> ```
> Y = A-SINF(B-C)
> ```
>
> means "replace the value of y by the value of a-sin(b-c)" [IBM, 1956, p. 12].

This definition distinguishes two aspects of the formula, namely the specification of the calculation to be performed and the identification of the storage location that is to hold the resulting value. A number of writers appear to have found this distinction a potential source of misunderstanding, because of the possibility of writing formulas such as $n = n + 2$. Viewed from the mathematical point of view as an identity, such an equation is meaningless, or at least has no solution. From the computational point of view, however, it defines a straightforward procedure, as the Fortran definition explains.

> The = sign in an arithmetic formula has the meaning "is to be replaced by". An arithmetic formula is therefore a command to compute the value of the right-hand side and to store that value in the storage location designated by the left-hand side. [IBM, 1956, p. 16]

This computational interpretation of an apparent equation, however, rules out the possibility of the system handling implicit equations such as $y - 2 = 3x$.

The second significant feature of Fortran's definition of formulas is the way in which the syntactic form of expressions was specified. Unlike many other systems, the Fortran manual gave a recursive definition of expressions which is very similar in style to the definitions of the terms of formal languages given in logic texts. Constants, variables and subscripted variables are first defined, and then the following definition of expressions is given; note that expressions could be of either fixed or floating point mode, but this does not affect the discussion.

> *Formal Rules for Forming Expressions* By repeated use of the following rules, all permissible expressions may be defined.
>
> 1. Any fixed point (floating point) constant, variable, or subscripted variable is an expression of the same mode. Thus 3 and I are fixed point expressions, and ALPHA and A(I,J,K) are floating point expressions.
>
>   ...
>
> 4 If E is an expression, then (E) is an expression of the same mode as E. Thus (A), ((A)), (((A))), etc. are expressions.
>
> 5 If E and F are expressions of the same mode ... then
>
> $$\begin{array}{ccc} \mathsf{E} & + & \mathsf{F} \\ \mathsf{E} & - & \mathsf{F} \\ \mathsf{E} & * & \mathsf{F} \\ \mathsf{E} & / & \mathsf{F} \end{array}$$
>
> are expressions of the same mode. ...The characters $+$, $-$, $*$ and $/$ denote addition, subtraction, multiplication and division. [IBM, 1956, p. 14]

The influence of formal logic seems clear here; the definition of expressions and their translation into machine code was largely the work of Peter Sheridan [Sheridan, 1959], who before joining IBM had completed a Masters degree in logic [Weiss, 1993]. It is striking that the recursive definition is given in the manual intended for programmers to read: this brought a level of precision to the definition of programming notations that was at the time unusual.

An important feature of this definition is its generality: there is no question of restricting expressions to a fixed number of operators or levels of parentheses. In one respect the syntax diverges from normal mathematical usage: whereas other systems allowed multiplication to be represented by juxtaposition, as in XY, in Fortran it must be represented explicitly as $\mathsf{X} * \mathsf{Y}$. Both these properties are consequences of the recursive definition, and this suggests that in some respects the desire for formal consistency was taking precedence over other goals, such as preserving conventional notation.

This raises the question of why this particular definition of expressions turned out to be so influential. It is tempting to answer this question by pointing to the success of Fortran and the consequent adoption of many of its features in later languages. The explanatory power of this answer is limited, however: many other features of Fortran were not so influential, and the language has subsequently changed in many ways, incorporating features derived from later languages and research. What was special about the definition of formulas given by Fortran that might account for its differential success and persistence?

One possible answer is that it was precisely the use of logic that gave rise to the success of the Fortran definition. Although a number of writers had perceived a general similarity between logic and programming, this was the first time that techniques from formal logic had been applied to a relatively mundane task like syntax definition. As well as providing a concise and general definition of expressions, this suggested a general approach to the design of programming languages, one which made use of the authority and established results and techniques of the discipline of logic.

At around this time, a reciprocal interest in programming notations was developing among logicians. The Summer Institute for Symbolic Logic, held at Cornell University in 1957, included a number of papers on computer-related topics, including programming notations, formal representations of computing machines, and mechanical theorem proving. These included a short talk by Sheridan describing the Fortran system [Sheridan, 1957].

Fortran's use of logical techniques was limited to definition of expressions, however, and the syntax of the remainder of the language was not given a recursive definition. In other respects, too, Fortran occupies an intermediate position between pseudo-codes and formal languages: despite being described as a language, it was viewed as an integral part of a wider system whose role was to "transform ... the 704 into a machine with which communication can be made in a language more concise and more familiar than the 704 language itself" [IBM, 1956, p. 2]. This description situates Fortran firmly in the semantic tradition of 1950s pseudo-codes, as described above. Furthermore, the design of many of the features of the language was influenced not by logic but by the desire to produce object code that was as efficient as possible [Backus and Heising, 1964].

5.5 Universal languages

By the end of the 1950s, a large number of automatic programming systems had been developed, and surveys showed that the overwhelming majority of them were only available on a single type of machine [Bemer, 1959]. At the same time, user groups for particular machines were discovering the advantages of being able to share programs and were beginning to distribute routines among the user community. For example, the group SHARE was started in 1955 as a "cooperative programming group for IBM 704 users" [SHARE, 1958a]. As the name chosen for the group implies, one aim of the group was to enable users to benefit from the work of other programmers. For example, in 1958 "SHARE agreed to accept for distribution self-contained routines in FORTRAN language. However, since appropriate conventions were not agreed upon, it was decided to defer distribution of subroutines for the time being" [SHARE, 1958b].

This last comment indicates recognition of the fact that for code to be easily shared between different groups, common languages and standards had to be agreed. Whereas mathematical formulas provided a standard notation for encoding simple computational procedures, traditional mathematics defined no universally accepted method for expressing the sequencing of operations in more complex algorithms. Machine code programming did provide a structure for expressing computations, but both machine codes and the later pseudo-codes were highly machine specific. Fortran marked a significant step forward: it was initially only available for the IBM 704, but by the early 1960s Fortran compilers had been written for a wide variety of machines.

A number of machine-independent programming notations had in fact been defined. Many of these originated in Europe, in circumstances suggesting that lack of easy access to a actual working machine was a factor in encouraging more theoretical work. For example, as early as 1948 Zuse had published a short description of his *Plankalkül* notation, based on work he had carried out immediately after the war [Zuse, 1948]. These proposals do not appear to have influenced the development of programming notations, however, in part because they could not be used on contemporary technology.

By the latter half of the 1950s, however, it had become feasible to experiment with new notations by writing an interpreter, and in some cases this even led to the construction of new machines based on the order code suggested by the new notation. Often, these experimental proposals were explicitly related to logic. For example, in 1957 Charles Hamblin observed that formula translation schemes were only necessary because of the obscurity of machine code, and reasoned that a better solution would be to design a machine whose basic operations were better adapted to the needs of programmers. He viewed this as "primarily a problem in applied formal logic" [Hamblin, 1957, p. 135], and proposed using an adapted version of a notation introduced by Łukasiewicz, which he dubbed ‘reverse Polish’ notation. As presented by Hamblin, this notation had the properties that every symbol could be viewed as denoting a machine operation and that an expression could be evaluated by performing the specified operations in the same order as the symbols were written in the expression. After being used in interpreted form on the DEUCE computer [Hamblin, 1958], Hamblin’s ideas for a so-called ‘zero-address’ computer were implemented in the architecture of a later computer, the KDF9.

Another factor leading to diversity in the field of programming notation was the perception that different notations were required for different application areas. Fortran was the first language to be implemented on more than one computer, and it became a *de facto* standard for scientific programming. It was felt that the language was too mathematically oriented for business users, however, whose primary concern was data processing, and a number of proposals were made aimed specifically at such users. Specialized requirements were also found in the new area of artificial intelligence, where programs needed to handle memory with greater flexibility than in scientific applications. Again, this led to the development of specialized programming notations.

Against this background, there were a number of calls for the development of ‘common’ or ‘universal’ languages. For example, following a conference in 1955, the German/Swiss *Gesellschaft für angewandte Mathematik und Mechanik (GAMM)* established a committee to define a common formula translation language. In 1957, members of this committee wrote to the American *Association for Computing Machinery (ACM)* proposing a conference with the aim of fixing on a common formula translation language [Bauer et al., 1957]. This led to a meeting in Zurich in 1957, attended by four delegates each from ACM and GAMM. The result of this meeting was a language proposal known officially as the International Algebraic Language (IAL) [Perlis and Samelson, 1958]. In the light of subsequent developments, this language is often referred to as Algol 58.

Following extensive discussion of this proposal, a further conference was held in Paris in January 1960, resulting in the publication of a report which defined a new language, Algol 60 [Naur et al., 1960]. Unlike other languages of the time, Algol 60 was not designed as part of a programming system for a particular type of computer, but was intended as a universal language for the expression of algorithms. There was considerable institutional support for this proposal: for example, even before the publication of the Algol 60 report the *Communications of the ACM* had started a “new editorial department . . . to publish algorithms consisting of ‘procedures’ and programs in the ALGOL language” [Wegstein, 1960]. Initially algorithms were published in Algol 58, but following the publication of the Algol 60 report, the required language was changed to Algol 60.

The details of Algol 60 will be considered in more detail in the next section. The definition of a single programming notation was not the only way to tackle the problem of diversity, however. An alternative approach was suggested by the UNCOL project. It had already been noted that "the scope of activity for SHARE was expanded with the advent of the IBM 709 and with the universal acceptance of Fortran as a language common to both the 704 and the 709" [SHARE, 1958a]. At a meeting in February, 1958, discussion took place on "ways to develop a universal language for the computing field" [SHARE, 1958b], and over the coming year a sub-committee of SHARE developed proposals to address this need.

The UNCOL project distinguished between machine-oriented languages and problem-oriented languages. Rather than defining a single problem-oriented language, like Algol, the idea was to define a single machine-oriented language which would be used to implement a variety of problem-oriented languages (POLs). This was seen as promising two benefits: firstly, problem-oriented languages would be better tailored to the needs of programmers, and could be expected to make the task of programming quicker and easier. Secondly, it was anticipated that the implementation of a new POL would require only a POL-to-UNCOL translator to be written, not a full compiler for every machine that the POL ran on. It would therefore be more economical to develop new POLs using the UNCOL approach [Steel, 1961]. It proved impossible at the time to develop a practical system based on these proposals, however, and Algol became seen as the most promising and fully developed proposal for a universal programming notation.

5.6 Algol 60 as a formal language

Subsequent chapters describe the influence of Algol 60 on the subsequent development of programming and programming notations in the 1960s. This influence is sometimes attributed to the way in which the language was defined rather than its practical success. Unlike most, if not all, of its predecessors, Algol 60 was consciously presented as a formal language; for example, in 1959 Puyen and Vauquis wrote of the emerging Algol definition in an manner very reminiscent of Carnap:

> Pour ce langage unique, il faudra, tout comme pour les systèmes de programmation, commencer par en définir les éléments: d'abord les symboles élémentaires et leurs divers rôles, ensuite les règles de formation d'agrétats de ces symboles pout obtenir des termes, enfin des règles de construction d'expressions á partir des termes ou á partir d'expressions plus simples. Il semblerait que l'expérience des auto-programmations

> actuelles puisse accélérer l'étude purement logique du langage en tant que système formel. [Puyen and Vauquois, 1960, p. 134]

By the end of the 1950s, the relationship between programming notations and formal languages was increasingly being commented upon, Woodger for example claiming that all order codes were formal languages [Woodger, 1960]. Woodger described a formal language as one defined by rules specifying its syntax and semantics; most order codes were not fully defined in this way, however, and relied for their semantic definition in particular on informal descriptions of the behaviour of a machine or interpreter. By contrast, the Algol 60 definition made a significant step forward in the explicit formalization of programming languages. This section describes the method of language description adopted for Algol, and the next section describes some of the ways in which logic influenced the features included in the language.

The alphabet

Tarski's first criterion for formal languages concerned the set of symbols used for constructing expressions in a language. Order codes and pseudo-codes usually adopted a subset of the characters provided by the available input devices as the alphabet of the language, and it was some time before the concept of a set of symbols became abstracted from the physical symbol set provided by the hardware.

The case of Fortran illustrates the difficulties experienced in moving to a more abstract definition. In the original programmer's manual, a "table of Fortran characters" was given [IBM, 1956, p. 49], comprising the 48 characters available on the IBM 704 together with the various ways they were coded on different media. There were two distinct '$-$' symbols: both could appear in data presented to a program, but only one of them could be used in program code, while the other was the only one to appear in program output. The '$' symbol, meanwhile, could only be used in a program within textual data that was to be output.

Sheridan later explicitly specified a Fortran "alphabet": he excluded one of the '$-$' signs and the '$' symbol, despite the fact that it could appear in the text of Fortran programs, but included a symbol '$\dashv$' which was "not a character explicitly indicated in any FORTRAN statement, serving solely as a statement endmark on the executive level" [Sheridan, 1959, p. 11], or in other words, not a symbol of the Fortran language at all. Both these definitions, then, failed to define exactly the set of characters that could appear in legal Fortran programs.

The Algol 58 group recognized that these difficulties would only be exacerbated in the case of a language intended to be used on many different machines:

> There are certain differences between the language used in publications and a language directly usable by a computer. Indeed, there are many differences between the sets of characters usable by various computers. Therefore, it was decided to focus attention on three different levels of language, namely a *Reference Language*, and *Publication Language*, and several *Hardware Representations*. [Perlis and Samelson, 1958, p. 9]

Of these, the reference language was the "defining language". The publication language had to ensure "univocal correspondence" with the reference language, but would allow for the use of, for example, subscript and superscript notation, and different national conventions for representing such things as the decimal point. Each implementation of Algol 58 would require a different hardware representation, depending on the capabilities of the target machine, but "[e]ach one of these must be accompanied by a special set of rules for transliterating from Publication language" [Perlis and Samelson, 1958, p. 10].

The basic symbols of the reference language comprised a rather heterogeneous set of individual characters, such as letters and digits, some digraphs, such as ':=', a range of mathematical and logical symbols, including a subscripted '$_{10}$', and a number of words and phrases, such as '*begin*' or '*go to*', all of which were considered to be indivisible, atomic symbols.

The picture that emerged from this account was rather a subtle one. By allowing different physical representations of the alphabet, the Algol 58 report made it apparent that, even in the reference language, the choice of physical symbols used was arbitrary. The alphabet of the language was therefore considered to be something more abstract than a set of characters. In 1959, commenting on the Algol 58 report, members of the Applied Programming Systems group at IBM put the point in the following way:

> The preliminary report on ALGOL defines the basic symbols of the language. A subset of these can be represented externally (now) only as words; e.g., **go to**, **do**, **if**, etc. Nevertheless, they stand for single characters which will have some internal representation. A good processor translates this external representation to internal. The dictionary used in making this translation should be flexible enough to allow arbitrary changing of the external representation of an internal symbol. We can therefore say that the processing of internal symbols can be independent of the external language. [Green et al., 1959]

In this respect, Algol differed from traditional accounts of formal languages which treated the expressions of a language as concrete sequences of characters [Tarski, 1933]. The variety of representations considered focused attention on the abstract structure of expressions rather than a particular representation, and this structure could be seen as defining the expression: "the syntax of the language will have to be the same for all levels" [Puyen and Vauquois, 1960, p. 134, my translation].

Object and metalanguage

One of the best-known features of the Algol 60 report is its use of a formal notation, now commonly known as *Backus-Naur form* (BNF), to specify the syntax of the language.

In a procedure reminiscent of Carnap's "syntactical Gothic symbols" [Carnap, 1937, p. 15], the Algol 58 report defined letters to represent syntactic categories, and used a mixture of informal definition and schematic templates to give syntactic definitions. For example, the set of digits is defined by:

> *Figures* ζ (arabic numerals 0, … , 9)

and the set of integers as follows:

> Strings consisting of figures ζ only represent the (*positive*) *integers* G (including 0) with the conventional meaning.

Based on this, numbers are defined as follows:

> Form: $\mathrm{N} \sim \mathrm{G.G}_{10}\pm\mathrm{G}$ where each G is an integer as defined above. [Perlis and Samelson, 1958, p. 11]

Backus, however, was not satisfied with this semi-formal approach, and in 1959 argued that if the language's goal of supporting a variety of implementations on different machines was to be met,

> There must exist a precise definition of those sequences of symbols which constitute legal IAL [i.e. Algol 58] programs … For every legal program there must be a precise definition of its 'meaning', the process or transformation which it describes, if any … Heretofore there has existed no formal description of a machine-independent language (other than that provided implicitly by a complete translating program). [Backus, 1959, p. 129]

Backus provided a formal metalanguage sufficient to define the syntax of Algol 58. The syntactic metalanguage was explained as follows:

> To begin with, we shall need *metalinguistic formulae*. Their interpretation is best explained by an example:

> $\langle$ab$\rangle$:≡ (*or* [*or* $\langle$ab$\rangle$(*or* $\langle$ab$\rangle\langle$d$\rangle$
>
> Sequences of characters enclosed in "$\langle\rangle$" represent metalinguistic variables whose values are strings of symbols. The marks ":≡" and "*or*" are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the strings involved. Thus the formula above gives a recursive rule for the formation of values of the variable $\langle$ab$\rangle$. It indicates that $\langle$ab$\rangle$ may have the value "(" or "[" or that given some legitimate value of $\langle$ab$\rangle$, another may be formed by following it with the character "(" or by following it with some value of the variable $\langle$d$\rangle$. [Backus, 1959, p. 129]

Backus' notation was used by Peter Naur in the Algol 60 report [Naur et al., 1960]. By the time the Algol 60 report was produced, this notation had been adapted slightly with "::=" replacing ":≡" and "|" replacing "*or*". In addition, it was made explicit that "the symbols used for distinguishing the metalinguistic variables (i.e. the sequences of characters appearing within the brackets $\langle\rangle$...) have been chosen to be words describing approximately the nature of the corresponding variable" [Naur et al., 1960, p. 301]; this was intended to provide an "immediate link between syntax and semantics" [Naur, 1981]. In the final notation, the definition of the syntax of integer constants appeared as follows:

> $\langle$digit$\rangle$ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
> $\langle$unsigned integer$\rangle$ ::= $\langle$digit$\rangle$ | $\langle$unsigned integer$\rangle\langle$digit$\rangle$
> $\langle$integer$\rangle$ ::= $\langle$unsigned integer$\rangle$ | $+\langle$unsigned integer$\rangle$ | $-\langle$unsigned integer$\rangle$

The background to the invention of BNF is rather unclear. Backus later claimed that he had been inspired by lectures given by Martin Davis on the work of Emil Post [Backus, 1980, Backus, 1981]. Davis, however, has stated that the only possible date for any such lectures was after the invention of BNF, and so they cannot have been the immediate source of inspiration [Davis, 1988]. There was some awareness within the computing community of Post's work, however: Rosenbloom's textbook of 1950 [Rosenbloom, 1950] contained a chapter on "The General Syntax of Language" which was largely an exposition of Post's results, and this textbook was cited in some more theoretical computing papers [Elgot, 1954, for example]. Other participants in the Algol development have suggested, however, that awareness of techniques for formalizing syntax and their advantages was rather widespread, at least among the European members of the committee [Bauer, 1981, Samelson, 1981].

Whatever the origins of the notation, however, the Algol 60 definition made explicit use of the logical distinction between object and metalanguage, thus drawing attention to the importance of giving unambiguous syntactic definitions of programming languages. The specific formal notation introduced for syntactic specification was highly influential and widely emulated, even being applied to existing languages such as Fortran [Rabinowitz, 1962].

Syntax

The syntax of Algol 60 was defined by means of a large number of BNF productions. In a few cases, the productions gave an informal characterization of a syntactic class, for example

$\langle$string$\rangle$ ::= $\langle$any sequence of basic symbols not containing ‘ or ’$\rangle$

where the ‘metalinguistic variable’ on the right hand side is only a description of the intended set of string and does not appear on the left-hand side of any production. To a much greater extent than any previous language, however, the Algol report formally defined which texts were legal Algol programs.

Various subformulas of the language were classified into three major categories, depending on their semantic role. *Expressions* were subformulas that denote values; like Fortran, Algol defined algebraic formulas as expressions which denote numbers, but included in the same category boolean expressions, denoting one of the values *true* and *false*, and designational expressions, which denoted program labels. *Statements* were subformulas describing the basic operations performed by the program, such as the assignment of a value to a variable, and also the structure of compound actions involving for example iteration. Lastly, *declarations* were subformulas which defined entities to be used elsewhere in the program, such as variables and subprograms.

The definitions of the formulas of the language were mutually recursive in various ways. For example, ‘compound statements’ were defined which grouped a sequence of statements within the ‘statement brackets’ **begin** and **end**:

$\langle$statement$\rangle$ ::= $\langle$basic statement$\rangle$ | $\langle$for statement$\rangle$ | $\langle$compound statement$\rangle$ | $\langle$block$\rangle$
$\langle$block$\rangle$ ::= $\langle$block head$\rangle$; $\langle$compound tail$\rangle$
$\langle$unlabelled compound$\rangle$ ::= **begin** $\langle$compound tail$\rangle$
$\langle$compound tail$\rangle$ ::= $\langle$statement$\rangle$ **end** | $\langle$statement$\rangle$; $\langle$compound tail$\rangle$
$\langle$compound statement$\rangle$ ::= $\langle$unlabelled compound$\rangle$ | $\langle$label$\rangle$: $\langle$compound statement$\rangle$

As the definition above shows, compound statements could include any other statements, including further compound statements to any level of nesting. Furthermore, declarations could include expressions, for example in defining the size of an array, and statements could include both expressions and declarations: for example, a ‘block head’ is a list of declarations that come into effect in a particular compound statement.

It was argued above that one innovation of Fortran, probably inspired by logical examples, was to give a recursive definition of the structure of arithmetical expressions. The Algol definition extended this recursive approach to all the syntactic categories of the language. Thus an Algol program could potentially have a complex, recursive structure quite different from the simple sequence of instructions that characterized programs in Fortran and other autocodes.

The identification of programs

One seemingly trivial property of the formal languages used in logic is that a ‘top-level’ category of expressions is identified. These are the expressions which can be used to perform the speech acts of interest: the predicate calculus, for example, is a language primarily designed to formalize assertions, and the category of well-formed formulas is defined accordingly.

In programming languages, the members of the top-level syntactic category are not declarative sentences but programs. A formally defined programming language, therefore, should therefore define in purely structural terms what constitutes a program. This approach took a while to evolve, however. The Algol 58 report gives the following explanation:

> Sequences of statements and declarations, when appropriately combined, are called programs. However, whereas complete and rigid rules for constructing translatable statements are described in the following, no such rules can be given in the case of programs. Consequently, the notion of program must be considered to be informal and intuitive, and the question whether a sequence of statements may be called a program should be decided on the basis of the operational meaning of the sequence. [Perlis and Samelson, 1958, p. 10]

In other words, the question of whether a given text was a program was considered to be a semantic, not a syntactic, matter. However, no attempt was made to spell out a sufficient set of semantic properties for qualification as a program, and by the time of the Algol 60 report the semantic elements of this definition had been dropped. In Algol 60,

> A program is a self-contained compound statement, i.e. a compound statement which is not contained within another compound statement and which makes no use of other compound statements not contained within it. [Naur et al., 1960, p. 300]

Semantics

When defining the syntax of Algol 58, Backus had written that “the formal treatment of the semantics of legal programs will be included in a subsequent paper” [Backus, 1959, p. 129]. No such paper appeared, however, and in the Algol 60 report the semantics of the language were defined informally.

The three syntactic categories, of expressions, statements and declarations were distinguished by their differing semantic roles. An arithmetic expression was defined to be “a rule for computing one real number by executing the indicated arithmetic operations on the actual numerical values of the constituents of the expression” [Perlis and Samelson, 1958, p. 13]; presumably other types of expressions, such as boolean expressions, were understood in the same way, though this was not stated explicitly. Statements were defined to be “[c]losed and self-contained rules of operation” [Perlis and Samelson, 1958, p. 13], and declarations “state certain facts about entities referred to within the program” [Perlis and Samelson, 1958, p. 17].

A very similar approach was adopted in the Algol 60 report, which stated that “[t]he purpose of the algorithmic language is to describe computational processes” [Naur et al., 1960, p. 300]. Many syntactic categories were accompanied by an description of the semantics of the formulas of that category, suggesting the intention to produce a compositional semantic account of the language. The description of the semantics were, however, informal and very similar in style to those given for Algol 58.

Although the informal semantics were largely stated in terms of the effect that a given formula would have on the execution of programs containing it, the precise nature of the virtual machine on which Algol 60 programs could be considered to run was not initially made explicit. This was probably a consequence both of the machine-independent aspirations of the language, and also its complexity, particularly in the area of the recursive definition of compound statements. The details of the ‘Algol machine’ were largely worked out in the course of writing compilers for the language, and for a number of years the complexity of Algol compilers was often remarked upon, and in some cases made the basis for criticism of the language.

5.7 The influence of logic on Algol

The previous section has described how Algol was presented as a formal language, using the meta-linguistic framework developed for formal logic. Logic also appears to have influenced the design of some of the features of Algol itself. For example, as well as arithmetic expressions, Algol defined a category of boolean expressions similar to those of propositional logic. The two truth values were defined, and a range of boolean operators defined. Algol therefore included an implementation of

Boolean algebra which allowed conditions to be defined more succinctly than in previous languages. Fortran, for example, originally only allowed conditions which compared the magnitude of a number with zero.

The designers of Algol also appear to have been influenced by the notation and concepts of the predicate calculus, and in particular by the ideas of substitution and of quantifiers as syntactic devices which bind variables. This section describes how these features were treated as Algol evolved.

‘Quantifiers’ in Algol 58

In Fortran, conditional execution of program statements was controlled by means of a conditional jump statement which differed little from the kind of statement available in machine codes. In Algol 58, by contrast, any statement could be preceded by an *if* statement, which made the execution of the statement depend on the truth-value of a given condition. For example, in

if (a $>$ 0) ; c := a $\uparrow$ 2 $\downarrow$ $\times$ b $\uparrow$ 2 $\downarrow$,

the assignment to c would only take place if the value of a was greater than zero. The *if* statement, and others such as the *for* statement which had a similar syntactic role, were called ‘quantifiers’. Presumably this terminology was chosen because, like the quantifiers of predicate logic, these statements are prefixed to other statements and affect their interpretation in some way. However, there is a significant syntactic difference between the two: whereas a quantified formula in logic is a single formula formed by prefixing a quantifier to a subformula, the example above is not treated as a single statement in Algol 58, but rather as two consecutive statements. This lead to a rather clumsy definition of its semantics: “If the value of [the condition] is *true*, the statement following the *if* statement will be executed. Otherwise, it will be bypassed and operation will be resumed with the next statement following” [Perlis and Samelson, 1958, p. 14].

By contrast, because of the recursive definition of the syntax of statements in Algol 60, the equivalent construct,

if $a > 0$ **then** $c := a \uparrow 2 \times b \uparrow 2$,

was a single statement whose effect when the condition is true is that of the substatement following **then**, and the description of its meaning does not refer to the subsequent statement in the program.

In Algol 60, however, the conditional part of the statement is no longer thought of as a prefix, and the terminology of 'quantifiers' is no longer used.

Substitution

The substitution of an expression for a variable as a means of generating new formulas from old was widely used in logic and the λ-calculus. Algol 58 defined a mechanism for substitution in the *do* statement, which had the following form:

$$do\ L_1, L_2\ (S_{\rightarrow} \rightarrow I, \ldots, S_{\rightarrow} \rightarrow I).$$

Here L_1 and L_2 are labels identifying a sequence of statements, and the parentheses define a number of substitutions whereby an identifier I would be replaced by an almost arbitrary string of symbols $S_{\rightarrow}$. The effect was defined to be the same as that of executing the resulting code in place of the *do* statement. In Algol 60, the *do* statement was removed, but the notion of textual substitution was preserved in the 'call by name' mechanism described below.

User-defined subroutines and parameter passing

Subroutines had been a prominent feature of machine code programing, and the methodological advantages of splitting a large program into a number of independent and reusable components were well understood. Integrating the subroutine concept with autocodes and formula translation languages proved not to be straightforward, however. The original version of Fortran, for example, allowed a predefined set of library routines to be called from a Fortran program, but there was no way within the language to define a new subroutine [IBM, 1956].

In 1958, both Fortran II [IBM, 1958] and Algol 58 introduced the possibility of defining subroutines in the high-level language. In Fortran II, it was possible to compile subroutines separately from a program, and combine the resulting machine-code files to create a complete program; this of course made it easy to reuse subroutines in more than one program. Algol 58, being an unimplemented language proposal, did not go into such detail, but it also included the ability to define functions and procedures within the language.

One issue in the design of a subroutine facility in a language is to decide how data is to be passed from the main or calling program to the subroutine. Fortran II did not specify the mechanism for this in detail, but assumed it was possible to pass both constant data and variables, including arrays, to subroutines, and that changes made by the subroutine to the data held in variables would be visible to the main program on return from the subroutine.

By contrast, Algol 58 defined two mechanisms for passing data to subroutines. In one-line function definitions, the formal parameters could only be identifiers, and the report implies that data would be assigned to these variables before the function was called. This mechanism, later known as 'call by value', fits the mathematical notion of a function where parameters are treated as input data which, from the point of view of the calling routine, cannot be changed by the function.

The second method of parameter passing used textual substitution, as defined independently by the *do* statement. The effect of calling a subroutine would be that of executing the statements making up the subprogram, after textually substituting the actual parameters for the formal parameters. The *do* statement was dropped in Algol 60, but substitution remained the default method for parameter substitution in subprogram calls, the technique being known as 'call by name'.

Thus Algol 60 defined two interpretation of the process of passing parameters to subroutines, call by value and call by name. It is a striking coincidence that these correspond closely to the two interpretations traditionally given to quantifiers in logic, with call by value resembling the traditional 'objectual' interpretation and call by name the 'substitutional' interpretation (re)introduced to the logical literature by Ruth Barcan Marcus [Barcan Marcus, 1962], but there appears to be no evidence that this work influenced the details of the parameter passing mechanisms of Algol.

Blocks and variable binding in Algol 60

A characteristic feature of quantifiers in logic is that they bind variables, in a sense making them inaccessible from outside the quantified formula. An analogous property of subroutine definitions was noted by Strachey and Wilkes, who in 1961 described the formal parameters of subroutines as "bound variables" and other variables occurring in the body of a subroutine as "free variables", commenting further that "the formal parameters in a function definition are strictly bound variables (that is, local to the definition)" [Strachey and Wilkes, 1961, p. 489]. The use of the term 'local' here makes a connection between variable binding and the Algol notion of 'block'.

The definition of statements in Algol stated that a sequence of statements enclosed within the special brackets **begin** and **end** was a *compound statement*. A *block* was a compound statement which additionally contained some declarations. These appeared at the start of the block, before the statements contained in the block. The Algol 60 report then stated:

> Any identifier occurring in a block may through a suitable definition be specified to be local to the block in question. This means (a) that the entity represented by this identifier inside the block has no existence outside it, and (b) that any entity represented by this identifier outside the block is completely inaccessible inside the block. [Naur et al., 1960, p. 9]

The following artificial example shows one block nested inside another.

```
OUTER: begin integer i, j ;
   j := 5 ;
   INNER: begin integer j, k ;
      j := 3 ;
      k := 2 × j ;
      i := j ;
   end block INNER
end block OUTER
```

Blocks implied a particular mechanism for the allocation of storage to variables. In the example above, storage will first be allocated for the variables i and j in the outer block. On entering the inner block, storage will be allocated for the variables j and k declared there. Crucially, the variable j in the inner block will be allocated a different storage location from the variable j in the outer block. At the end of each block, the storage allocated will be again deallocated and any values in the variables of that block will be lost.

The outer block has no access to the variables declared in the inner block, but the inner block can access the variables of the outer block, as the assignment to i indicates. Further, variables in the inner block are distinct from and ‘hide’ any variables with the same name in outer blocks: thus in the example above k is assigned the value 6. On completion of the inner block, i and j in the outer block have the values 3 and 5 respectively.

The local variables in a block, then, share some of the properties of bound variables in logic, in that they are inaccessible outside the construct in which they are defined, and they can, for example, be systematically renamed within such a construct without change of meaning, subject to the familiar restrictions on avoiding name clashes. Procedure declarations also bound the variables appearing as formal parameters, a process explained by invoking a “fictitious block” in which variables corresponding to the parameters were defined [Naur et al., 1960, p. 12]. As the quotation from Strachey and Wilkes indicates, the interpretation of blocks as variable binding mechanisms was made soon after the publication of the Algol report, and by 1980 it was apparently a commonplace, Mark Wells writing that the concept of block structuring “appeared first in ALGOL 58–60, although it is related of course to the idea of bound and free variables of logic” [Wells, 1980].

5.8 Lisp and recursive function theory

The last two sections have argued that both the design of the Algol language and the way in which it was presented were in many ways influenced by the existing example of logic, and that Algol was conceived of as a formal language in the sense in which that term was understood in logic. This was not the only direction in which programming notations developed, however. In the area of data processing, for example, it was believed that the use of anything resembling even elementary mathematical notation would be unacceptable to users, and languages designed for use in this application area such as FLOW-MATIC [Taylor, 1960] and its successor Cobol took their inspiration from natural rather than formal languages.

Algol did not even represent the only way in which the resources of logic could be applied to the task of designing programming languages. In 1960, shortly before the publication of the Algol 60 report, John McCarthy published the first description of the language Lisp [McCarthy, 1960]. As this section explains, Lisp just as much as Algol could be described as being ‘based on logic’, but with very different results.

Lisp was developed in response to the demands of programming artificial intelligence applications. Experience had indicated that a particular requirement of programs in this area was to be able to handle data structures which were of unpredictable size and which might vary in size throughout the time a program was executing. In 1956, Newell and Simon developed the ‘Logic Theorist’, a program intended to discovered proofs in propositional logic; they observed that “machine code, although suitable for communicating with the computer, is not at all suitable for human thinking or communication about complex systems” [Newell and Simon, 1956, p. 62]. They therefore developed a pseudo-code designed specifically to support the operations required in this application. Initially known as the “logic language” (and, incidentally, described as a “formal language”, though without being defined in a particularly formal manner) this evolved into a family of notations known collectively as ‘Information Processing Language’ (IPL) [Newell and Tonge, 1960].

The key data structures for this class of problem became known as *lists*: “IPL-V allows two kinds of expressions: *data list structures*, which contain the information to be processed, and *routines*, which define information processes” [Newell and Tonge, 1960, p. 205-6]. The system was conceived of as a virtual computer, the “IPL Computer”, which included memory suitable for storing list structures and a set of primitive processes, analogous to the basic orders on a conventional computer, defining basic operations on lists. Programs were then written by combining these primitive processes in a similar manner to conventional interpreted pseudo-codes.

Lisp itself combined the data structures used in IPL with the algebraic approach adopted by Fortran and was characterized by McCarthy as an "algebraic list-processing language" [McCarthy, 1981, p. 174]. Whereas IPL, like a machine code, only permitted sequences of the basic list operations to be written, McCarthy's approach would allowed complex expressions to be formed, analogous to the conventional algebraic expressions supported by Fortran.

Although Lisp later became described as a programming language, it was originally referred to as a "programming system ... based on a scheme for representing the partial recursive functions of a certain class of symbolic expressions" [McCarthy, 1960, p. 184]. McCarthy's initial presentation of Lisp in many ways echoed the details of the logical work on computability carried out in the 1930s, as the following summary indicates.

Firstly, McCarthy defined some mathematical notation for describing partial functions. As well as the conventional means of forming new function from old by using substitution and definition by recursion, a new notation for *conditional expression* was introduced, allowing 'definitions by cases' to be given by means of a single, formal expression. Church's λ-notation was used to represent functions, and a new construct 'label' was introduced to bind names in function definitions [McCarthy, 1960, p. 186].

McCarthy then defined the data objects that were intended to be the objects of computation, namely the class of *symbolic expressions*, or *S-expressions*. S-expressions were based on a set of *atoms*, represented by strings of upper-case letters, and were defined by the following two rules:

1. Atoms are S-expressions.

2. If e_1 and e_2 are S-expressions, so is $(e_1 \cdot e_2)$.

Some notational abbreviations were then introduced so that a more convenient list notation could be used. In particular, the list $(a_1, a_2, \ldots a_n)$ was defined to be the S-expression $(a_1 \cdot (a_2 \cdot (\ldots (a_n$ $\cdot$ NIL$) \ldots)))$, where NIL is a distinguished atom representing the empty list.

Having defined S-expressions and lists, the next step was to define some specific functions to manipulate them. McCarthy defined five elementary functions, namely *atom*, *eq*, *car*, *cdr* and *cons* which tested whether an S-expression was an atom and whether two atoms were equal, and allowed non-atomic S-expressions to be constructed and their two components retrieved. All other functions over S-expressions were defined from these basic functions using the methods specified earlier for the construction of recursive functions.

A class of *meta-expressions*, or *M-expressions*, was defined to represent functions over S-expressions. These were distinguished from S-expressions by using lower-case letters and different forms of punctuation. For example, the function 'ff' returning the first atomic symbol in an S-expressions could be defined by the following M-expression:

ff[x] = [atom[x] $\rightarrow$ x; T $\rightarrow$ ff[car[x]]]

By this point McCarthy had defined a class of data, the S-expressions, and a class of functions over these data elements, or S-functions, represented by M-expressions. These two notations were distinct: individual S-expressions could be represented by meta-notation in M-expressions. However, McCarthy's next step was to describe a method for representing M-expressions by S-expressions, "in order to be able to use S-functions for making certain computations with S-functions and for answering certain questions about S-functions" [McCarthy, 1960, p. 189].

Although McCarthy did not make this explicit, this was a form of Gödelization. Gödel had showed how expressions denoting functions over natural numbers could be encoded as natural numbers, and in exactly the same way, McCarthy encoded M-expressions, which represented functions over S-expressions, as S-expressions.

The purpose of this representation was to enable the definition of "a universal S-function *apply* which plays the theoretical role of a universal Turing machine and the practical role of an interpreter" [McCarthy, 1960, p. 184]. *apply* is universal in the following sense: "if f is an S-expression for an S-function f$'$ . . . then apply[f; args] and f$'$[arg1; . . . ; argn] are defined for the same values of arg1, . . . , argn, and are equal when defined" [McCarthy, 1960, p. 189]. *apply*, therefore, is capable of 'simulating' every other S-function, given an encoding of it as an S-expression, in the same way that the universal Turing machine can simulate the behaviour of any other machine, given a suitable encoding of its machine table.

The Lisp programming system itself was based on a program APPLY which implemented the universal function *apply*. 'Lisp programs' are S-expressions representing the functions to be computed, and these S-expressions are then evaluated by APPLY. Lisp can therefore fairly be described as a programming language which to a large extent is based on prior work in formal logic. Unlike Algol, however, Lisp is not presented as a formal language.

As noted above, Lisp is described by McCarthy as a programming 'system', not a language. The purpose of the system is to compute functions of S-expressions; these functions are denoted by M-expressions, but these must be translated into S-expressions before they can be submitted to the machine. In the context of the description given, it is impossible, and probably inappropriate, to single out either of these notations as 'the Lisp programming language'. Furthermore, the technical apparatus associated with the definition of a formal language is missing from McCarthy's paper. Despite their name, M-expressions are not a metalanguage in the sense of Tarski and Carnap, and only an informal presentation of the legal forms of M-expression is given.

This is not to say, of course, that a description of Lisp as a formal language could not easily be given, nor that McCarthy was unaware of the importance of formal languages; the discussion of alternative formalisms, such as "linear Lisp", at the end of the paper is evidence to the contrary. Rather, Algol and Lisp should be viewed as embodying two very different visions of how programming language development could be rooted in logic. Rather than seeing existing programming notations as examples of a new type of logical formalism, McCarthy emphasized the continuities with existing notations, showing how expressions directly representing recursive functions over a given class of data items could be executed by a machine.

5.9 Conclusion

This chapter has traced one path through the development of programming languages in the 1950s, and argued that the desire to automate parts of the programming, or coding, process led, through the development of systems for formula translation, to an understanding of programming notations themselves as being formal languages, a view made most explicit in the Algol 60 proposal. Section 5.6 argued that Algol was explicitly defined as a formal language, in the same way as logical notations, and Section 5.7 described the way in which specific features of Algol were influenced by logic.

This was not the only approach that was taken, however. Some languages, particularly those intend for data processing applications, such as FLOW-MATIC and Cobol, emphasized instead the extent to which programming notations could be made to resemble natural language, as a means of generating naturalness of expression and readability. A third approach, originating in the needs of artificial intelligence, used the resources of mathematical logic, but in a very different way from Algol.

However, it was the Algol proposals that caught people's attention, and largely inspired the developments in programming languages in the following decade. These developments are the subject of the following chapters.

Chapter 6

The Algol research programme

Compared to some other early programming languages, Algol 60 was not particularly successful in practical terms. Fortran and Cobol were very widely used in their respective application areas and many systems using these languages are still in operation, as the efforts made to update software before the year 2000 revealed. Lisp has also had a long history as a major implementation language in the field of artificial intelligence. By contrast, the take-up of Algol 60 was widely regarded as disappointing, even by advocates of the language.

At the same time, however, Algol 60 is widely considered to have been of great importance in the development of programming languages. In the preamble to the published proceedings of the 1978 ACM conference on the history of programming languages, for example, it was described as "an obvious landmark" and it was stated that "[m]ost theoretical, and much practical, language and compiler work since 1960 has been based on ALGOL 60" [Wexelblat, 1981, p. xviii].

The conjunction of these two facts presents something of a puzzle: how did a language which was a relative failure in practical terms later come to be regularly described as the most influential of early programming languages? This question is made explicit, but not answered, in a detailed history of the development of Algol 60 published by Bemer in 1969; in the introduction Bemer quoted a comment made by Ershov, that "the reading of this history . . . does not enable the beginner to understand why ALGOL, with a history that would seem more disappointing than triumphant, changed the face of current programming" [Bemer, 1969, p. 151].

This chapter suggests an answer to this question. What changed the face of programming, it will be argued, was not Algol 60 itself, but rather a coherent and comprehensive research programme within which the Algol 60 report had the status of a paradigmatic achievement, in Kuhn's terminology. This research programme led to significant developments in the design and theoretical understanding of programming languages, and also to proposals concerning the process of software development, the subject of the next chapter.

6.1 Algol 60 as a concrete paradigm

The creation, publication and subsequent development of Algol involved a large number of people in both Europe and the USA, and the language has a rich and well-documented "politico-social history" [Bemer, 1969]. The language acted as a catalyst for the formation of many new groups and initiatives, some of which are described in this section. There were earlier examples of social groups forming around particular computing technologies, notably the SHARE group formed by users of the IBM 704 computer as a vehicle to enable the sharing of code examples and working practices [Akera, 2001]. Algol in 1960 was not a fully developed technology, however, but a partially implemented language proposal, and the groups that formed round it had rather different purposes and trajectories.

Following the publication of the Algol 58 report, a number of computing centres in Europe began projects to create implementations of the language. In early 1959, representatives from these centres met in Copenhagen and agreed to start a newsletter, the "ALGOL-Bulletin", to enable continued collaboration and communication; the first bulletin was circulated in March 1959 [Naur, 1959]. As well as information about the development of "generators for translating ALGOL into machine language", the subject matter of the bulletins was expected to include discussion on aspects of the language that were found to be unclear in the published report, with a view to informing the subsequent description of the language.

The publication of the Algol 60 report was followed by a flurry of journal articles [Bemer, 1969, p. 219–234]. Three topics were particularly prominent in this literature. First was the issue of implementation: unlike Fortran, which had been made public in the form of a working system, the definition of Algol preceded any implementations, and it turned out that many new techniques were required in order to create Algol translators. A second topic was discussion of the language itself: there were many proposals for changes to the language, and the question of how such changes should be approved while maintaining the hoped-for universality of the language proved to be difficult to settle. Finally, the form of the language description, and in particular the use of a formal metalanguage to describe the syntax, gave rise to a lot of discussion [Floyd, 1964].

Algol turned out to be rather controversial, and never succeeded in gaining universal support, particularly in the United States where as early as 1961 there was a perception that the language had failed. SHARE withdrew support for the language, and work on an IBM translator stalled. Support was much stronger in Europe, where the first translator had been completed in 1960.

Despite these practical problems, the first institutional support for work on programming languages emerged at this time, supported by the International Federation for Information Processing (IFIP). In 1962 IFIP formed a technical committee on programming languages (TC-2), with the responsibility to look both at "general questions on formal languages, such as concepts, description and classification" and also the "study of specific programming languages" [Bemer, 1969, p. 197]. At the same time, a sub-committee, known as "Working group 2.1 (WG2.1)", was established to "assume responsibility for the development, specification and refinement of ALGOL" [Bemer, 1969, p. 198]. This strongly suggests that Algol had played a significant part in focusing interest on a more systematic approach to the study of programming languages.

At the same time, computing conferences and symposia began to take a greater interest in issues related to programming languages. In 1962, for example, the general conference organized by IFIP was described as being "[i]n virtually all respects . . . a programming-oriented conference" [Bemer, 1969, p. 202]. More specialized events soon followed: a symposium on "Symbolic Languages in Data Processing" was organized in 1962 by the International Computation Centre in Rome [International Computation Center, 1962], and in 1964 TC-2 organized a working conference on "Formal Language Description Languages" [Steel, 1966]. As the name of this later event suggests, attention was focused not only on programming languages themselves, but also on the metalinguistic techniques used to describe them. The catalytic role of Algol in this explosion of interest in programming languages was commented on by, among others, Edsgar Dijkstra, who wrote that "through its defects [Algol 60] has induced a great number of people to think about the aims of a 'Programming Language'" [Dijkstra, 1962b, p. 537].

By the middle of the 1960s, then, the study of programming languages, and in particular an approach treating programming notations as formal languages, was sufficiently well established to have attracted considerable institutional support and recognition. The Algol 60 report played a crucial role in this development as a "concrete paradigm", in Kuhn's sense of an exemplary achievement which is "sufficiently unprecedented to attract an enduring group of adherents" and "sufficiently open-ended to leave all sorts of problems for the redefined group of practitioners to re-solve" [Kuhn, 1962]. In recognition of its importance, the phrase 'the Algol paradigm' or 'research programme' will henceforth be used to refer to the tradition of work on programming languages inspired by the Algol 60 report.

6.2 Normal science in the Algol paradigm

In Kuhn's account, acquisition of a paradigm marks the maturing of a scientific field, and enables a transition to 'normal science' in which effort is focused on the solution of well-defined problems using standard techniques. A comprehensive and highly influential description of the problems and methods of normal science in the Algol research programme was given by John McCarthy, who in the early 1960s outlined a programme for the development of a mathematical theory, or science, of computation, stressing the relationship between this proposed theory and mathematical logic: "[i]t is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last" [McCarthy, 1963a, p. 69].

For McCarthy, the central problem that a theory of computation had to solve was a practical one: "we would like to be able to prove that given procedures solve given problems" [McCarthy, 1962, p. 21]. The ability to do this would radically change the nature of programming: "It should be possible almost to eliminate debugging ... Instead of debugging a program, one should prove that it meets its specification" [McCarthy, 1962, p. 21]. This goal was restated in 1965: "The prize to be won if we can develop a reasonable mathematical theory of computation is the elimination of debugging" [McCarthy, 1965, p. 219].

However, the existing theories of computability and finite automata were oriented towards the proof of general theoretical results, such as unsolvability theorems, and were unsuitable for application to more concrete and practical problems. McCarthy therefore listed some of the specific results and techniques that would be required, such as the ability to transform "an algorithm from a form in which it is easily seen to give the right answers to an equivalent form guaranteed to give the same answers, but which has other advantages such as speed" [McCarthy, 1961, p. 225].

A prerequisite for the development of a theory of the desired type was the existence of convenient notation for describing the "entities with which computer science deals", namely "problems, procedures, data spaces, programs representing procedures in particular programming languages, and computers" [McCarthy, 1962, p. 22]. For McCarthy, this notation should take the form of a universal programming language, thus ruling out theoretical notations on the one hand and machine-specific languages on the other; Algol was described as "being on the right track but mainly lack[ing] the ability to describe different kinds of data"[p. 225] [McCarthy, 1961]. As a preliminary, McCarthy described in detail a formalism similar to Lisp, based on the λ-calculus, which included ways of recursively defining functions computable on the basis of a given set of primitive functions, and methods for defining new data spaces in terms of old ones. This notation was not presented as a candidate for the universal language, but McCarthy believed that the design of programming languages could be systematized and improved by applying the results of a theory of computation to the task.

This chapter and the next consider in detail these two central aspects of McCarthy's programme. This chapter examines how the problems of describing programming languages and defining new ones were approached in the Algol research programme, and in the following chapter the project of replacing debugging with proof is examined.

6.3 The description of programming languages

As described in Chapter 1, a distinction between syntax and semantics had been described for logic by Tarski and Carnap in the 1930s. This distinction was drawn upon by Backus in his treatment of Algol 58 [Backus, 1958], and this approach was followed by the Algol 60 report. Syntax was explicitly distinguished from semantics, and a formalized metalanguage was used to specify the syntax, while the semantics were described in stilted English. The success of this approach provided a powerful motivation for exploring further the application of these metalogical notions to programming languages.

The link to the earlier work on the formal language of logic was made explicit by a number of people, such as Saul Gorn who, as part of an extended research project into the "theory of mechanical languages" [Gorn, 1962], produced a glossary of fundamental terms in the area [Gorn, 1961]. Gorn applied Morris's threefold division between pragmatics, semantics and syntax to the study of mechanical languages, but gave slightly revised definitions which reflected the fact that programming languages were intended to be processed by machine. Thus while Morris had characterized pragmatics as being concerned with "the relation of signs to interpreters" [Morris, 1938, p. 6], Gorn glossed this as follows: "We will 'interpret' the words *user* and *interpreter* to have a mechanical sense, i.e. to mean 'processor"' [Gorn, 1961, p. 337].

The application of Morris's scheme to programming languages provides a clear example of 'bridging', in the terminology of Pickering's scheme for conceptual innovation discussed earlier. However, the differences between programming languages and logic meant that the subsequent phase of 'transcription', the application of moves in the old domain to the new, was not as straightforward as might have been hoped. This section examines some of the issues that arose in this process.

The role of syntax

According to the traditional view, the role of syntax was to define the set of sentences comprising a language by means of purely formal or 'structural' rules; the semantics would then assign a meaning to each of the sentences defined by the syntax, which could therefore be understood as specifying a class of 'meaningful' sentences. The role of syntax and the nature of the relationship between syntax and semantics came under some discussion at the Rome symposium in 1962, however, and there were signs that this distinction could not be applied to programming languages without some refinement.

In the semantic account given by Tarski for first-order logic, it was impossible to have a syntactically correct sentence to which the semantics do not assign a meaning. It was assumed, for example by Gorn, that this property would hold also for programming languages. Christopher Strachey believed that this was the ideal situation, arguing that what he called the "integration" of syntax and semantics would make it "impossible to make a statement which is syntactically correct but semantically meaningless" [Strachey, 1962, p. 102]. However, he felt that this ideal could not be achieved for programming languages: "For nonsense program I mean one that makes the machine work indefinitely for example ... if you want a language powerful enough to ... specify all the programs that you want to run, then we must allow the possibility of a language being misused" [Strachey, 1962, p. 103]. Strachey here describes a situation where the syntax of a programming language permits 'nonsense', or non-terminating, programs to be written, but any attempt to modify the syntax to outlaw the offending programs would leave a language in which many desirable and meaningful programs could no longer be expressed. Although the extent to which this is seen as a problem depends on the contestable semantic judgement that non-terminating programs are to be treated as meaningless, it does at least point to a significant difference between the formal languages used in programming and logic, and suggests that the work of transcription might not be straightforward.

A more radical assault on the conventional metalinguistic scheme was made by van Wijngaarden and Dijkstra, who introduced a notion of "syntax-free languages", or more precisely, languages for which syntactical rules did not have their conventional function: "The main idea in constructing a general language, I think, is that the language should not be burdened by syntactical rules which define meaningful texts" [van Wijngaarden, 1962, p. 409].

Dijkstra later gave an account of the philosophy underlying this view, in which meaning is inextricable from the act of communication: "the reaction of my listener determines what my utterances mean" [Dijkstra, 1963, p. 33]. It follows, according to Dijkstra, that to know the meaning of an utterance is to be able to predict the reaction of a listener. This cannot be done precisely if the listener is a human being, and Dijkstra describes conversations between humans as devices which provide feedback enabling one to improve one's predictive ability. If the listener is a machine, however, as in the case of programming languages, its responses can be precisely predicted. The semantics of a programming language can then be specified by "the description of a machine that has as reaction to an arbitrary process description in this language the actual execution of the process" [Dijkstra, 1963, p. 34], the point being that in the case of programming languages we can tell from the text alone what process will be executed.

Given such a description, "syntax does not have a defining function" [Dijkstra, 1963, p. 34]. The semantic description will tell us what the machine will do in response to any program text presented to it, so syntactical rules are no longer needed to define a set of meaningful expressions. It may still be found useful to formulate such rules, but they will have only a practical value, to illustrate structural relationships that exist between program texts and the machine's responses, or to make it easier to formulate texts that elicit a particular response from the machine.

The meaning of programs

As with syntax, the differences between programming languages and conventional logic meant that there was considerable debate about how the meaning of a program could be characterized, and what form a semantic definition of a programming language could take.

An early idea was to extend Backus's notation to deal with more than just the syntax of a language. Edgar Irons described a technique for "syntax directed compilation" of an object language such as Algol into a target language, typically machine code, and pointed out that a compiler "also serves to *define* the object language in terms of the target language" [Irons, 1961, p. 51]. The technique adopted was to extend the syntactic production rules with clauses which described the meaning of the expressions defined by a rule by specifying the target language expressions they would be translated into. In general this would be expressed as a function of the meanings of the subexpressions of an expression. Later work based on this approach made the link with semantics quite explicit. Feldman, for example, described the target language in this scheme as a "semantic meta-language" and described his overall system as giving a "formal semantics" of the object language [Feldman, 1966, p. 3].

This work forms a distinctive approach to the problem of specifying programming language semantics, rooted in the practical problem of writing compilers for the large number of new high-level languages that were being developed. The hope was that a single 'compiler-compiler' could be written which would automatically generate a compiler for a new language from its formal specification. Two characteristic features of the approach derive from this orientation. Firstly, the meaning of a program was taken to be its translation into some other language, often an idealized machine code. A semantic definition of a language was therefore an explanation of how to carry out this translation in the general case. Secondly, the method drew on the existing work in syntax, structuring the semantic definition according the formal rules defining the syntax of a language. This strategy therefore guaranteed a compositional semantics like that developed for mathematical logic and also preserved the traditional role of syntax as defining the set of meaningful expressions.

As discussed above, however, the appropriateness of this account of syntax in the case of programming languages was questioned, and the view that semantics consisted primarily in translation also came under direct attack: at an ACM workshop on mechanical languages in 1963, McCarthy stated that "to describe semantics by means of a translation rule is an incorrect thing to do" [McCarthy, 1963b, p. 134], and similar views were expressed by Ken Iverson and Maurice Wilkes.

An alternative approach was related to the existing practice, described in the previous chapter, of specifying the meaning of machine codes and pseudo-codes by describing the real or virtual machine which interpreted the code. This technique was applied to the new programming languages of the 1960s, but with a new emphasis on giving a formal definition of the interpreting machine. McCarthy hinted at this approach, stating that one of the goals of a mathematical theory of computation was "[t]o represent computers as well as computations in a formalism that permits a treatment of the relationship between a computation and the computer that carries out the computation" [McCarthy, 1961, p. 225], and both van Wijngaarden and Dijkstra described abstract machines which were, according to Dijkstra, "suitable means for the formulization of the semantic definition of an algebraic language" [Dijkstra, 1962a, van Wijngaarden, 1962]. McCarthy had himself explained the meaning of Lisp programs by giving the definition of the 'apply' function which evaluated them [McCarthy, 1960]. Although 'apply' was defined in the same formalism used for the Lisp language, it was the description of a mechanical process for evaluating Lisp expressions, a 'Lisp machine' in effect.

The machine-based approach to semantics was further developed during the 1960s. In 1963, Gilmore described a "Lisp-like" language, stating that "[i]t is our belief that important purposes can be served by defining the semantics of a programming language by defining an abstract computer for which the programming language is the machine language" [Gilmore, 1963, p. 73], and a year later Elgot and Robinson described a class of "random-access stored-program machines", emphasizing that thereby "a basis is provided for endowing programming languages with semantics" [Elgot and Robinson, 1964, p. 365]. This general approach was adopted in a project to define the semantics of the programming language PL/I, about which it was stated "[t]he method used for the definition of a programming language is based on the definition of an abstract machine described by the set of its states and its state transition function" [Lucas and Walk, 1969, p. 105]. By the end of the decade, this general strategy was being referred to as the *operational* approach to programming language semantics [Lucas, 1972, Wegner, 1972].

However, in 1962 McCarthy had proposed a more abstract approach in which the details of the computation performed dropped out of the semantic account, leaving just the relationship between the initial data presented to the program and the results in produced. In general terms, he wrote, "[t]he meaning of a program is defined by its effect on the state vector ... In the case of ALGOL we should have a function $\xi' = algol(\pi, \xi)$ which gives the value ξ' of the state vector after the ALGOL program π has stopped" [McCarthy, 1962, p. 27]. This approach was exemplified in a later paper for a small subset of Algol, where it was explained that the state vector included "the value currently assigned to each variable and also the statement number about to be executed" [McCarthy, 1964, p. 3]. In this paper McCarthy also asserted that his approach to semantics "corresponds to the notions of Tarski, *et al.*, that are current in mathematical logic" [McCarthy, 1964, p. 6].

It is interesting to note how this account of semantics dealt with the non-terminating programs that Strachey wanted to describe as being meaningless. In the case of non-termination there is no final state, so the semantic function is undefined for such programs. For some, this was an objection to McCarthy's semantic account: because it omitted any account of the actions performed by programs, it could not distinguish between, for example, two non-terminating programs which were nevertheless performing very different computations [McCarthy, 1964, p. 10].

In giving this account, McCarthy appears to have been trying to align programming languages with established recursive function theory, much as he had done in the definition of Lisp. The syntactic form and machine-based interpretation of programs in languages like Fortran and Algol made them appear quite different from traditional notations such as the λ-calculus, but at the semantic level McCarthy suggested they were in fact similar, being just new ways of defining recursive functions of their input data. McCarthy also suggested a strategy for dealing with Algol-like languages, whereby a program would be translated into a single expression defining the function computed by the program [McCarthy, 1964].

This strategy was developed in greater detail by Peter Landin, who described a form of "Applicative Expression" (AE) based on the λ-calculus, together with an abstract machine which would evaluate AEs [Landin, 1964]. This was soon followed by an explicit proposal for a programming language based on AEs [Landin, 1966]. The semantics of this language were given an operational definition by describing a machine which would execute AEs. For Algol 60, however, Landin adopted McCarthy's proposal, arguing that Algol programs could be translated into semantically equivalent AEs; in fact, in order to deal with imperative features of Algol, such as assignment, an extended form of 'Imperative AEs' were used, with a suitably extended abstract machine [Landin, 1965a, Landin, 1965b].

Landin's work therefore combined two approaches to semantics: the meaning of an Algol program was to be given by translating it into the language of AEs, but the resulting AE program was to be understood in the traditional way by describing a machine to interpret AEs. Christopher Strachey proposed to go one step further, doing away with the need for an abstract machine in explaining the semantics of a language and describing "even the imperative parts of a programming language in terms of applicative expressions" [Strachey, 1964, p. 201]. This required some deviation from the techniques used in conventional logic, however. For example, the meaning of an expression in the predicate calculus is built up in a strictly bottom-up way from the meanings of its subexpressions. In an assignment statement, however, variables are interpreted differently depending on whether they are on the 'left', in which case they denote assignable locations in the store, or the 'right', in which case they denote storable values. This distinction had been made explicit in connection with the programming language CPL [Barron et al., 1963], and in order to get a compositional semantics for programming languages, Strachey found it necessary to make use of this idea, describing how a subexpression could have an “L-value” or an “R-value” depending on its context in a larger expression.

Strachey developed his ideas further in the following years into a comprehensive programme that became known as *denotational* semantics [Tennent, 1976]. Although it had its roots in the idea of translating Algol programs into AEs, Strachey developed a distinctive view of the role of syntax which helped differentiate denotational semantics from earlier accounts of semantics as translation. Strachey felt that an emphasis on the syntactic definitions of existing programming languages obscured important, and as yet ill-understood, semantic ideas, and rather than describing a fixed language he preferred to discuss “basic” or “fundamental” concepts [Strachey, 1967]. Following McCarthy, he viewed details of syntax as essentially irrelevant, and worked instead with ‘abstract syntax’, chosen to articulate clearly what he considered to be the important semantic concepts. This concern that the syntactic structure of a programming language clearly reflect its semantics was shared by others in the field of programming language design, as the following sections describe.

To summarize, then, the application of the metalogical distinction between syntax and semantics to programming languages resulted in the early 1960s in the development of at least three distinct approaches to the problem of giving the semantics of programming languages. The translation-based account was from the beginning associated with the practical task of writing compilers, but despite occasional proposals “to define languages by their compilers” [Garwick, 1964], became less frequently referred to as an approach to semantics, compared with the operational and denotational techniques, in part because of the “inscrutable” nature of the semantic description that a compiler embodied [Rochester and Goldfinger, 1964].

Pragmatics

Compared with syntax and semantics, the semiotic notion of pragmatics was rather underdeveloped in mathematical logic, and did not establish a very clear identity in the field of programming languages. One contributory factor in this may have been uncertainty as to whether it concerned the relationship between programming languages and human users, as implied by Morris’s original definition, or mechanical processors, as in Gorn’s reformulation. This ambiguity is reflected in the papers presented at an ACM conference in 1965 on “Programming Languages and Pragmatics”: some sessions were devoted to topics in the machine processing of languages, such as ‘translation’ and ‘interpretive assembly’, while others considered the requirements for programming languages that were to be used in specific application areas, such as real-time applications and information retrieval [ACM, 1966]. In an overview paper, Heinz Zemanek listed four specific areas as being relevant to the pragmatics of programming languages—compilers, hardware and operating systems, intended application areas and human users—but commented that “we are very far from any formal treatment” [Zemanek, 1966].

One widely discussed aspect of pragmatics concerned the question of whether different programming languages were required for different application areas. Distinctions were commonly drawn, for example between so-called ‘scientific’ languages such as Fortran and Algol, and languages such as Cobol which provided facilities for data description that were felt to be necessary for commercial applications. The perception of such differences had implications for the design of new programming languages, as the next section discusses.

6.4 Different philosophies of programming language design

Investigation into new programming language concepts, and the development of new languages, continued throughout the 1960s. Conflicts between the basic assumptions made by different groups led to a more general debate on the principles that should guide language design.

One approach is illustrated by the ‘New Programming Language’ (NPL), later to be known as PL/I, whose development was started by IBM in 1963. The aims of the language emphasized convenience and usability: it was intended to be used by programmers in a very wide range of application areas, to be usable by both novice and expert programmers, and “to take a simple approach which would permit a natural description of programs so that few errors would be introduced during the transcription from the program formulation into NPL” [Radin and Rogoway, 1965, p. 9].

NPL was intended to be a language that would be easy to program in. It should not be necessary to know every detail of the language before making productive use of it, and the language specification should not place obstacles in the way of programmers. Two specific design criteria suggested a route to this goal. First, “*Anything goes.* If a particular combination of symbols has a reasonably sensible meaning, that meaning will be made official” [Radin and Rogoway, 1965, p. 9]. Secondly, ‘modularity’ would allow programmers to remain in ignorance of aspects of the language which were not appropriate for their current task or level of expertise: “one cannot get a compile error by leaving something out” [Radin and Rogoway, 1965, p. 10]. The overall impression gained is that NPL was intended in many respects to emulate natural, not formal, languages: the programmer, or ‘speaker’, was allowed a great range and flexibility of expression, and it was assumed that the interpreter had a considerable degree of sophistication enabling it to make out the intended meaning.

The desire to produce a language usable in different application areas and the concern showed for the experience of programmers working with the language suggest that the NPL project was primarily influenced by pragmatic concerns, as defined above. In contrast, in accordance with McCarthy’s overall goal of eliminating debugging, the Algol research programme placed more emphasis on the avoidance of errors in programming, and evolved an approach to language design that was more rooted in semantic issues.

In 1965 Dijkstra wrote a paper which considered how a “lone programmer” could have confidence that the results of a program were in fact those intended [Dijkstra, 1965]. In some cases, the results of a program can be directly checked, but in other cases this is not feasible. Dijkstra considered the example of a program which tests the primality of large integers. If such a program generates purported factors for a large integer, this result can be checked by direct calculation. If on the other hand the program reports that there are no factors, the programmer has to decide how much credence to put in this report. In its general form, this is an epistemological question. Very many programs function as potential sources of knowledge, whether concerning the primality of integers or the size of a gas bill, and Dijkstra asked in what circumstances we can place confidence in the knowledge generated by such programs, and what we can do to increase this degree of confidence.

Dijkstra proposed an answer to this question which was based on an analogy between mathematical proofs and computer programs. He considered mathematical proof to be the best available model of how to gain confidence in the correctness of assertions, and planned to apply the lessons learnt from proof to the task of programming:

> In spite of all its deficiencies, mathematical reasoning presents an outstanding model of how to grasp extremely complicated structures with a brain of limited capacity. And it seems worthwhile to investigate to what extent these proven methods can be transplanted to the art of computer usage. [Dijkstra, 1965, p. 5]

This analogy was to be exploited by adopting what a strategy of ‘divide and rule’, whereby a complex artefact is treated as an assemblage of simpler ones.

> The analogy between proof construction and program construction is, again, striking. In both cases the available starting points are given (axioms and existing theory versus primitives and available library programs); in both cases the goal is given (the theorem to be proved versus the desired performance); in both cases the complexity is tackled by division into parts (lemmas versus subprograms and procedures). [Dijkstra, 1965, p. 5]

It seems clear from this that Dijkstra thought of the activity of programming as largely text-based: a programmer should examine the source code of a program, and arrive at a conviction of what the program is doing in much the same way as a mathematician reads a proof and comes to accept the truth of the result that is proved. The application of these ideas to program development are considered in the next chapter, but this approach also had a consequence for programming language design: designers should identify the characteristics of programming languages that help or hinder the efficacy of programs as documents which engender conviction, and design languages which gave programmers the best chance of writing correct programs.

This issue came to prominence in the debate within WG2.1 about a successor language to Algol 60. The dominant tendency within the group was towards a form of generalization known as 'orthogonality', where "all possible combinations of two or more independent concepts were allowed" [van der Poel, 1986]. Given even a small number of basic concepts, this approach would quickly lead to a large and complex language; the alternative was "only to insert those possibilities in the language as were seen fit for some purpose" [van der Poel, 1986]. The orthogonal approach formed the basis for the language Algol 68, while an alternative view was put forward in a paper by Hoare and Wirth, who described the characteristics of a language that would be suitable for Dijkstra's purposes:

> The perspicuity of programs is believed to be a property of equal benefit to their readers and ultimately to their writers . . . [A language's] power and flexibility should derive from unifying simplicity, rather than from proliferation of poorly integrated features and facilities. As a consequence, for each purpose there will be exactly one obviously appropriate facility, so that there is minimal scope for erroneous choice and misapplication of facilities, whether due to misunderstanding, inadvertence or inexperience. [Wirth and Hoare, 1966, p. 414]

The next two sections describe how this principle was applied in practice in the two areas of control and data structures. This work formed a basis for a general approach to programming known as *structured programming*, which had a great influence on programming and program language design, as discussed in the following chapter.

6.5 Logic and the design of control structures

The debate over program language design became particularly heated over the design of control structures, and in particular a controversy about the role of the jumps in programming. It had been always been recognized that the conditional execution of code and the repeated execution of a block of code were essential coding patterns, but in machine code these were implemented using jump instructions to navigate around a program. In Fortran and later languages, unconditional jumps were provided by means of a special statement, the so-called 'goto' statement.

In addition to a primitive jump statement, programming languages gradually introduced specialized statements, or control structures, which encapsulated these common patterns of control. For example, Fortran's DO statement provided a basic iteration facility [IBM, 1956] and the conditional expressions of Lisp made the conditional execution of code explicit [McCarthy, 1960]. Algol 60 included both a **for** statement for writing loops, and an **if** statement for conditional execution [Naur et al., 1960].

The supposed benefits of specialized notation for describing the flow of control had been commented on by a number of people. For example, Hamblin had written that "'Control transfer' instructions represent the biggest problem in this kind of notation ... there are some hopes that control transfer may be unnecessary in other cases if a sufficiently flexible system of conditional instructions can be found" [Hamblin, 1957, p. 138-9], and McCarthy wrote of traditional notations for recursive functions that "controlling the flow in this way is less natural than using conditional expressions which control the flow directly" [McCarthy, 1961, p. 237]. It was Dijkstra, however, who brought the issue to prominence and linked it with the more general issue of the readability of programs:

> I have done various programming experiments and compared the ALGOL text with the text I got in modified versions of ALGOL 60 in which the goto statement was abolished and the for statement ... was replaced by a primitive repetition clause. The latter versions were more difficult to make: we are so familiar with the jump order that it requires some effort to forget it! In all cases tried, however, the program without the goto statements turned out to be shorter and more lucid. [Dijkstra, 1965]

The explanation that Dijkstra gives for the increase in clarity has specifically to do with the termination properties of programs. Failure to terminate is usually caused by faulty iterations: if iteration is consistently expressed throughout a program by a single control structure, rather than by a number of unstructured jumps, it is plausible that it will be easier to tell from an examination of the program text whether or not it terminates.

Dijkstra's comments about the benefits of programming without jumps raised the question of whether it was in fact always possible to eliminate goto statements. In 1966 Böhm and Jacopini published a technical result on normal forms in an artificial flowchart language which was widely interpreted as showing that it is possible to write a program for any algorithm using only conditional and iterative control structures, and hence as showing the dispensability of the goto statement [Böhm and Jacopini, 1966].

The theoretical possibility of doing without goto statements did not directly address Dijkstra's requirement for lucidity, however. As he later pointed out, there is no a priori reason to suppose that a goto-less program produced by means of Böhm and Jacopini's method will be any more comprehensible or convincing than one using goto statements. Dijkstra expanded on his argument in a famous letter to the editor of the Communications of the ACM, which appeared under the strap-line "Go To Statement Considered Harmful":

> More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). [Dijkstra, 1968b]

Dijkstra gave two distinct arguments for this recommendation. The first was related to a comment made by Wirth and Hoare claiming that "[t]he notational structure of programs expressed in the language should correspond closely with the dynamic structure of the processes they describe" [Wirth and Hoare, 1966]. Dijkstra made the same point as follows: "we should do ... our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible" [Dijkstra, 1968b, p. 147].

This argument is related to the requirement that a compositional style of semantic explanation should be given for programming languages, with the corollary that languages should not contain statements which do not permit of such an explanation. Consider, for example, a control structure that defines an iteration. The meaning of this statement, in the sense of the computational processes it gives rise to when a program is running, is determined by the syntax of that single statement: in order to understand what controls the number of times the iteration will take place, for example, it is not necessary to look at any statements before or after the for statement itself. This can be contrasted with a goto statement, which specifies a label which may be anywhere in the program:

without knowing the location of the labelled statement, it is impossible to give the operational meaning of the goto statement. The use of goto statements, therefore, suggests that the meaning of a program can only be given globally.

Dijkstra's second argument for the abolition of jumps was related to interest in the relationship between programs and proof that was generated by, for example, papers by Floyd and Hoare [Floyd, 1967, Hoare, 1968]. This work is considered in detail in the following chapter; in 1968 Dijkstra made relatively informal use of it, arguing that to understand a program, we must be able to interpret the values of the variables in it. However, "we can interpret the value of a variable only with respect to the progress of the program" [Dijkstra, 1968b, p. 147]. In order to do this, Dijkstra claimed that it was necessary to be able to specify "textual indices to the dynamic process", or in other words properties of the program text which will enable us to characterize the point that the dynamic process has reached when the program is running.

For example, in a program without jumps, the statements could be numbered, and the state of the dynamic program could be given by simply stating the number of the currently executing statement. The numbering scheme needs to be made more complex to cope with iteration constructs and subroutine calls, and Dijkstra demonstrated how this could be achieved. If a program includes goto statements, however, the graph of potential paths through the program becomes arbitrarily complex, and there is no possibility of identifying the state of an executing program by any number of textual indices. This argument was later summarized as follows:

> Investigating how assertions about the possible computations (evolving in time) can be made on account of the static program text, I have concluded that adherence to rigid structuring disciplines is essential ... sequencing should be controlled by alternative, conditional and repetitive clauses and procedure calls, rather than by statements transferring control to labelled points. [Dijkstra, 1969b, p. 85–86]

Eliminating the goto statement did not, however, settle the question of what control structures should be provided by a language. On the basis of practical experience and on Böhm and Jacopini's theoretical result, it was accepted that programmers needed the means to express the sequencing of statements, and conditional and repeated execution of statements, but there were many ways to do this, with a wide variety of control statements being defined in contemporary languages.

Initially, rather informal arguments were given for and against various constructs. For example, Dijkstra referred to the **for** statement in Algol 60 as being "pompous and over-elaborate" [Dijkstra, 1965], and Wirth and Hoare wrote that:

> The purpose of iterative statements is to enable the programmer to specify iterations in a simple and perspicuous manner, and to protect himself from the unexpected effects of some subtle or careless error. ... It is notorious that the ALGOL 60 **for** statement fails to satisfy any of these requirements, and therefore a drastic simplification has been made. [Wirth and Hoare, 1966, p. 415]

For Dijkstra, a more principled way to evaluate different control structures was to consider their proof-related properties. The choice of control structures should be subordinated to the need to produce convincing arguments for the correctness of programs containing them, and not made by appealing to their 'power' or the 'usefulness' they had for programmers. In 1969, Dijkstra made this point by stating that he had "focused [his] attention ... on the questions 'for what program structures can we give correctness proofs without undue labour, even if the programs get large?'" [Dijkstra, 1969b, p. 85]. The same point is made more explicit in his monograph on structured programming, published in 1972 but widely circulated before that: "Why do I propose to adhere to this sequencing discipline? ... For all three types of decomposition—and this seems to me a great help—we know the appropriate pattern of reasoning" [Dijkstra, 1972, p. 20]. Hoare made a similar point: "there is a theory that a high-level language feature should also simplify the task of proving the correctness of programs expressed in the language" [Hoare, 1972a, p. 336].

In order to reason about conditional statements, Dijkstra appealed to what he called "enumerative reasoning". In order to prove a given property, it might be necessary to consider a number of different cases which together exhaust all the possibilities. If the desired result follows from each case individually, then it is proved by appeal to a theorem of the form $(b \supset p \wedge \neg b \supset p) \supset p$. For example, suppose it is desired to show that execution of the following statements will preserve the truth of the relation $0 \leq r < dd$ [Dijkstra, 1972, p. 7]:

```
dd := dd / 2 ;
if dd ≤ r do r := r – dd
```

There are two possible cases, depending on whether or not the relation $dd \leq r$ holds after execution of the first statement, and Dijkstra showed that in each case execution of the **if** statement will leave the desired relationship true.

In order to reason about iterations, Dijkstra made use of mathematical induction in conjunction with a very simple form of statement, the **while** statement, which repeats a statement so long as a specified condition remains true. Suppose that a program must examine a sequence of values d_i, where $d_1 = D$ and $d_i = f(d_{i-1})$, and locate the first value d_k which satisfies a given property $prop$.

By a proof based on induction over the number of times the statement $d := f(d)$ has been executed, Dijkstra showed that the following statements will achieve the required effect [Dijkstra, 1972, p. 8]:

```
d := D ;
while non $prop(d)$ do d := f(d)
```

Dijkstra also provided a proof that the loop terminates after the kth iteration.

Dijkstra also gave an example of how enumerative and inductive reasoning can be used to prove the correctness of a small program [Dijkstra, 1972, pp. 12–14]. The proof was presented in a style typical of informal mathematical reasoning, consisting of explanatory text in English interspersed with formally expressed propositions and pieces of program text, but nevertheless Dijkstra professed himself “infuriated” by the length and complexity of the proofs obtained in demonstrating the correctness of even extremely small program fragments, and recognized the impracticality of such proofs being carried out as a normal part of software development.

The solution proposed to this problem was the idea that certain program structures should acquire the status of theorems. For example, Dijkstra suggested that the conclusion proved about the loop above could be considered as the “Linear Search Theorem”, and claimed that

> when a programmer considers a construction like [the loop above] as obviously correct, he can do so because he is familiar with the construction. I prefer to regard his behaviour as an unconscious appeal to a theorem he *knows*, although perhaps he has never bothered to formulate it; and once in his life he has convinced himself of its truth, although he has probably forgotten which way he did it [Dijkstra, 1972, p. 10].

In summary, then, this section has shown how specific proposals for the design of control structures in programming languages were strongly influenced by the logical orientation of the Algol research programme. A general desire to align the syntactic and semantic structure of program texts prompted a move away from goto statements to more specialized control structures, the specific form of which was motivated by a desire to make the correctness programs accessible to logical reasoning, even if programmers made only informal and second-hand use of the logical results.

6.6 Logic and data structures

As well as the flow of control, the manner in which programming languages enabled the description and manipulation of data was extensively investigated in the 1960s. To a greater extent than with control structures, it was believed that the requirements for data representation differed in different application areas. In the period around 1960, at least three distinct approaches can be identified.

Firstly, the so-called scientific languages such as Fortran and Algol 60 were acknowledged to be rather weak in their support for different kinds of data. With their emphasis on numerical calculation, scientific languages distinguished between integer and floating-point numbers, but textual data was poorly supported. The only widely available data structure was the array, which could store a fixed-size collection of numbers: arrays enabled mathematical structures such as vectors and matrices to be modelled.

However, the development of non-numerical programs in the field of artificial intelligence had revealed a class of applications which manipulated symbolic rather than numerical data, and in which the amount of data that a program would need to handle, and the structure of that data, could not be predicted in advance. Languages were therefore developed which enabled programmers to define data structures of arbitrary size and complexity: the best known were the list structures present in languages such as IPL [Newell and Tonge, 1960] and Lisp [McCarthy, 1960].

Finally, commercial data processing applications were accustomed to handling files of data, consisting of a set of records, each of which was in turn made up of a number of fields or data items which could be in a variety of textual or numeric formats. Languages designed for these applications, such as Cobol, provided the means to give a detailed description of the structure of the files that would be manipulated by a program.

Attempts were made, both in practice and in theory, to unify these different approaches. Practical proposals included a number of ad hoc suggestions to incorporate the features from one area into a language of a different type; for example, proposals were made to add support for strings and lists to Algol 60 [Green et al., 1959], and some proposals for new languages, such as NPL [Radin and Rogoway, 1965], attempted to include features from all areas.

The general problem was summarized by Douglas Ross in the context of a computer-aided design system which needed to be able to model the properties of a wide range of objects: "before anything else we must provide for a completely general method of storing and manipulating arbitrarily complex information from any source, and a powerful language facility for describing data forms and the desired manipulations of data" [Ross and Rodriguez, 1963, p. 306]. Ross's solution envisaged "problems as being composed of interconnected n-component elements of a general type" [Ross, 1961, p. 147]. An n-component element provided a way of grouping together an arbitrary number of symbolic and numeric data items to provide “a single unit of information about a problem, which specifies in each of its components one attribute or property of the element” [Ross and Rodriguez, 1963, p. 306]. Elements were allowed to refer to each other, and the resulting network of linked elements was described by Ross as a *plex*. Ross viewed plexes as simpler in general than list structures, in which each element could hold only two data elements, and as providing a way of uniting the manipulation of both symbolic and numerical data.

The integration of these ideas into Algol-like languages was considered by Wirth and Hoare. Their proposal for a successor to Algol 60 introduced the concept of a *record*, which could be used to “represent inside the computer some discrete physical or conceptual object to be examined or manipulated by the program” [Wirth and Hoare, 1966, p. 416]: this was essentially the same as Ross’s n-component element, but the change of name made an explicit link to the terminology employed in the field of data processing. With each record was defined an associated value known as a *reference* which uniquely identified that record: by including in one record references to others, complex data structures equivalent to Ross’s plexes could be constructed.

Unlike arrays, records could be created as required when a program was running, thus providing programmers with the ability to create data structures whose size could vary dynamically according to the requirements of a program. Thus this single mechanism provided a way of unifying the three distinct approaches to data structuring found in programming languages.

Records with a common structure intended to capture “the natural classification of objects under some generic term, for example: *person*, *town* or *quadrilateral*” [Wirth and Hoare, 1966, p. 417] were considered to be grouped into equivalence classes, known as *record classes*. By defining the class of each record explicitly in a program, in the same way as numeric variables were declared to hold integers or floating-point numbers, it was proposed that the compiler could detect programming errors that might be caused by mistaking the structure of a record indicated by a particular reference.

These proposals about records and record classes were incorporated into the Pascal language [Wirth, 1971b]. Pascal defined a number of *scalar* types, representing atomic data values such as numbers, characters, user-defined symbols and references (called ‘pointers’ in Pascal), and a number of *structured types* by means of which data values could be combined to create more complex, structured data values. Structured types were defined by *type expressions*, and the form of these type expressions, and by extension the data structures defined by them, were strongly influenced by theoretical work on data structures that had been carried out in parallel with the practical language developments.

An early theoretical proposal was made by McCarthy. As we have seen, McCarthy viewed computation as the definition of computable functions over given classes of data, but he pointed out that the theory of data was not as well developed as that of computable functions: "Procedures operate on members of certain data spaces and produce members of other data spaces ... A number of operations are known for constructing new data spaces from simpler ones, but there is as yet no general theory of representable data spaces comparable to the theory of computable functions" [McCarthy, 1962, p. 21]. McCarthy sketched the beginnings of such a theory by identifying data spaces with sets, arguing that data spaces could be defined by recursive equations which used the primitive operations of Cartesian product, direct union and the formation of the power set. For example, the equation $S = A \oplus S \times S$ could be interpreted as defining "the set of S-expressions on the alphabet A" [McCarthy, 1961, p. 231-2].

This approach was further developed by Hoare, who developed a theory proposing that types in programming languages could be understood as denoting sets of data values. Given a number of basic types, defined by enumeration, further types could be defined by means of a range of operators, in the manner proposed by McCarthy. The link with set theory was made explicit: "The types in which we are interested are those already familiar to mathematicians: namely, Cartesian Products, Discriminated Unions, Sets, Functions, Sequences and Recursive Structures" [Hoare, 1972b, p. 93]. Some of these operations corresponded to existing data structures: records, for example, were understood to be elements of the Cartesian product of the types of their components. Others, such as the set of subsets of a given set, corresponded to purely mathematical operators which had not been implemented in practical languages.

Pascal drew upon this theoretical work by defining a number of structured types many of which were based upon the set theoretical operators described by McCarthy and Hoare [Wirth, 1971b, p. 37]. For example, record types could be defined which corresponded to the Cartesian product of the types of the record components, and a "powerset structure" defined a type whose elements were sets of elements of a given type.

It proved impossible, or impractical, for Pascal to implement fully Hoare's theoretical account. For example, the powerset structure was limited so that only powersets of certain small scalar types could be formed. Also, Pascal did not provide a data structure corresponding directly to the discriminated union operation of set theory. Instead, record types could include a number of *variants* identified by tags; by this means, a record type could represent either a Cartesian product or a discriminated union.

Another area of difficulty was presented by the pointers, or references, used to construct linked networks of records. In Pascal, pointers were themselves data values, represented as values of pointer types. A pointer could be stored in a record, say, allowing structures analogous to Ross's plexes to be constructed. However, there is no obvious set-theoretic analogue to pointers: in set theory, there is no intrinsic connection between one data value and another, and no obvious way of interpreting the computer-based notion of one data value 'pointing to' another. Instead, McCarthy and Hoare had defined plex-like structures by means of recursive type definitions.

A recursive type definition would model a relationship between data values a and b by including a copy of b in a. By contrast, a Pascal representation would include a pointer to b in a. However, the semantics of these two representations are different, as can be seen by considering the situation where the value of b is updated. With pointers, this update is immediately visible to a, as it contains only a pointer to the now updated value of b. With a recursive type, however, a now contains an out of date copy of b, and clearly it may take significant programming effort to make sure that this copy is kept consistent with the changing value of b.

Despite these shortcomings and inconsistencies, however, Pascal's type system was a product of a collaboration between theory and practice similar to the case of control structures. In both cases, the design of certain central aspects of programming languages was profoundly influenced by theoretical considerations drawn from logic and set theory.

6.7 Modelling data for information retrieval

At the beginning of the 1960s, the development of the two areas of scientific and data processing programming systems were carried out largely independently. Nevertheless, logic and set theory played a significant role in the development of information retrieval systems as well as in programming languages oriented towards scientific applications.

The assumption underlying the design of the so-called scientific languages was that programs were written to perform particular computations, to generate a set of results from a given set of input data. This assumption lay behind McCarthy's proposal, described above, to model the semantics of programs by their input-output functions. Data structures such as variables and arrays were defined as required in the program itself, in the blocks containing the code that manipulated that data, and it was assumed that it would be a relatively straight-forward task for a program to read in the data that it required for a particular run.

A different model was assumed in the field of data processing applications: "an information retrieval system consists of a file structure to index and hold information ... [and] a body of programs for performing the various processing tasks" [Colilla and Sams, 1962, p. 11]. Thus many programs might be written to process the same set of data, which therefore had to be understood to exist independently of any particular program. This differing philosophy of data had implications for programming language design: in Cobol, for example, the description of the structure of the external files and other data used by a program was placed in a 'data division' which was separate from the 'procedure division' containing executable statements [Sammet, 1962].

Cobol encapsulated a model where data was thought of as being grouped into a number of *files*, each consisting of a set of *records*. *Elementary items* in records were 'atomic' pieces of data, and records could be given a hierarchical structure in which subrecords at various levels could be defined to enable a number of elementary items to be handled as a single unit. However, from the beginning of the 1960s proposals were also made to treat data in a more abstract way. For example, Lionello Lombardi objected to the separation of "descriptive" and "executable" statements, and proposed a "boolean algebra of files" which would enable these two aspects of information retrieval systems to be better integrated [Lombardi, 1960].

A more comprehensive attempt along the same lines was the proposal for an "information algebra", published in 1962 by the Language Structure Group of the CODASYL Development Committee [Bosak et al., 1962]. This group had been established in 1959 to study and make recommendations on the languages to be used for data processing applications. The information algebra aspired to provide a theoretical foundation for information systems, based on "the concepts of Modern Algebra and Point Set Theory", which would guide the development of future programming languages. The report enumerated various shortcomings in existing languages, and hoped to address them largely by defining a declarative rather than a procedural framework.

The report gave the following general definition of how an information system should deal with those aspects of the world relevant to a given application:

> An information system deals with objects and events in the real world that are of interest. These real objects and events, called "entities", are represented in the system by data. The data processing system contains information from which the desired outputs can be extracted through processing. Information about a particular entity is in

> the form of “values” which describe quantitatively or qualitatively a set of attributes or “properties” that have significance in the system. [Bosak et al., 1962, p. 190]

The designer of an information system for a particular application should begin by defining all the relevant properties of the entities involved in the application. With each property was associated a *value set*. For example, the value set associated with a property representing the salary of employees in a company might be the set of natural numbers. The *property space* of the application was defined to be the Cartesian product of the value sets defined for the various properties.

Entities were represented by points in this property space, or in other words by a ordered set (or *tuple*) of values. This implies that each entity was associated with exactly one value from the value set for each property. Special null values were defined to deal with properties that might be irrelevant for given entities.

The information algebra itself provided a way of defining groups of data points and operations on these groups. This was intended to provide a means to define the data processing functions required by a typical application.

It is interesting to compare the approach taken by the information algebra to that of researchers interested in the application of mathematics and logic in programming. The same areas of mathematics were used to model data in both areas, namely set theory and abstract algebra, but in one respect the approach taken by researchers in the Algol research programme differed from a purely algebraic approach.

In the formal presentation of the information algebra, it was stated that “[t]he Algebra is built on three undefined concepts: entity, property and value” [Bosak et al., 1962, p, 191]. However, the concept of an entity played little part in the subsequent formal definition of the algebra, referring instead to the external objects being modelled. A methodological principle in constructing a property space for a given application was that each entity should be represented by a unique point in property space.

A model constructed using the information algebra, therefore, contained no direct representation of the entities being modelled. An entity was represented solely as the collection of the values of its properties at a given time. A consequence of this type of representation is that, over time, a given entity would be represented by many different points in property space, as the values associated with its various properties changed. The model itself provided no representation of the fact that these are properties of the *same* entity at different times.

This approach requires that care be taken in the selection of the set of properties to be used in a given information system to avoid the situation where more than one entity is represented by the same point in property space. For example, a payroll system that used only the properties of ‘employee name’ and ‘salary’ would be unable to handle the situation where two employees had the same name and salary. An information system designer must ensure that different entities will always have different values for some subset of the properties in use. This was usually achieved by defining properties such as ‘employee payroll number’ which by would be guaranteed to be distinct for each entity modelled by the system.

As discussed in the previous section, Ross had proposed a general technique for modelling data about an arbitrary collection of entities using “plexes” of “n-component elements”. Rather than being based on an abstract data space, however, Ross’s proposal was based on an abstract view of a computer’s memory, in which data about distinct entities which happened to share the same properties could easily coexist at different locations in the store. They would be distinguishable by the fact that the values referring, or pointing, to them would be distinct. Thus, in contrast with the information algebra, Ross’s proposal made use of data, in the form of references, that was not part of the application being modelled.

The distinction between these two approaches was maintained later in the 1960s as more concrete proposals for database systems emerged. It was increasingly felt that even a file-based model like Cobol’s did not recognize the centrality of data in many application areas. Particularly in large commercial organizations, the data that was held could be a significant economic asset and have a lifetime much longer than that of the programs which manipulate it. The same data set might need to be processed by many different programs, for different purposes. An alternative perspective was required, one which made data independent of programs, allowing it to take on a life of its own. As Charles Bachman, a database researcher, put it in 1973, the move from files to database could be viewed as a kind of Copernican revolution, challenging the perceived centrality of programs and proposing a new model of computation in which programs were viewed as satellites of a central data repository [Bachman, 1973].

A number of different database models were put forward, but they shared a number of characteristics. Firstly, like the information algebra and Cobol, databases were based on a model consisting of files and records, with each record consisting of a number of primitive data items. However, unlike the Information Algebra, which defined a single undifferentiated property space to cover all the data in an application, and Cobol, which assumed a collection of independently defined files, a database is conceived of as a structured collection of heterogeneous files whose interrelationships are specified by means of a single overarching database *schema*.

Secondly, as databases are assumed to be independent of particular programs, programs using databases cannot in general access data items based on their location in computer memory. Entities can only be identified in a database by looking at the actual data values stored for each. For this to be possible, records must have some unique attribute distinguishing them from all other records in the file. Entities often do not have this property: for example, we cannot assume that the individuals in a group of people will be uniquely identified by their names. To get round this problem, the records in a database typically include an attribute or attributes, known as a *key*, whose value is guaranteed to be unique within the file.

Finally, a database schema will normally record information about significant relationships between the entities. This is done by associating in some way the key values for related entities. The key for one entity might occur in the record for another, or a particular record might store only the key values of related entities. For example, one field in a record for an employee may be the key attribute for a file of departments within a company. The value of this field in an employee record would enable a particular department record to be located, thus modelling the fact that the employee works in a particular department.

A prominent proposal for database design at the end of the 1960s was for 'network' databases, based on earlier work by Charles Bachman and formalized by the CODASYL committee which also maintained the definition of the Cobol languages [CODASYL Data Base Task Group, 1969]. Network databases are based on two primitive concepts, the file and the 'set'. Sets are the vehicles for representing relationships within a network database: all the records which are related in a particular way to a given record, such as the set of employees that work in a particular department, constitute a set of records, explicitly linked together by pointers into a chain of records.

Bachman described the way a programmer worked with network databases as "navigation" [Bachman, 1973]; a similar metaphor was used by others, such as Jay Earley who referred to "access paths" through data structures [Earley, 1971]. Programs accessing the data have various notions of a 'current location', and by means of commands embedded in a programming languages such as Cobol can update the current location and thereby move from data item to data item within the database. For example, suppose a program has to process all the employees working in a particular department. The relevant records are physically linked in the database as part of a set; the program will record the current position within this set, and a programming operation is provided to move to the next item in the set. This can be repeated until every record in the set has been processed. Thus programs access such databases ‘from the inside’, as it were, a record at a time.

An alternative model, the ‘relational’ model was introduced by Ted Codd in a paper published in 1970 [Codd, 1970]. There were two main differences between the relational and network model. Firstly, the relational data model was based on a single structuring concept, the relation. This is basically the set-theoretical concept of a relation, or Cartesian product of sets, used here as a formal model of records. No special data structure, such as CODASYL sets, was used to model relationships between entities. Rather, relationships were modelled by pairing up the key fields of the related entities, and storing these pairs in a further relation. So whereas network databases had two primitive concepts, files and sets, corresponding to the informal notions of entity and relationship, relation databases have one primitive concept, the relation, which models both. One advantage claimed for this was that it kept the logical structure of the data was independent of its physical representation, thus making updates and modifications to the storage strategy easier, because they would not necessarily imply changes to the application programs using the database.

Secondly, data manipulation in the relational model does not proceed by means of record-at-a-time navigation through the database. Rather, a number of high-level operations on relations are provided, the most significant of which is perhaps the ‘join’, an operation whereby two relations can be combined into one. These operations are defined to work on whole relations, rather than on individual records, and return new relations as their results. These resulting relations are logical, rather than being physically stored in the database, but as data structures they are identical to the relations defined in the database schema. This means that they can be used as the input to further operations, thus enabling data manipulation to be defined by means of the repeated application of a small set of powerful operations.

6.8 Conclusions

This chapter has discussed the use of logic and algebra in computer science in the 1960s in the areas of programming language design and the development of theoretical models for databases. It was argued that the publication of the Algol 60 report catalysed the formation of a coherent research programme aimed at using logic as a foundation for understanding and developing programming languages. This proposes an answer to Ershov's implicit demand, quoted at the beginning of the chapter, for an explanation of the influence of Algol 60 given its relative lack of practical success.

Even in the context of this research programme, however, the use of logic was not simply a case of applying theoretical results and drawing straightforward consequences for programming languages. Even fundamental concepts, such as the roles of syntax and semantics, could be contested and significant amounts of work were required to establish how logic could be applied to programming languages. Nevertheless, significant results were obtained: in particular, an influential tradition of program language design was formulated, based on the idea that the syntax of a language should as far as possible reflect its semantics in a clear and unambiguous way.

Logic and algebra were also used to investigate the properties of data structures, both in the so-called scientific languages and the field of data processing applications. In this area, the concepts of syntax, semantics and proof turned out to be less useful than the tools of set theory and abstract algebra, but nevertheless there are structural similarities between the issues raised in both areas.

Perhaps the most striking of these can be described as a move away from a step-by-step approach to computation to one which made greater use of high-level operators described in terms of their overall effect. In the area of control structures, this can be seen in the introduction of control structures, embodying frequently occurring patterns of computation, in place of the goto statement. In the database world, the relational model with its set of general algebraic operations would in the coming years supersede the network model with its reliance on navigation from record to record in the database, a procedure in itself reminiscent of a jump operation.

However, this transition was incompletely carried out in programming languages themselves. For example, Pascal incorporated a 'network' model of data in the form of records and pointers, and despite Hoare's attempt to provide a theoretical model for this in the form of recursive definition of data types, later programming languages have preserved a form of programming which relies on 'navigation' between data items. These developments will be considered in Chapter 8; the next chapter considers a different aspect of the Algol research programme, namely the introduction of logical ideas into the process of program development.

Chapter 7

The logic of correctness in software engineering

This chapter describes the impact of the Algol research programme on the practice of software development. The aim was to improve the quality of software development and to ensure that systems that met their users' expectations and were completed economically and on schedule. This concern came to prominence in the mid-1960s in response to a perceived 'software crisis' widely discussed at a NATO conference which brought the term 'software engineering' to prominence [Naur and Randell, 1969].

The approach of the Algol paradigm to these concerns was twofold. Firstly, a particular notion of "correctness" was defined for software, namely the existence of a particular type of consistency between a program and its specification. This was claimed to be the most important property of a software system, and was characterized in such a way as to make plausible the possibility of applying a type of proof to software development.

Secondly, practical programming techniques were proposed which would increase the likelihood of correct programs being developed. Some of these techniques drew upon the work on desirable properties of programming languages that was described in the previous chapter, but from the beginning of the 1970s this work was increasingly presented in a way that made it accessible to the software industry and not solely to researchers.

7.1 Checking computations

The use of large-scale automatic computers raised the question of how the correctness of the results produced could be guaranteed. The meaning of ‘correctness’ in this context changed as computing and programming technology evolved and brought different issues to prominence. With the earliest machines, whose reliable functioning could not be taken for granted, the problem was taken to be that of checking the computation performed by the machine, to see if a correct answer had been produced. So, for example, Aiken and Hopper wrote of Mark I that “of paramount importance in the design of a sequence control tape, are the checks on the computation” [Aiken and Hopper, 1946, p. 525].

Aiken and Hopper identified three distinct sources of error. Firstly, errors could be made in the mathematical formulation of the problem being solved. These errors did not differ in principle from those that had been made in the context of manual computation, however, and familiar mathematical checks could be applied to detect them. Secondly, errors could be introduced by malfunctioning hardware. These raised issues of reliability, but were relatively easily dealt with by electrical engineering methods for ensuring the reliability of circuits.

A third and novel source of error was introduced by the processes involved in transferring the mathematical solution of a problem onto the computer. Aiken and Hopper classified these as human error: “two major sources of human error, incorrect switch settings and incorrect plugging, are perhaps the most serious of all” [Aiken and Hopper, 1946, p. 525], as in the absence of a feasible mathematical check on the final results of a computation these errors could easily go undetected. The precautions required to avoid such errors were described as “meticulous precision of the operator’s part and careful checking of all manual operations” [Aiken and Hopper, 1946, p. 525].

With the advent of stored-program machines, the manual operations involved in setting up the machine to perform a particular calculation were no longer required, and the importance of the design of the sequence of operations to be carried out was made explicit. At the conference held in Cambridge in 1949, J. C. P. Miller discussed the errors arising from “[p]rogramming and coding the [mathematical] solution for the machine” [Miller, 1949].

‘Programming’ here refers to the design of a suitable algorithm to perform a calculation, and programming errors correspond to the mathematical errors identified by Aiken and Hopper; it is noteworthy that they did not identify the coding process, whereby the algorithm was translated into machine code instructions, as a separate source of errors. Errors in coding were only gradually recognized to be a significant problem: a typical early comment was that of Miller, who wrote that such errors, along with hardware faults, could be “expected, in time, to become infrequent” [Miller, 1949]. Two years later, however, Maurice Wilkes and his colleagues reported that “such mistakes are much more difficult to avoid than might be expected” [Wilkes et al., 1951, p. 38], and similar comments were made by others [Brooker et al., 1952, for example].

Programming and coding errors are design errors: unlike hardware errors, they are not caused by mechanical or electronic failure, and so cannot be removed by increasing the reliability of any device. A variety of techniques for preventing such errors were considered, including the inspection of programs to reveal common mistakes, the inclusion of additional code to check the results being obtained, and the automation of the programming process itself. The use of library subroutines was also found to reduce errors: as these contain reusable code performing various common tasks, they were used frequently and were found more likely to be free from errors than new code.

During this early period much emphasis was placed on the avoidance of error by uncovering mistakes before a program was executed, and there is little mention of testing, understood as the repeated execution of a program with particular data values for which the expected results are known, as a technique for identifying errors. The scarcity and expense of machine time appears to have ruled out such an approach: Aiken is reported as having had “very little patience for an error-infested trial session” [Bloch, 1999, p. 97], and Wilkes refers to the amount of machine time that could be lost running erroneous programs.

Initially, then, the notion of correctness was applied generally to the computations carried out by an automatic computer. Correct computations were taken to be those which produced correct results, although it was not always easy to tell which these were: “It cannot therefore be assumed that if a program apparently operates correctly it is giving correct results, and careful numerical checks must always be applied” [Wilkes et al., 1951, p. 41]. The coding process gradually emerged as a significant source of errors, and the desirability of reducing the number of coding errors was recognized. Correctness was understood to be a product of many factors, however, including the algorithm used, the coding of it for a particular machine, any library routines utilized, and the physical machine itself: as Miller put it, “*[a]ll* stages must be fully checked if a satisfactory solution is to be obtained” [Miller, 1949].

7.2 Debugging and testing

In the 1950s, increased experience of programming, and in particular of the problems of developing larger software systems, led to the development of new techniques and approaches to the problem of program correctness. Increasing machine reliability led to more emphasis being placed on mistakes "arising because the orders or data presented to the machine are not those required to obtain the results sought" [Gill, 1951]. Initial optimism had given way to a belief that such errors were not "a temporary evil, due to lack of experience", and "some attention has, therefore, been given to the problem of dealing with mistakes after the programme has be tried and found to fail" [Gill, 1951].

Gill did not feel that detecting that an error had occurred was the significant issue: at least for programs performing computations, "[i]f its presence is not immediately apparent, it will be detected by the arithmetical checks which must be incorporated in every calculation" [Gill, 1951]. Rather, the immediate issue was to locate and correct the error, a process that came to be known as *debugging*. Standard test tapes were used to diagnose faults in the operation of the machine itself, and a variety of techniques for diagnosing program errors were introduced, such as push-button operation in which the program was run manually, one instruction at a time. Many of these techniques had severe disadvantages: push-button operation, for example, was exceedingly slow and expensive, and prevented a machine from being used for other work.

A promising direction of research was to investigate approaches which used the machine itself to assist in debugging. As with automatic coding, programmers were quick to realize that the repetitive aspects of their work could be carried out by machine. Gill described different "checking routines" in use on the EDSAC, the most useful of which interpreted a program line by line and printed out the function letters of the orders being executed, thus allowing the programmer to trace the history of the program execution. Similar approaches were adopted at other computer installations. For example, Ira Diehm described how the SEAC computer of the National Bureau of Standards in the USA was used to analyze coding errors by means of techniques such as the use of "breakpoints" at which program execution could be interrupted, and an "automonitor" checking routine, among others [Diehm, 1952].

The development of large systems raised further unanticipated problems, and in 1956 Herbert Benington described the lessons drawn from experience gained on the SAGE air defence system [Benington, 1956]. Benington described a process for the "production of a large-program system" which surrounded the coding activity with a preliminary stage of preparing specifications, and a subsequent stage of testing the program produced against its specifications. The detection of errors was no longer felt to be the unproblematic activity portrayed by Gill: the tests to be carried out were themselves planned and specified, and a clear distinction was drawn between the detection of errors in a testing activity, and their location and correction in debugging. Debugging itself was, as on other systems, partially automated by means of a system program known as the "checker".

At the same time, Benington felt that there were limitations in the use of testing as a method of ensuring correctness. It was, he wrote, "debatable whether a program ... can ever be thoroughly tested—that is, whether [it] can be shown to satisfy its specifications under all operating conditions ... one must accept the fact that testing will be sampling only ... many sad experiences have shown that the program-testing effort is seldom adequate" [Benington, 1956]. Like debugging, the testing process could be automated by programs which performed "test instrumentation" using simulated live inputs.

It was widely expected that the development of automatic programming and the use of pseudocodes would reduce the frequency of programming errors. Diehm believed that "[t]he trend toward automatic performance of the clerical parts of the coding process should reduce the number of coding errors" [Diehm, 1952, p. 19] and Gill wrote that "[i]t is to be hoped ... that many of the tiresome blunders that occur in present-day programmes will be avoided when programmes can be written in a language in which the programmer feels more at home" [Gill, 1953, p. 291]. This expectation was soon found to be ill-founded, however: one early programmer recalled that "[t]he only place where we made a mistake ... was believing that when FORTRAN came along we wouldn't make any mistakes in coding", and cited a survey which indicated that Fortran programs typically had to be compiled up to 50 times before they were correct [Bemer, 1984].

The increasing use of pseudocodes raised the question of how best to carry out debugging: initially, debugging efforts were directed towards the machine code generated from the pseudocode, but it was recognized that it would be more convenient if a program written in a symbolic pseudocode could be debugged by examining that code rather than the machine code. Katz discussed the issues raised by debugging programs written in pseudocode; after discussing various tools for performing "symbolic debugging" of pseudocode programs, however, he restated the belief that a much lower frequency of programming errors would obtain when "compiling techniques are sufficiently improved and our pseudo-codes are completely natural and simple to use" [Katz, 1957, p. 21]. By the end of the decade, Gill was suggesting a two-level approach to the debugging problem, where "experts" would want to debug machine code and "novices" would require debugging information presented in terms of the "hypothetical machine which is visualized by the user" [Gill, 1959].

With increasing reliability of hardware, then, the coding, or programming, activity became widely recognized to be the most significant source of errors in computer programs, despite repeated expressions of optimism that improvements in the design of codes would remove this problem. Correctness became understood more as a property of the program than of the overall computation, and testing and debugging were identified as the key techniques for identifying and locating errors in programs.

7.3 Proof and program development

One of the goals of the Algol research programme was to utilize the resources of logic to increase the confidence that it was possible to have in the correctness of a program. As McCarthy put it, "[i]nstead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program" [McCarthy, 1962, p. 22]. McCarthy thus envisaged using the computer to automate the routine or mechanical parts of the proof-checking process, as was already being done in the areas of testing and debugging.

The limitations of testing that had been pointed out by Benington were further articulated, by Dijkstra in particular, and developed into a more general argument for the necessity of stronger techniques to demonstrate the correctness of programs. The fact that a computer passes an acceptance test, according to Dijkstra "only says that in these specific test programs the machine has worked correctly" [Dijkstra, 1962b, p. 537], and does not permit us to conclude that the machine will work correctly when presented with other programs. In its structure, this is similar to an argument against induction, the point being that from a finite set of observations nothing can be inferred about future or unobserved events. Similarly, when a program fails a test, this can be taken as evidence of an error in the program, but passing a test only demonstrates its correctness in one particular case. As Dijkstra later put it epigrammatically, "Program testing can be used to show the presence of bugs, but never to show their absence!" [Dijkstra, 1969b, p. 85].

In 1966, Peter Naur described it as "deplorable . . . that the regular use of proof procedures . . . is unknown to the large majority of programmers" [Naur, 1966, p. 310]. In Naur's view, an algorithm performed a transformation on some data, and the role of a proof was "to relate the transformation defined by an algorithm to a description in some other terms, usually a description of the static properties of the result of the transformation" [Naur, 1966, p. 311]. To incorporate proof into the program development process, Naur proposed a methodology which would start with a static description of the properties of the algorithm, then "construct an algorithm ... using examples and intuition to guide us" [Naur, 1966, p. 311], and finally prove that the algorithm had the required properties.

One difficulty in the way of constructing proofs of programs arises from the semantic difference between the assertions that describe the transformation carried out by an algorithm and the imperative program code that describes how the algorithm will perform the transformation. Naur described this as the problem "of relating a static description of a result to a dynamic description of a way to obtain the result" [Naur, 1966, p. 312]. Observing that one way to follow the execution of an algorithm was to look at "snapshots" describing the data held in the variables at different times, he proposed a technique of "General Snapshots" which would not describe individual data values, but rather define predicates which the program data should always satisfy at specific points in the execution of the program. By appealing to properties of the program code, it could be established that the general snapshots would always be true when a running program reached them. The snapshots would therefore give a static description, in propositional form, of the transformation carried out by the program. This could then be related to the specification to demonstrate the correctness of the program.

A similar approach was proposed by Robert Floyd, who put forward "the notion of an interpretation of a program: that is, an association of a proposition with each connection in the flow of control through a program, where the proposition is asserted to hold whenever that connection is taken" [Floyd, 1967, p. 19]. The correctness of a program could then be obtained "by an induction on the number of commands executed" [Floyd, 1967, p. 19], enabling proofs of propositions of the form "[i]f the initial values of the program variables satisfy the relation R_1, the final values on completion will satisfy the relation R_2" [Floyd, 1967, p. 19]. Floyd made the issue of semantics explicit, referring to his technique as a way of "assigning meanings to programs".

The technique of using propositions to make assertions about properties of program executions had, by 1966, quite a long history. For example, Richard Bloch described his practice in programming Mark I in 1944 as follows: "I carefully annotated the code using mathematical symbolism pertinent to the problem being solved. I marked the quantities being transferred as well as the location of partial results in order to assist in tracing the flow of the program, and I maintained a dynamic series of ‘snapshots’ of the storage register contents as the program progressed” [Bloch, 1999, p. 94].

Similarly, von Neumann and Goldstine, as part of their technique of flow diagrams for program development, had observed that “[i]t may be true, that whenever [control] actually reaches a certain point in the flow diagram, one or more bound variables will necessarily possess certain specified values, or possess certain properties, or satisfy certain relations with each other” [Goldstine and von Neumann, 1947, p. 92]. Such properties were recorded in special *assertion boxes* at various points in a flow diagram and used to argue for the correctness of the algorithm depicted. In a paper delivered in 1949, Turing adopted von Neumann’s notation and made the connection with program correctness explicit, writing “[h]ow can one check a routine in the sense of making sure that it is right? ... the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows” [Turing, 1949]. However, despite these suggestions, it was only in the context provided by the Algol research programme in the mid-1960s that the use of assertions was systematically researched and serious attempts made to apply it to programming practice.

The use of general snapshots, or assertions, to prove the correctness of programs required that arguments could be made about the effect of individual statements on program data. Naur made such arguments informally: “suppose that $A[i] > A[r]$ is true. Then clearly $A[i]$ is the greatest among the elements up to $A[i]$. Changing r to i as is done in the assignment then makes it again true to say that $A[r]$ is the greatest” [Naur, 1966, p. 324]. To formalize such arguments required a definition of program statements in terms of their effect on the assertions preceding and following them. Such definitions could act as axioms and rules of inference in constructing proofs of programs.

A first attempt to give such rules was made by Floyd, who defined for each type of statement a condition “guarantee[ing] that whenever a command is reached by way of a connection whose associated proposition is true, it will left (if at all) by a connection whose associated proposition will be true at that time” [Floyd, 1967, p. 19]. Floyd only applied the technique to a flowchart representation of programs, however, but his ideas were soon applied by Hoare to a textual programming language.

Hoare made a strong and explicit connection between programming and logic, stating that “all the properties of a program ... can, in principle, be found out from the text of the program itself by means of deductive reasoning”, and his paper aimed to “elucidate the axioms and rules of inference which underlie our reasoning about computer programs" [Hoare, 1969, p, 576].

Hoare's logic was based on statements of the form $P\{Q\}R$, which were to be interpreted as meaning "[i]f the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion" [Hoare, 1969, p. 577]. Axioms in the system defined the properties of individual statements, and inference rules defined the properties of control structures. The axiom for the assignment statement was given as

$$\vdash P_0\{x := f\}P$$

where "P_0 is obtained from P by substituting f for all occurrences of x" [Hoare, 1969, p. 577]. Hoare took as the control statements in his example language those previously identified by Dijkstra as giving rise to particularly simple patterns of proof. Thus for iteration, he gave the following inference rule:

$$\text{If } \vdash P \land B\{S\}P \quad \text{then} \vdash P\{\textbf{while } B \textbf{ do } S\}\neg B \land P$$

This rule shows what can be asserted of an iteration controlled by a while statement, given a previous demonstration of a certain property of the statement, or program fragment, S that is being iterated.

Hoare's proposal, then, completed the project of showing how at least some program language constructs could be embedded in the familiar logic of propositions, thus enabling proofs about algorithms and programs to be carried out using the existing machinery of formal logic.

7.4 Constructive methods

The existence of a candidate proof theory for programs did not settle the question of how proofs of programs could be applied in practice, however. The methodology proposed by Naur, whereby a program created using "examples and intuition" was subsequently proved to be correct, had the disadvantage that the insights provided by the proof theory were not used in program development, and it was found in practice that it was rather difficult to argue mathematically about existing programs: the arguments needed to prove the correctness even of very trivial programs turned out to be rather long and tedious.

An alternative approach would be to employ a development process which would guarantee that the resulting programs were correct. Such a process had been outlined by McCarthy, who assumed as a prerequisite a theory stating when two programs or program fragments were equivalent. Given such a theory, transformations that preserved equivalence could be defined and “used to take an algorithm from a form in which it is easily seen to give the right answers to an equivalent form guaranteed to give the same answers but which has other advantages such as speed [or] economy of storage” [McCarthy, 1961, p. 225].

This idea was taken up by Dijkstra, who in 1968 published a paper outlining what he called “a constructive approach to the problem of program correctness” [Dijkstra, 1968a]. Rather than proving the correctness of an existing program, Dijkstra wanted to tackle the problem of deriving a correct program from “the specifications of the desired dynamic behaviour”. Taking as his example a simplified version of a multiprogramming problem, Dijkstra gave as the starting point of the derivation a simple and high-level version of the required program, without making clear whether this was intended to be a specification of the required program, or a form of it which could easily be seen to be correct.

Steps in the derivation process involved the introduction of variables enabling the required behaviour to be more precisely specified, the definition of assertions involving these variables, and the further articulation or refinement of the program to ensure that the assertions are satisfied at the appropriate times. The style of argumentation used was reminiscent of informal mathematics: Dijkstra stressed that he was not attempting to derive a program within a formal system, and that significant “mathematical invention” was required in the refinement process. Nevertheless, the emphasis was firmly placed on guaranteeing program correctness, and logic was used to justify individual steps in the argument.

Naur then proposed an approach which would combine the earlier work on assertions and correctness with the constructive approach suggested by Dijkstra [Naur, 1969]. Naur proposed identifying variables in terms of which the program requirements could be more precisely stated, by using assertions, or general snapshots. “Action clusters” were then defined to carry out the required operations on these variables; an action cluster was a sequence of program statements which would always be performed as a whole and whose effect could be characterized in terms of assertions. The correctness of the final program could then be assured by examining the relationship between the assertions defining the action clusters.

Hoare subsequently combined the ideas of constructive development with the formal logic of programs, giving a formal account of the development of an algorithm to perform certain manipulations on an array of data [Hoare, 1971]. He began by giving an informal explanation of the algorithm, but the development of a corresponding program was carried out completely formally.

Hoare started by giving a "rigorous formulation of what is to be accomplished", in the form of predicates defining the assumptions made at the beginning of the program and the required final state of the data being manipulated. As with Dijkstra and Naur, the general method for refinement that Hoare proposed started by introducing new variables required by the program, and defining their properties. Statements could then be written to solve the overall problem using the new variables, and these statements could be proved correct using the rules of the program logic. So after the initial introduction of two variables, Hoare could write: "At this point, the general structure of the program is as follows:

```
m := 1; n := N;
while m < n do "reduce middle part"
```

Furthermore, this code has been proved to be correct, provided that the body of the contained iteration is correct" [Hoare, 1971, p. 41]. Iteration of this procedure allowed Hoare to demonstrate a complete program together with the annotations required to demonstrate the correctness of the code using the rules of program logic.

However, Hoare was careful to point out that the procedure did not, in fact, prove that the final program was absolutely correct. Firstly, a separate proof was required to show that the program would terminate. Secondly, there were aspects of the program which were not covered by the initial formal requirements: for example, the algorithm was meant to rearrange the data in a given array, but the initial requirements used in the derivation of the program did not state that the array contained the same data at the end as at the beginning. He commented that it was difficult to formulate this requirement perspicuously, and that its inclusion would significantly increase the length and complexity of the proof.

Hoare described the method as "top-down ... split the process into a number of stages, each stage embodying more detail that the previous one" [Hoare, 1971, p. 45]; a similar approach was adopted, though in a less formal manner, by Niklaus Wirth [Wirth, 1971a]. For Wirth, "[i]n each step, one or several instructions of the given program are decomposed into more detailed instructions", the process terminating when a complete executable program in the desired language is obtained. Individual steps, known as *refinement steps* implied that some design decisions had been taken, often involving the introduction of new variables. Wirth's starting point resembled that of

Dijkstra rather than Hoare: he did not attempt to give a formal characterization of the problem to be solved, but instead presented a rather high-level program which was supposed to be an accurate rendition of the algorithm proposed for the solution of the problem.

By the early 1970s, then, McCarthy's programmatic suggestions about program transformation and proving the correctness of programs had been given concrete form for imperative languages as a methodology for program development, namely stepwise refinement, and a supporting logic by means of which such a development could be shown to deliver provably correct programs.

This raised the possibility, in principle at least, of treating programming as a purely formal activity, in which refinement steps corresponded to the application of inference rules in the appropriate calculus. A number of heuristics were proposed to assist in this process, notably the introduction of auxiliary data required in the program, together with the code fragments necessary to work with the new data.

This in turn raised the possibility of the extent to which the programming process could be automated. As Floyd saw it, there was an inescapable role for human creativity, because of the very large number of programs that might satisfy given input and output specifications. He imagined an interactive process of design, in which the checking of individual refinement steps and similar mechanical aspects would be handled by a computer, with the more creative design decisions being generated by the human programmer [Floyd, 1971].

7.5 Specifications and correctness

With the development of constructive methods, correctness was generalized from a simple property of programs to an equivalence relation between different programs which could be proved to have the same effect, or to implement the same specification. Program specifications, however, were normally only stated informally, and the technique of stepwise refinement recommended starting with a simple program whose correctness was self-evident and did not need to be formally established.

In his attempt to produce a completely formalized program derivation, however, Hoare had also stated the requirements for the example program formally, giving the assumptions made about the data at the start of the process and the required properties that it should have at the end. This more formal notion of program specification quickly became widespread. For example, Manna and Waldinger proposed automating the process of program development, using automatic theorem provers. They formalized the "specifications for the program to be written" as pairs of predicates, an “*input predicate* $\phi(\overline{x})$ and an *output predicate* $\psi(\overline{x},\overline{z})$” and claimed that “[i]n order to construct such a program, we prove the theorem $(\forall\overline{x})[\phi(\overline{x}) \supset (\exists\overline{z})\psi(\overline{x},\overline{z})]$” [Manna and Waldinger, 1971, p. 152].

With this increased explicitness and formalization of the notion of a specification came an associated change in what was meant by the correctness of a program. Previously correctness had been thought of primarily as a property of programs, but now it became understood as a relation between a program and its specification, thus making explicit something that had hitherto been assumed implicitly. For example, Liskov and Zilles believed that in general “[w]hat we are looking for is a process which establishes that a program correctly implements a *concept* which exists in someone’s mind” [Liskov and Zilles, 1975, p. 7]. The effect of increased formality, however, was that “a *specification* is interposed between the concept and the programs ... and the correctness of a program is established by proving that it is equivalent to the specification” [Liskov and Zilles, 1975, p. 8]. While recognizing that the equivalence between concept and specification could not be formally established, Liskov and Zilles argued that, at least for programs which were primarily intended to be used by other programs, the “hierarchical nature of the proof process” meant that “the concept which [a program] was intended to implement can safely be ignored” [Liskov and Zilles, 1975, p. 8]. The notion of correctness as a relationship between a program and a preferably formal specification was widely adopted:

> To determine whether a program is correct, we must have some way of specifying what it is intended to do; we cannot speak of the correctness of a program in isolation, but only of its correctness with respect to some specifications. After all, even an incorrect program performs *some* computation correctly, but not the same computation that the programmer had in mind. [Manna and Waldinger, 1978, p. 201]

This change had the effect of increasing the importance of specifications: if correctness amounted to correctness with respect to a specification, a program could only be said to be correct to the extent that its specification was clearly stated and understood. Furthermore, if that correctness was to be provable, the specification had to be written in a form accessible to existing proof methods, for example as a pair of input and output predicates as suggested by Manna and Waldinger.

A second consequence was that the analogy between software development and logical deduction was enriched. A formalized specification can be understood as defining a set of ‘axioms’; from the specification other expressions are ‘deduced’, by correctness preserving refinement steps, culminating in the production of ‘conclusions’ in the form of executable programs in the target programming language. Hoare's axioms did not quite function as inference rules, but they did enable the correctness of individual refinement steps to be checked, and heuristically suggested the form that such steps might take. The formal structure of software development is therefore seen as being identical to a logical theory: this tendency can be expressed by saying that software development was coming to be understood as a *quasi-deductive* activity.

The fact that specifications are viewed as axioms does not imply that they are immune from revision, of course. A program can be correct with respect to a specification that is completely inadequate from the users' point of view. The activity of specification revision, however, plays no role in quasi-deductive models: the overall development process is split between an initial phase in which a specification is fixed, and a subsequent phase of refinement and implementation based on the assumption that an adequate specification exists.

This understanding of the process of software development was widely adopted, even when development was not being carried out in a formal manner. A large number of 'methodologies' for software engineering were developed which explained how to develop software satisfying a specification by going through a number of steps which, even if expressed in a mixture of informal text and graphical notations, preserved the essence of the quasi-deductive approach, namely a process of refinement leading from a specification to a conforming implementation.

7.6 Structured programming

In the early 1970s, many of the concerns of the Algol research programme moved from the research community into the mainstream of commercial and industrial programming. They were widely thought to represent a new approach to the problems of programming, an approach that became identified by the term 'structured programming'. This section examines a number of different ways in which this term was understood.

The term 'structured programming' appears to have been coined by Dijkstra, who wrote some widely circulated 'Notes on structured programming' in August, 1969 [Dijkstra, 1969a] and presented a working paper titled simply 'Structured programming' at the NATO conference on software engineering techniques in that year [Dijkstra, 1969b]. For Dijkstra, the central issue was how to be assured of the correctness of "intrinsically large programs" [Dijkstra, 1969b, p. 222], where to get any reasonable assurance of a program's correctness it was necessary to have a very high degree of confidence in the correctness of the modules making up the program. The constructive approach to program development was outlined, with an argument for the use of specific control structures rather than jumps. Dijkstra also emphasized the use of abstract data structures in the program development process, along with a metaphorical description of a program as "an ordered set of pearls, a 'necklace"' [Dijkstra, 1969b, p. 225], each pearl representing a program module written in terms of the facilities provided by the module below it in the string.

The notion of 'pearls' derived, like the idea of constructive programming, from Dijkstra's experience in designing and writing a multiprogramming system [Dijkstra, 1968c]. This system had been designed as a hierarchy of levels with the characteristic that each level in the hierarchy was written strictly in terms of the level immediately below it. In explaining this idea, Dijkstra drew upon the old idea of program semantics being given in terms of a virtual machine: "Between two successive pearls we can make a 'cut', which is a manual for a machine provided by the part of the necklace below the cut and used by the program represented by the part of the necklace above the cut" [Dijkstra, 1969b, p. 255].

The 'Notes on structured programming' presented a more leisurely explanation of these ideas, together with complete examples of constructive program development. Some flowchart illustrations of the recommended control structures were given, and Dijkstra pointed out that "[t]hese flowcharts share the property that they have a single entry at the top and a single exit at the bottom" [Dijkstra, 1972, p. 19]: as a result, they could be treated as a single indivisible action in a sequential program. This was important in constructive programming, as it meant that a single high-level action could be refined by introducing a control structure whose properties could then be argued about independently of the rest of the program.

Dijkstra's notes were published in 1972 in a book entitled *Structured Programming* which contained in addition contributions from Hoare and Dahl. Hoare contributed an essay on data structuring, in which he argued that set theory could be used to define a range of data structures, and an essay on 'hierarchical program structures' by Hoare and Dahl described how programs were structured in the Simula 67 programming language. Simula 67 is considered in more detail in the following chapter; in the context of structured programming it was argued that it contained features which allowed the hierarchical structuring of programs, as recommended by Dijkstra.

As presented in these texts, then, structured programming encompassed not only a recommended set of control and data structures, but also a concern with the idea of the provable correctness of programs, the constructive method by which such programs could be produced, and a general scheme for program modules. This rich collection of ideas provided scope for selection and interpretation. For example, Henderson and Snowdon described an "experiment in structured programming" which adopted a "'top-down' structural approach with the hope that the program can be seen to be correct by its very structure" [Henderson and Snowdon, 1972, p. 38]. They discovered that the application of this technique did not prevent the occurrence of errors in the finished program, and concluded that "in such a technique we must apply formal methods" [Henderson and Snowdon, 1972, p. 51]. However, a response to this paper by Henry Ledgard concluded that "[t]he method used ... is strictly speaking not really 'structured programming', at least as conceived by Dijkstra", precisely because the authors had not "*formalized* and debugged each of the levels" [Ledgard, 1974, p. 49]. To add to the confusion, Ledgard, despite claiming to make a case for structured programming, defined a programming methodology of his own which not only combined the ideas of structured and top-down programming and stepwise refinement, but also adopted a programming style using the goto statement.

In 1973 Barbara Liskov offered a definition which extended Dijkstra's simple emphasis on correctness: "Structured programming is a programming discipline intended to support the production of correct, understandable programs which are easy to modify and maintain" [Liskov, 1973, p. 5]. An example of top-down decomposition of a program into modules was presented, using "the three sequential control structures proposed for structured programming"; these were justified not by an appeal to proof, however, but because they had a "1-in, 1-out" property which made the flow of control through a program easy to visualize. Indeed, "[t]here are many control structures other than [these three] which preserve the 1-in, 1-out property, and all of these are permissible in structured programming" [Liskov, 1973, p. 6]. Liskov, however, downplayed the significance of proof on the grounds of uncertainty about the form that a *specification* language would have to take in order to serve as a foundation for proofs of the correctness of programs.

Within the wider software industry, interest in the ideas of structured programming was stimulated by reports of their successful application on a project carried out by IBM for the New York Times [Baker, 1972a, Baker, 1972b]. This project was used as a vehicle to test a new approach to the management of software projects, the use of 'chief programming teams', which were intended to move projects away from a situation where each programmer had complete responsibility for everything to do with a particular part of the program to one where particular functional responsibilities were assigned to individuals. Like a surgical team, the project would be lead by an experienced designer, the "chief programmer", who would be assisted by "back-up programmers", "programming librarians" and other team members.

In addition to this novel form of organization, the project used a top-down approach to program design and implementation and employed structured programming, understood as "a set of rules that enhance a program's readability and maintainability … the rules state that any proper program — a program with one entry and one exit – can be written using only the following programming progressions" [Baker, 1972a], namely sequence, if-then-else statements and do-while loops. The project was described as being highly successful, and it was stated that "[s]tructured programming, and the organization and tools used to achieve it, were key factors in developing this kind of system" [Baker, 1972b]. Even though it was admitted that the use of chief programmer teams was not essential to employing structured programming, widespread interest in this project meant that 'structured programming' became understood as a general approach to software projects and not simply a technical approach to the organization of programs.

In 1973 interest was further encouraged when *Datamation*, a magazine that was widely read throughout the computing industry, published a special issue on structured programming. An introductory article was titled 'Revolution in Programming' and asserted that "[s]tructured programming is a major intellectual invention, one that will come to be ranked with the subroutine concept or even the stored program concept" [McCracken, 1973]. The articles in this issue of *Datamation*, however, characterized structured programming in terms derived more from Baker's description of IBM's experience on the New York Times project than from Dijkstra's theoretical writings.

In theoretical writings on structured programming, the issue of program correctness was of great importance; for example, Wirth wrote, echoing Dijkstra, that "[i]n order to achieve intellectual manageability, the elementary composition schemes must be simple. … The simplicity consists in the ease with which we can infer properties about the [composition scheme] from known properties of the constituent statement" [Wirth, 1974, p. 252]. This point was echoed in the *Datamation* articles, but with a slightly different emphasis and wider applicability: "since flow of control is simpler in a structured program, the development and execution of test cases to adequately debug the program is simpler … structured programs are very easy to read and verify for correctness" [Donaldson, 1973]. Verification was here being taken in an informal sense, however, and Donaldson went on to state that "study of program proof-of-correctness … has not yet produced any practical results" [Donaldson, 1973].

In particular, for a more practical audience, the key point about the adoption of a particular set of control structures was described in terms of increasing the readability of programs; for example, McCracken stated that "[u]sing only these constructions ... it is possible to write programs that can be read from top to bottom without ever branching back to something earlier ... Programs are accordingly *much* easier to read and understand" [McCracken, 1973]. The benefits of making programs clear and comprehensible extended not just to the writing of correct programs, but more widely across the whole software lifecycle. Thus it was held to be easier to test structured programs, and that ease of understanding made it simpler to correct errors in programs or to modify programs to provide new or enhanced functionality. The conferences on software engineering in the late 1960s and subsequent work had drawn attention to the costs of software development across the whole lifecycle, and suggestions of how to reduce these costs were highly attractive to industry.

There can be no doubt that structured programming made a significant and lasting contribution to programming language design and programming practice. Older languages, such as Fortran, which did not include the required control structures, were soon revised to include them, and they have been a constant part of all languages developed since. The controversy over the goto statement has died away, and in some modern languages it is not even available. Ideas of 'structure' were soon more widely applied: for example, the term 'structured design' was soon coined to describe an approach that emphasized a modular structure of programs consistent with structured programming [Stevens et al., 1974].

Structured programming emerged from work carried out in the logic research programme, and in particular from its concern with proving the correctness of programs. As it became known and applied in industry, however, the ways in which it was characterized changed. Management issues were emphasized, and the key point about the suitability of control certain structures was rephrased in the form of recommendations that would be immediately applicable by programmers, such as rules about the indentation of code [McCracken, 1973].

In particular, it is noticeable that any formal relationship between structured programming and program correctness was played down in favour of a more diffuse connection. As McCracken put it, "[p]rogram proving isn't yet a practical matter for programs of realistic size, but the theory influences the daily practice of programming anyway" [McCracken, 1973]. By using control structures which had been designed with a view to making proof easier, it was believed that programmers would obtain the benefits of programs that were easier to write correctly, to understand and to modify even without involving the construction of formal proofs.

7.7 Proof and testing

As noted above, one of the central goals of the research programme articulated by McCarthy in 1962 was that of replacing testing and debugging by proof, and this chapter has described the evolution of some of the techniques necessary to make the construction of program proofs feasible. At the end of the 1960s, researchers were optimistic about the possibilities for proof: Hoare emphasized the expense of testing and expressed the belief that "the practical advantages of program proving will eventually outweigh the difficulties, in view of the increasing costs of programming error" [Hoare, 1969, p. 579]. Despite expressing reservations about the power of the proof techniques then known, he was soon suggesting that "if a proof is constructed as part of the coding process for an algorithm, it is hardly more laborious than the traditional practice of program testing" [Hoare, 1971, p. 39], as well as offering a much stronger guarantee of reliability.

However, despite the success of structured programming, proof never became widely used in the software development industry, and testing never lost its role as the principal method for gaining assurance about the correctness of programs. Despite this, great progress was made in ensuring the reliability of software. Twenty years later, Hoare himself revisited the topic in a paper titled "How did software get so reliable without proof?". He commented that "the problem of program correctness has turned out to be far less serious than predicted" and suggested that the systematic application of traditional engineering techniques to software development was largely responsible for the observed increase in software reliability [Hoare, 1996].

This situation raises the question of to what extent the Algol research programme can be credited with improvements in programming practice if a central part of its programme, namely proof, was not at all widely employed? One possible answer to this question might be based on the observation that, as described earlier in this chapter, the Algol research programme introduced a general model of program development where programs were systematically derived from specifications by a process of *refinement*. This process could be carried out formally, but more often it was not; however, even informal versions of the process, which included testing, were found useful and widely adopted by industry. Rather than being completely opposed techniques with nothing in common, as McCarthy and Dijkstra suggested, proof and testing came to be viewed as complementary techniques for ensuring the correctness of software within the context of refinement-based methods.

The remainder of this chapter will give a more detailed analysis of this situation by drawing on accounts of scientific methodology developed in the philosophy of science which relate the notions of theory and experiment. A useful categorization of the possible positions was provided by Imre Lakatos [Lakatos, 1967], who modelled scientific and mathematical theories as deductive systems which relate axioms to 'basic statements', the 'final conclusions' drawn by the theory. In scientific theories the basic statements are said to be those which make some testable, empirical assertion.

These ideas can be applied to the software development process proposed by the Algol research programme by identifying the specification of a software system with the axioms of a deductive theory. In a top-down process, a high-level program is then written and its correctness argued for; by a series of refinement steps a low-level, executable program is then derived. Refinement steps are akin to inferences within the system, and the final program, fully expressed in the target programming language, is the equivalent of a basic statement, the point at which derivation stops. As in a scientific theory, where the basic statements make testable assertions, programs are run and tested, and accepted or rejected on the results of these tests.

Lakatos identified two basic types of theory, which he termed "Euclidean" and "quasi-empirical". These were distinguished by the place where truth values are "injected" into the system. Euclidean theories inject truth at the top, by assuming the truth of the chosen axioms and by truth-preserving inference steps deducing valid conclusions from them. Quasi-empirical theories inject falsity at the bottom, by testing the basic statements; a failed test indicates the falsity of a basic statement, which in turn, forces some modifications at higher points in the theory if consistency is to be maintained.

Software engineering as a Euclidean theory

It is evident from what has been said in previous sections that the overall view of software engineering evolved by the Algol research programme was that it was Euclidean, in Lakatos' sense. Correctness is injected at the top of the system, in the form of a fixed specification of what the system is to do. The task of software development was understood to be that of systematically developing from the specification a program which implemented the specified functionality, by means of a number of development steps which preserved correctness. Once techniques had been developed to carry out such derivations effectively, it was expected that alternative approaches such as testing and debugging would become obsolete:

I should like to point out that the constructive approach to program correctness sheds

> some new light on the debugging problem. Personally I cannot refrain from feeling that many debugging aids that are en vogue now are invented as a compensation for the shortcomings of a programming technique that will be denounced as obsolete in the near future. [Dijkstra, 1968a, p. 185]

Support for this position was largely rooted in the belief that testing on its own could not guarantee the correctness of software:

> Since it is well known that no foolproof methods exist of knowing that the last error in a program has been found, there is much practical confidence to be gained in never finding the first error in a program, even in debugging. [Mills, 1976, p. 269]

The role of testing within Euclidean methods

However, even though a broadly Euclidean approach to software development was widely adopted, formal proof was not, and testing retained a central role in assuring the correctness of software. A number of views have been put forward to justify or explain the coexistence of the two approaches in software engineering.

Shapiro [Shapiro, 1997] described the use of quasi-deductive methods and testing on the same project as a pragmatic approach employing two independent verification techniques to maximize the chances of producing a correct system. One motivation for such an approach might be an acknowledgment of the possibility of errors even in a proof of a program. In 1976, Gerhart and Yelowitz listed a number of errors detected in publications illustrating the formal derivation of programs. While broadly sympathetic to mathematical approaches to program development, they cited evidence in support of the fallibility of mathematical proof. While accepting that program verification had a role to play in ensuring that a program was "substantially correct", they concluded that "we must simply learn to live with fallibility" [Gerhart and Yelowitz, 1976, p. 206].

This implies a mixed Euclidean and quasi-empirical approach to software engineering, in which the typical response to an error detected in testing would be to correct the faulty code. Proponents of the Euclidean approach, however, have subsequently articulated a distinctive view of the role of testing: its purpose is not to assess directly the correctness of the software, but rather, by checking consistency between the specification and the program, to assess whether the development process has been correctly carried out. The recommended response to a failed test on this approach is not to correct the final artefact, but rather to modify and make less fallible the process that led to it, and then recapitulate the development: "The real value of tests is not that they detect bugs in the code, but that they detect inadequacy in the methods, concentration and skills of those who design and produce the code" [Hoare, 1996].

A third justification for including testing in a Euclidean model arises from the distinction between a program text and the executing program that is derived from it. As Fetzer pointed out [Fetzer, 1988], this is a contingent relationship: the fact that a given program behaves in a certain way when executed can only be established empirically, not by an examination of the program text, and so testing is necessary to verify the run-time properties of a program, even when the program itself is assumed to be correct as a result of a formal derivation.

Despite the use of the quasi-empirical technique of testing, however, these positions remain largely Euclidean in that they retain a belief in the feasibility of the goal of developing correct software. This assumption is not shared by more thorough-going quasi-empirical approaches, discussed below.

An inductive view of testing

The traditional view of testing was that programmers should keep running, testing and modifying a program until it passes all its tests. A passed test represents an injection of *correctness* at the bottom of the system, a confirmation that the program was behaving as required. As Lakatos points out, the belief that correctness can be injected at the bottom of a deductive system is tantamount to a belief in inductive methods, and the comparison between induction and the traditional account of testing has been made in the software engineering literature. The thought is that successful tests are singular statements of a program's correctness; from a set of such statements, we want to be able to infer that the program as a whole will give correct results at all times in the future.

Although this belief underlies much informal, small-scale programming practice, positive statements of an inductive principle are rare in the software engineering literature, no doubt because of the prominence of Dijkstra's early attack on the position. Ironically, however, a mixed position which included elements of an inductive approach was employed by Dijkstra in the development of the THE multiprogramming system. He describes how the system was designed in such a way that it could be formally proved that "the number of relevant test cases will be so small that [the designer] can try them all" [Dijkstra, 1968c, p. 344].

A later attempt was made by Goodenough and Gerhart to characterize the "logic of testing"; they proved a "fundamental theorem of testing" which "states that in some cases, a test *is* a proof of correctness" [Goodenough and Gerhart, 1975, p. 157]. The idea underlying this theorem was to partition the input space of a program in such a way that the successful completion of one test would imply that the program would function correctly for all other inputs from a given partition. This attempt foundered, however, on the difficulty in practice of finding a partition of the testing space with the required formal properties.

Quasi-empirical software development

Quasi-empirical views of software engineering would be those which characterize failed tests as injections of *incorrectness* at the bottom of the quasi-deductive system. This has suggested to a number of commentators an analogy between the testing of programs and the refutation of scientific theories: for example, Fetzer wrote that "it might be said that programs are conjectures, while executions are attempted—and all too frequently successful—refutations (in the spirit of Popper)" [Fetzer, 1988, p. 1062], and Dasgupta articulated the thesis that problem solving in design, including the design of programs, "is a special instance of (and is indistinguishable from) the process of scientific discovery" [Dasgupta, 1991, p. 353].

There was a significant tradition in software engineering which adopted a broadly quasi-empirical approach. In a rather Popperian spirit, this tradition did not take as its primary aim the development software that was absolutely correct, but instead accepted the inherent fallibility of software. In 1971, Bauer wrote in an overview of the then young field of software engineering that the aim of the discipline was "to obtain economically software that is reliable and works efficiently on real machines" [Bauer and Wössner, 1972]. It is noteworthy that Bauer refers not to the correctness of the software, but rather its reliability; unlike correctness, reliability is not understood in engineering as an all-or-nothing goal, but rather a property which systems can possess to different extents, depending on contextual and economic factors. A later paper surveying approaches to the study of reliability in software made this point explicitly: "Our position is that it is neither necessary nor economically feasible to get 100 per cent reliable (totally error-free) software in large, complex systems" [Schick and Wolverton, 1978, p. 105]. Rather than trying to ensure the absolute correctness of software, software engineers who accept the inevitability of errors have been concerned with techniques for developing fault-tolerant systems and for statistical characterizations of the reliability of software [Randell, 2000].

A further characteristic that we might expect to find in quasi-empirical software engineering is 'bold hypotheses, followed by dramatic refutations', as described in Popperian rhetoric about science. Much current practice can in fact be interpreted in this way. For example, it is a commonplace that commercial software products are full of errors, and frequently revised with patches or intermediate releases which correct faults. Traditional software engineering views this as a problem, feeling that a mature engineering profession ought to be able to do better. It is precisely what would be expected, however, if software engineering was in fact a quasi-empirical discipline[1].

Correctness and the user

In empirical science, quasi-empirical approaches can lead to the rejection of the axioms assumed to be the foundations of a theory. If the analogy is fully applicable, we should expect to find approaches to software engineering that allow for the revisability of the specification in the light of errors and problems discovered in the process of software development. An early statement of this position was made by Douglas Ross:

> The most deadly thing in software is the concept, which almost universally seems to be followed, that you are going to specify what you are going to do, and then do it. And that is where most of our troubles come from. The projects that are called successful, have met their specifications. But those specifications were based upon the designers' ignorance before they started the job. [Ross, 1968]

Similar views were later expressed by McCracken and Jackson, who commented that "systems requirements cannot ever be stated fully in advance, not even in principle, because the users doesn't *know* them in advance" [McCracken and Jackson, 1982, p. 31], an argument based on the observation that the development process itself frequently changed, among other things, users' perceptions of requirements.

This implies a view of program correctness which is based on something other than the relationship between a program and a specification. If specifications are revisable as users' insight in the system requirements grows, correctness should instead be understood as a relationship between a program and its users. As this distinction became appreciated, the process of checking that a program meets its specification became known as *verification*, whereas the process of checking that

[1]In the 1990s, a particular approach to software engineering characterized itself as 'empirical', based on the belief that "the most important thing to understand is the relationship between various process characteristics and product characteristics" [Basili, 1996]. In the Lakatosian framework, this approach would seem to fall squarely in the Euclidean tradition, but emphasizing the external, managerial aspects of the development process rather than the internal properties of software-related artefacts. What is being proposed appears to be an empirical study of a Euclidean process, not an empirical approach to development itself.

a software engineering artefact—either specification or program—meets the actual requirements of its users became known as *validation* [Boehm, 1984].

Responses to this situation took the form of proposals for software development methods that would involve the user extensively throughout. Originally known by terms such as 'prototyping' or 'evolutionary development', a similar approach is still extant, now often referred to as 'agile' or 'iterative and incremental' development. A partly anecdotal history of this approach that traces its roots back to the late 1950s has been compiled by Craig Larman and Victor Basili [Larman and Basili, 2003].

Such approaches do not view the software development process as quasi-deductive; instead, development is viewed as a continuing dialogue between user and developer. Aspects of contemporary package software also appear to fit this model, with the functionality of a program evolving over a series of releases in response to direct or indirect demands from users. Recent work in the philosophy of science has described models in which candidate scientific knowledge is not articulated as part of a deductive structure, but rather emerges in the course of an unpredictable process in which scientists explore the 'resistances' provided by a variety of human and non-human actors. An early example of this approach was the actor-network [Callon, 1987], and it has been taken up and refined in Pickering's notion of the 'mangle' [Pickering, 1995]. However, it is beyond the scope of this thesis to explore further the connections between this work and evolutionary approaches to software development, though some attempts to link software development with the ideas of post-modernism have been made [Robinson et al., 1998, for example].

7.8 Conclusions

This chapter has considered the influence that the Algol research programme had on the practice of software development. The ideas that became popularized under the label of 'structured programming' were widely influential in the computing industry, and widely perceived to have introduced a more formal approach:

> Before this decade of intense focus, programming was regarded as a private puzzle-solving activity of writing computer instructions to work as a program. After this decade, programming could be regarded as a public, mathematics-based activity of restructuring specifications into programs. [Mills, 1986]

The emphasis on specifications was key to the new programming techniques, and also became a cornerstone of software engineering practice more generally, introducing the idea of Euclidean models of the software lifecycle which covered not only programming but also other activities such as design, testing and maintenance. In this context, Boehm stressed the "extreme importance" of "a complete, consistent, unambiguous specification", in the absence of which problems could be anticipated in many other stages of development [Boehm, 1976].

Structured programming was frequently [McCracken, 1973, Knuth, 1974] described as a 'revolution' in programming, and it is interesting to consider how well this usage corresponds to Kuhn's sense of the term. There was certainly a sense of crisis associated with software development in the late 1960s and early 1970s, as evidenced by the NATO conferences on software engineering, and many people greeted the ideas of structured programming as a novel technique which would address these practical problems and make software development a more straightforward and predictable process. However, for Kuhn revolution is associated with the adoption of a new paradigm, and as the last two chapters have argued, structured programming can be viewed as the outcome of a logic-inspired paradigm whose revolutionary moment came with the publication of the Algol 60 report. So the application of Kuhn's historiographical schema in this case seems not to be straightforward, with the perception of crisis and the adoption of a new paradigm occurring at different times in the research and industrial communities.

An alternative model for the adoption of structured programming can be found in the traditional model of new ideas being developed in a research environment and then, when mature, transferred for application in an industrial setting. However, it is apparent that the ideas themselves may be significantly altered in such a transfer. Structured programming as conceived of by industry highlighted certain aspects of the academic work while ignoring or downplaying others. In particular, program proof and the elimination of testing and debugging was a central goal of researchers in the Algol paradigm, but presentations aimed at industry downplayed these aspects, emphasizing instead issues to do with the management of software projects, even though these formed no part of the theoretical notion of structured programming.

Chapter 8

The unification of data and algorithms

In the early 1970s, programs were frequently characterized as having two main aspects, namely the data structures that the program required and the algorithms used to manipulate the data. Programming languages were described along similar axes: a typical example is Pascal which defined control and data structures largely independently [Wirth, 1971b].

Work on data types, however, had made a strong case for considering these two aspects together, and this led to research into a new form of abstraction which would define a particular data structure together with the algorithms needed to make use of it, and present it to programmers as a single entity. As described in Chapter 6, this work was pursued as part of the Algol research programme, and in the early 1970s a number of languages were proposed which supported the idea of data abstraction in various ways.

This chapter examines in more detail the development of programming language support for data abstraction, and argues that a stable configuration of ideas, one that has profoundly influenced the design of programming languages up to the present day, was achieved by the Smalltalk language. Although drawing on work in the Algol research programme, the designers of Smalltalk were also strongly influenced by ideas from completely different areas, and the chapter concludes by arguing that Smalltalk marks a limit to the influence of logic on programming language design.

8.1 Simulation languages

A number of important ideas about the unification of data and algorithms emerged from attempts to write programs to perform simulations. Simulation had always been an important application of digital computers, and in the early 1960s a number of general-purpose simulation languages were developed to make it easier to write programs to simulate particular systems; these languages included SIMSCRIPT [Markowitz et al., 1963] and GPSS [Gordon, 1961] and two slightly later languages, Simula [Dahl and Nygaard, 1965] and SOL [Knuth and McNeley, 1964].

Of these, Simula turned out to be particularly influential on later work on the design of general-purpose programming languages. It was developed at the Norwegian Computing Centre by Kristen Nygaard and Ole-Johan Dahl, and had its roots in work in operational research carried out by Nygaard, specifically in “the necessity of using simulation, the need of concepts and a languages for system description, lack of tools for generating simulation programs” [Nygaard and Dahl, 1981, p. 440]. In particular, Dahl and Nygaard noted that for various technical reasons “simulation programs are comparatively difficult to write in machine language or in ALGOL or FORTRAN” [Dahl and Nygaard, 1966, p. 671] and that for this reason alone it would be convenient to develop specialized simulation languages.

Simula and SOL were both influenced by, and in some ways modelled on, existing high-level languages and in particular Algol 60; indeed, Simula was designed to contain Algol as a subset. Nevertheless, the demands of the specialized problem domain of simulation meant that there were also significant differences. For example, Simula was intended to be not only a programming language, but more generally “a language for a precise and standardized description of a wide class of phenomena, belonging to what we might call ‘discrete event systems’” [Dahl and Nygaard, 1965, p. 1]. (An echo of Algol 60, which had been initially characterized as a language “to describe computational processes” [Naur et al., 1960, p. 300] rather than simply a programming language, can be heard here.) The systems of interest were initially characterized as consisting of a number of active components, or ‘stations’, which processed data held in passive components, or ‘customers’. Inspired by examples such as an office with a number of clerks dealing with customers or a production line in a factory, each station maintained a queue of customers, and once a customer had been dealt with it could be passed on to join the queue at another station, thus modelling its progress through the system.

By the time the language was implemented in 1964, the two concepts of ‘station’ and ‘customer’ had been merged into a more general notion of ‘process’ which combined both the data associated with the passive customers and the operations carried out by the active stations. This generalization, which was very similar to the approach adopted by SOL, was the result of experience gained in modelling a greater range of systems and also in implementing simulations based on the resulting models [Nygaard and Dahl, 1981].

Simula made use of and extended the Algol concept of a block: “An ALGOL program (block) specifies a sequence of operations on data local to the program, as well as the structure of the data themselves. SIMULA extends ALGOL to include the notion of a collection of such programs, called ‘processes,’ conceptually operating in parallel” [Dahl and Nygaard, 1966, p. 671]. Technically, this extension had to do with the lifetimes of blocks: in Algol, one block could be defined inside another, with the consequence that any data defined in the inner block could only be maintained so long as the outer block was still in existence. For simulation programs, however, the lifetime of data in the program was unpredictable and depended on the events being simulated, so the Algol discipline was too restrictive. Simula therefore generalized the notion of a block so that “a process may remain and operate after [the block in which it was created] is out of the system, i.e. the life spans of different processes may overlap each other in any way” [Dahl and Nygaard, 1965, p. 14].

Individual processes in Simula therefore shared many of the properties of blocks in Algol: they contained both data, defined in local variables, and statements defining how that data was to be processed; these statements could belong to the block body or be contained in procedures and functions local to the block. A class of processes with the same structure was defined by an ‘activity’, which was essentially the same as an Algol procedure declaration. However, whereas in Algol a program was defined by a single, top-level block, in simulation languages the emphasis was placed rather on the system constituted by a number of processes, or blocks, executing simultaneously: in SOL, for example,“[a] complex system can be represented as a number of individual processes, each of which follows a *program* very much like a computer program” [Knuth and McNeley, 1964, p. 401]. The ‘master program’ did not encode an algorithm, as in Algol, but instead defined the processes that would exist initially and the data items that they used to communicated with each other.

8.2 Modelling with records

Because of the nature of simulation systems, languages like Simula were naturally described as providing facilities to *model* certain aspects of the real world. This modelling capability became understood as providing programmers with the ability to work with structured data, where an individual object could be characterized within a program by a collection of data items of varying types, with the number and type of the data items required varying from object to object. Apart from simulation languages, the ability to work with structured data was available in certain other areas, notably business data processing where Cobol, for example, provided the ability to exhaustively describe the structure of data stored in files.

Scientific languages such as Fortran and Algol did not provide such capabilities, however, and as discussed in Section 6.6 in the mid-1960s proposals were put forward, most notably by Wirth and Hoare, for adding a general record handling capability to such languages. These proposals made the connection with modelling clear: for example, Hoare wrote that "we often need to construct within the computer a *model* of that aspect of the real or conceptual world . . . In such a model each object of interest must be represented by some computer quantity . . . Such a quantity is known as a *record*" [Hoare, 1968, p. 294].

In their most developed form, proposals for record handling defined the relationship between the real world and the computer model in terms of four properties [Hoare, 1968]. Firstly, objects were considered to possess a number of *attributes*, each of which was modelled by a data item stored in a *field* of the record. Secondly, similar objects would naturally have the same kinds of attributes, though perhaps with different values. Objects could therefore be grouped into classes, and a *record class* in a program would define the attributes belonging to a particular class of objects. Next, it was also considered important to model relationships between objects: in the simple case of functional relationships, this was done by defining a new kind of data value which defined a *reference* to a record, and allowing records to hold references to other records to which they were related. Finally, it was recognized that many classes consisted of disjoint subclasses of objects, in the way that the class of vertebrates consists of the subclasses of mammals, birds and so on. The proposals allowed record classes to contain subclasses, with 'private' fields that defined attributes that applied only to objects belonging to certain subclasses.

Hoare and Wirth's record handling proposals on the one hand and Simula on the other therefore represented two alternative proposals for extending Algol 60 to permit the manipulation of structured data which modelled real-world entities. Simula achieved this by generalizing the Algol notion of a block; however, Hoare pointed out that this integrated the record concept with that of the process, defined by "a rule of behaviour as specified by procedural statements" [Hoare, 1968, p. 330], and brought with it the complexities of parallel processing. Records, by contrast, provided a new language feature which isolated the central problem of handling structured data, thus reinforcing such characteristic themes of the Algol research programme as clarity and the need to be able to understand and control the behaviour of programs.

8.3 Simula 67

Simula 67 was a revised version of the Simula produced as a result of experience gained with Simula and also in response to proposals made for using records to handle structured data.

Unlike Simula, Simula 67 was intended as a general purpose programming language. It was assumed that high-level languages like Algol had succeeded in the goals of enabling "precise formal description of computing processes" [Dahl et al., 1968] and making it easier for non-specialists to write programs. Simula 67 was intended more generally to help "those who are confronted with the task of organizing and implementing very complex, highly interactive programs" [Dahl et al., 1968, p. 1]; simulation programs were considered to fall into this category, but were no longer the sole focus of interest.

In its basic structures, however, Simula 67 was very reminiscent of the original Simula. It was recommended that the components that problems were divided into should each be describable as individual programs, implemented as before by an extended version of the blocks of Algol 60. Through a terminological change influenced by Hoare's work on record classes, the 'activities' and 'processes' of Simula were renamed as *classes*, and *objects*. Objects, like the processes of Simula, consisted of "an aggregated data structure and associated algorithms and actions" [Dahl et al., 1968, p. 5]. The latter consisted of local procedures which could act on the data stored in an object, and a block body which could be executed in a quasi-parallel fashion along with the bodies of other objects.

A significant innovation in Simula 67 was the introduction of *prefix classes*, intended as an alternative to the record subclasses that Hoare had described [Dahl and Nygaard, 1968]. The idea was that the definition of a new class could specify a single prefix class: the attributes of the prefix class would become attributes of the new class, and further attributes could be added to specialize the concept being modelled by the class. Class prefixing could be carried out repeatedly as often as required, allowing a hierarchy of classes to be defined.

Two significant aspects distinguished Simula 67's prefix classes from the record subclass proposed by Hoare. Firstly, prefix classes are more flexible than record subclasses. Hoare's proposal required all the subclasses of a given record class to be specified at the point of definition of the class. In Simula, on the other hand, any class can be used as prefix in any other class definition, giving an ability to reuse code that went beyond that offered by record subclasses. Consider, for example, the idea of a linked list, a dynamic data structure of records or objects linked by embedded references. Linked lists of many types are required in programming, and it would be nice to find a way of defining the concept of a linked list once and for all, rather than having to repeat the relevant definitions whenever a new type of list is required. In Simula 67 this can be done by defining the basic linked list functionality in a class which is then used as a prefix class to make linked lists of a particular sort of data: the required data fields are simply added when defining the new class. By contrast, when using records the definitions for the linked list would have to be repeated in the record class definition for every type of data that was to be stored in a list.

A second point of difference was the notion of *virtual quantities*. Using this mechanism, a prefix class could declare a field, say, which it does not itself define, but which it is planned will be defined in subclasses. For example, a vehicle class might define a field called ‘capacity’, even though that field was only defined in the subclasses of vehicle. The importance of this notion lies in the ability to define the capacity field differently in different subclasses of vehicle. A programmer might then refer to the capacity of a vehicle without knowing in detail what sort of vehicle is being referred to at run-time. Record handling proposals contained no similar capability: the emphasis on the concept of typing in the Algol research programme made it highly desirable that the every field in a record was fully defined when a program was compiled.

8.4 Data abstraction

In Section 7.6 it was argued that structured programming became principally identified with two of the ideas presented by Dijkstra, namely the use of a restricted repertoire of control structures and the employment of a top-down approach to program development. A third idea, that programs could be structured as a layered hierarchy of machines, was rather overlooked and became confused with the view that program structure was a hierarchical decomposition of functions, themselves identified as part of the top-down method. For example, Wirth described the method by stating that “[i]n each step a given task is broken down into a number of subtasks” [Wirth, 1971a, p. 226] and later that “an abstract program emerges, performing specific operations on abstract data . . . The operations are then considered as the constituents of the program which are further subjected to decomposition” [Wirth, 1974, p. 249].

As discussed above, Simula and Simula 67 incorporated an extension of Algol’s blocks which provided a way of unifying data and operations. However, for reasons that will be considered in more detail below, the details of Simula’s classes were not widely adopted in other languages, despite the inclusion of an extended discussion in the widely read 'Structured Programming' book published in 1972 [Dahl et al., 1972]. Instead, the early 1970s saw extensive discussion of and experimental proposals for new language mechanisms intended to provide support for more data-oriented program modules.

Writers on structured programming did of course recognize that control structures and data needed to be considered together: "[a]s tasks are refined, so the data may have to be refined, decomposed or structured, and it is natural to refine program and data specifications in parallel" [Wirth, 1971a, p. 221]. However, the commonest form of program module remained an Algol-like block, thought of primarily as defining a single operation. Blocks could contain local data or make reference to data defined in outer blocks, but they did not provide an adequate means of dealing with data generally.

One important theme in generating more complex proposals was the idea of isolating data and only allowing direct access to it to limited parts of a program. Various advantages were thought to follow from this. For example, the designers of the BLISS language identified as a significant problem the fact that, in the domain of systems programming, data structures frequently needed to be changed. It was important, therefore that "the structure definition and the algorithms which operate on the elements of a structure must be separated in such a way that either can be modified without affecting the other" [Wulf et al., 1971, p. 787]. This was achieved by enabling access to the elements of a data structure through a function-like interface, and defining the data structure along with an algorithm for accessing the elements of the structure. If the data structure was changed, the algorithm for accessing its components would also need to be changed, but code which made use of the functional interface would be unaffected. It was hoped that this would increase the ease with which programs could be modified and reused.

This line of thought was taken further by James Morris under the general heading of 'protection' [Morris Jr., 1973]. As well as describing programming language mechanisms which would enable a data structure and a set of procedures to be closely associated, Morris described methods for preventing other parts of the program from accessing the data structure directly. A related approach was described by Stephen Zilles as "procedural abstraction ... the technique of representing system components in terms of one or more procedures such that interactions among components are limited to procedure calls" [Zilles, 1973]. As this description suggests, it was widely assumed that the procedure, or block, was the fundamental type of program module, and proposals typically tried to show how procedures could be used or adapted to provide a structure that was more focused on data.

The question of whether program modules were in fact best understood as simple functions was explicitly addressed by David Parnas, who concluded on the contrary that "it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart". Instead, Parnas appealed to a principle of 'information hiding', and recommended that "one begins with a list of difficult design decisions which are likely to change. Each module is then designed to hide such a decision from the others" [Parnas, 1972, p. 1058]. This idea is clearly related to that of Dijkstra, who described program modules as "pearls", each embodying "a specific design decision (or, as the case may be, a specific aspect of the original problem statement)" [Dijkstra, 1969b, p. 87].

One specific type of design decision that could be hidden in a program module was the choice of representation for a particular data structure. As Parnas, like the designers of BLISS, pointed out, if knowledge of a particular data representation is shared between many modules, as typically happens when modules represent tasks, a change in that data representation will require associated changes to many program modules. An alternative approach is to conceal the choice of data representation in a single module, which will then make available to other modules a more abstract representation of the data, together with the ability to manipulate it. Dijkstra had recommended that "[s]uch a joint refinement of data structure and associated statements should be an isolated unit of the program text: it embodies the immediate consequences of an (independent) design decision and is as such the natural unit of interchange for program modification" [Dijkstra, 1969b, p. 87]. Parnas gave a slightly expanded version of this idea, recommending that "[a] data structure, its internal linkings, *accessing procedures and modifying procedures* are part of a single module" and "not shared by many modules as is conventionally done" [Parnas, 1972, p. 1056].

The connection between these ideas was made explicit in session on structured programming at an ACM meeting in 1973, where Barbara Liskov explained that "a hypothetical structured programming language could provide levels of abstraction as follows. We assume that an abstraction is presented to the user as an abstract data type together with the operations available on that type ... the entity 'level of abstraction' must be a syntactic unit of the language" [Liskov, 1973, pp. 6–7]. It was noted that the class concept of the Simula 67 programming language provided a similar feature, but because it made the data representation accessible to other modules, it did not fully support the desired notion of abstraction. A new language meeting these requirements was more fully described in the following year by Liskov and Zilles. They again emphasized the connection with structured programming, here understood as “a process of successive decomposition. The first step is to write a program which solves the problem but which runs on an abstract machine, one which provides just those data objects and operations which are ideally suited to solving the problem” [Liskov and Zilles, 1974, p. 50].

The language proposed by Liskov and Zilles, later named CLU, defined a new program structure, the *cluster*, intended to provide a way of implementing an abstract data type, understood as “a class of abstract objects which is completely characterized by the operations available on those objects” [Liskov and Zilles, 1974, p. 51]. A cluster defined a set of data objects which, from the point of view of a program using them, were completely abstract and could only be manipulated by using the operations defined in the cluster. The cluster itself defined a suitable representation for the abstract objects, in terms of other, lower-level clusters or the basic types provided by the language, and implemented the operations in terms of this representation. A cluster, therefore, was a concrete proposal for a program module corresponding to Dijkstra’s ‘pearls’, and provided a kind of abstraction closely related to the notion of a ‘cut’ in the necklace of pearls that Dijkstra had described.

Liskov and Zilles also saw a strong connection between the use of abstract data types and issues to do with proving the correctness of programs. The use of abstraction enabled the task of proving the correctness of a program to be split into two parts: proving the correctness of the abstract program that used the data abstraction, and proving the correctness of the implementation of the data abstraction itself. In order to carry out such proofs formally, however, it was necessary to have some way of writing formal specifications of abstract data types. Liskov and Zilles argued that an “*input-output specification*, which describes the mapping of the set of input values into the set of output values” [Liskov and Zilles, 1975, p 10] was suitable for defining procedural abstractions, but not for data abstractions, and considered various ways in which formal specifications of data abstractions could be given.

From the mid-1970s on, the notions of data abstraction and the formal specification of abstract data types became very significant areas of research and practical work, and many programming languages were developed in which these ideas were applied. This section has demonstrated the close relationship between the origins of this work and issues raised in the development and application of some of the ideas of structured programming, particularly those relating to program decomposition and proof. This suggests that this tradition of work should be seen as an integral part of the Algol research programme.

The key innovation in this work was the construction of a definitive notion of abstract data type. This raises the question of why a new concept was felt to be necessary, or rather why existing mechanisms, such as the classes of Simula 67, were not felt to be adequate. In the first place, classes were not types. Types were widely thought of as sets of data values with associated operations. This was consistent with the way in which basic types in programming languages were defined, and also provided a natural way in which the properties of the type could be formalized. Classes, on the other hand, were thought of as mechanisms for the production of objects, each of which contained data and operations; objects were therefore significantly different from data values. Furthermore, a different model of processing is involved: when an operation is invoked on an object, the object updates itself *in situ*; with an abstract data type, by contrast, a data value is passed as an argument to a function and an updated value returned.

Another important difference was the issue of *protection*: abstract data types were intended to provide a barrier which allowed programmers to manipulate data only by means of the provided operations. Simula 67, by contrast, allowed programs unrestricted access to the attributes of objects and therefore did not support a crucial part of the notion of abstraction. Finally, although Simula 67's classes did provide a unification of data and algorithms, they did a lot else besides. In particular, they provided support for a coroutine mechanism which allowed objects to exist in a quasi-parallel fashion. Although useful for applications such as simulation, this provided a complication that obscured what was felt to be the important new concept of an abstract data type.

8.5 Smalltalk

By the mid-1970s, then, the Algol research programme had developed a solution to the question of how to unify algorithms and data in programming languages, in the form of a fully articulated notion of abstract data type. A number of significant languages were based on the idea, pioneered by CLU, of a program module which defined and encapsulated an abstract data type. Perhaps the most notable of these languages was Ada, developed in the late 1970s and early 1980s by the US Department of Defense [Department of Defense, 1983].

However, later languages such as C++ and Java did not follow this style, adopting instead a form of program module derived from the Simula notion of a class. Languages adopting this style became known as *object-oriented* languages: this approach to programming language design became prominent in the early 1980s and has remained dominant until the present time. As with structured programming, earlier languages have been extended with object-oriented features: this occurred, for example, in the 1995 revision of Ada. An important influence in the development and adoption of object-oriented ideas was the Smalltalk language, developed at the Xerox Palo Alto Research Centre (PARC) from the early 1970s onward.

Smalltalk was designed as the programming language to be used on a new hardware device, described by Alan Kay and Adele Goldberg as “a personal dynamic medium the size of a notebook (the *Dynabook*) which could be owned by everyone and could have the power to handle virtually all of its owner’s information-related needs” [Kay and Goldberg, 1977, p. 31]. The Dynabook was to possess a high-quality graphical display that would be able to present information in a way not inferior to the printed page, the capability for high-fidelity sound reproduction, and a variety of input devices that would enable users to perform a multitude of tasks, including editing text, drawing images, and composing music. It was thought of as a “metamedium, whose content would be a wide range of already-existing and not-yet-invented media” [Kay and Goldberg, 1977, p. 40].

The Algol research programme had originated in the tradition of scientific programming carried out in the 1950s. A typical problem in this tradition was to devise algorithms for solving particular computational problems, and Algol was conceived of originally as a language for the expression and communication of algorithms. The background for Smalltalk, on the other hand, was the development of a highly interactive device which was intended to be usable by a wide range of people, including young children. Smalltalk was intended to be not only the language in which the system was coded, but also a medium through which users would work with the system. Programming was conceived not as the production of code by following an engineering-like process, but as an ongoing interaction with a complex and reactive system, an outlook which profoundly shaped the design of Smalltalk: Kay and Goldberg referred to Smalltalk as a “communications system ... implemented on small computers” [Kay and Goldberg, 1977, p. 31] rather than simply as a ‘language’.

The first stable version of the language, known as Smalltalk-72, was designed in 1972 and in use at PARC from 1973 on the “Interim Dynabook”, a small computer system being used to research aspects of the Dynabook idea. The instruction manual for Smalltalk-72 was written with an audience of high-school students in mind, and in style and content is strikingly different from the manuals written for languages in the Algol tradition [Goldberg and Kay, 1976]. As a communications system for the Dynabook, Smalltalk was intended to be used interactively, and use of the language is introduced under the heading “talking to Smalltalk”. Programming took place in a “Smalltalk dialog window”: the user could enter arithmetic expressions for immediate evaluation, or draw simple pictures by issuing commands to control a ‘turtle’ capable of drawing lines on the screen.

As well simple commands, conditional statements and iterations could be specified. An introductory example showed how to create an infinite loop which enabled the mouse cursor to be used as a simple drawing device. This provides a striking example of the effect that a different set of priorities could have on programming style: in scientific programming, an infinite loop was generally taken to be a severe error in a program and methods for proving that a program terminated when expected were intensively investigated. In Smalltalk, however, the advice given was simply “[t]o escape from the loop and get Smalltalk to listen to you again, press the key marked ‘ESC’” [Goldberg and Kay, 1976, p. 5].

Smalltalk was often described as a system which carried out simulation, and Simula was a significant influence on the development of Smalltalk. However, what was influential was the model of computation that was implied by Simula’s notion of simulation, which was taken up and generalized by Smalltalk. Syntactically, Smalltalk did not resemble Simula, or indeed any other language. For example, Smalltalk-72 included a number of hieroglyph-like characters, including a pointing hand, an eye and a smiley face, on the grounds that these had been found to convey the meaning of certain operations to children better than a set of reserved words. Furthermore, because the language was to be used interactively, there was no concept or textual representation of a complete program. Instead, users could add definitions to the system, and then use them interactively as required.

Kay had been directly inspired by Simula’s notion of an object, and in particular the integration of data and procedures into a single structure. He later wrote, “[f]or the first time I thought of the whole as the entire computer and wondered why anyone would want to divide it up into weaker things called data structures and procedures. Why not divide it up into little computers . . . ?” [Kay, 1996, p. 516]. However, rather than directly developing ideas of simulation or abstract data types, for Kay “[i]t was the promise of an entirely new way to structure computations that took my fancy” [Kay, 1996, p. 517]. At about the same time as he came across Simula, Kay also studied Lisp in detail and became fascinated by the idea of building an entire programming language on one single abstraction, in the way that lambda abstraction had been fundamental to the design of Lisp.

The original design of Smalltalk was written as a conscious attempt to emulate the McCarthy's original definition of Lisp [McCarthy, 1960], but based on a different primitive notion, namely the idea of message passing between objects [Kay, 1996, p. 531].

The Smalltalk-72 manual put it as follows: "Every entity in Smalltalk's world is called an object. Objects can remember things and communicate with each other by sending and receiving messages" [Goldberg and Kay, 1976, p. 6]. Despite the Simula-like terminology, however, it was recognized that there were important differences [Shoch, 1979]. Firstly, Simula's classes and objects were provided as an extension to Algol 60, leading to inconsistency in the way that different data items had to be treated; Simula 67, for example, had two assignment operations depending on whether the assignment involved an object or an Algol data item. Secondly, objects communicated in Simula by means of a "fairly typical procedure invocation" [Shoch, 1979, p. 72]. By contrast, the object receiving a message in Smalltalk could examine or manipulate the message, in effect deciding on its interpretation. To an extent this capability had been included in Simula 67, thanks to the notion of virtual quantities, but the motivation for including these had more to do with accessing attributes of objects than dynamically interpreting messages [Dahl et al., 1968, p. 24]. By contrast, Smalltalk made it a fundamental feature of the language, applying to all inter-object communication.

As in Simula, every Smalltalk object belonged to, or was an 'instance of', a class which defined the way in which its instances would respond to messages. Users of Smalltalk could extend the system by defining new classes, by providing suitably formatted text. Unlike Simula, however, classes were not thought of primarily as pieces of program text. Rather, in accordance with Smalltalk's overall philosophy of treating everything as an object, classes themselves were objects within the running Smalltalk system. Defining a class was thought of not so much as writing a program text as formatting a particular type of message which would tell the Smalltalk system to create a new class. As objects, classes could be sent messages, for example a message to create a new instance of the class.

The Smalltalk designers were aware of the potential for confusion in this approach. If all objects are instances of classes, then classes, being themselves objects, had to be instance of a class, the 'Class class'. The Class class would define the behaviour of classes, such as their ability to respond to messages asking for the creation of new objects. The Class class, however, had in turn to be thought of as an instance of itself, a form of self-inclusion that might be thought of as paradoxical in philosophy or set theory. In attempts to clarify what was going on here, explicit reference was sometimes made to Plato's theory of forms, the Class class being identified with the Form of the Good [Shoch, 1979].

Subsequent versions of Smalltalk introduced additional language features, notably subclassing, the Smalltalk analogue of Simula 67's prefix classes, in Smalltalk-76 [Ingalls, 1978] . A class could be defined as a subclass of another class, its 'superclass', from which it would 'inherit' behaviour. Behaviour that was shared by a number of class could therefore by this method be written in one superclass and inherited and reused in as many subclasses as necessary.

In a description of Smalltalk-76, Ingalls distinguished Smalltalk's "object oriented" approach from a traditional "function oriented" approach: in a function oriented language, the expression '$3 + 4$' would be interpreted as passing the arguments 3 and 4 to the operation '$+$'; in Smalltalk, on the other hand, it was interpreted as sending the message '$+4$' to the object representing the number 3. Whereas function oriented languages would provide a library of useful functions for programmers to use, Smalltalk provided "a set of well developed superclasses from which most of the system classes are derived" [Ingalls, 1978, p. 9]. User-defined classes could equally well be derived from any available superclass.

Smalltalk reached a definitive form in the 1980 and experienced a considerable growth of influence and use during the 1980s; in the 1990s it was for a while quite widely used industrially, particularly in the finance sector. The details of this later history are outside the scope of this thesis, however, and the relationship between Smalltalk and the Algol research programme will now be considered.

8.6 The relationship between Smalltalk and logic

The previous section has described the key features of Smalltalk, and suggested that its origins and the motivations of its designers were very different from those of the designers of Algol 60. Although the influence of Simula was acknowledged, the development of Smalltalk seems to have owed little else to the Algol research programme. This section supports this claim by describing ways in which the Smalltalk language itself differs from the notion of programming language, based on Carnap's notion of a formal language, found in the Algol research programme.

Smalltalk and the concept of a formal language

In one sense, every programming notation or language can be thought of as a formal language: without the existence of decidable syntactic rules it would not be possible for the notation to be processed by machine. However, a stronger claim was made in Chapter 5, namely that the Algol 60 report led to programming languages being considered to be formal languages in the sense of that notion articulated by Carnap and Tarski in the 1930s, and programs as terms in such a language. Smalltalk deviated in many ways from this notion, not least in its use of text to represent programs.

Certain aspects of the Smalltalk system made use of machine-readable text. One way in which the user could interact with the system was by typing text into a dialogue window; such text was interpreted by the system as a request to send a certain message to a specified object. However, other, non-textual, forms of interaction were also available, using additional interaction devices such as a mouse. In terms of their effect on the system, however, textual and non-textual interactions were semantically equivalent, both specifying that a message be sent to an object.

Text was also used for the definition of new classes, which were typed by the user into an editing window. However, whereas in more conventional languages the programming language text was taken as definitional of the program being written, in Smalltalk more emphasis was placed on the existence of the class within the complete Smalltalk system. For example, when a class text is brought up for editing, Ingalls describes the situation by saying that "[t]he class has thus provided a simulation of itself as structured text" [Ingalls, 1978, p. 10]. In the context of the Smalltalk system, the text entered by the user was not the definition of the class, but rather a *message* to the object in the system responsible for creating new classes. Other messages, perhaps utilizing different and even non-textual representations of the required behaviour, could equally well have been used.

Because Smalltalk was thought of not just as a programming language but more generally as a programming system, there was no clear notion of what a Smalltalk program might consist of. As opposed to the traditional model, where program texts were submitted to a computer system that would execute them, the Smalltalk system was the executive computer system and the user programmed by communicating with the system in various ways. A Smalltalk program could not therefore be isolated from its environment and dealt with in purely linguistic terms. Programming was not thought of as the task of constructing a linguistic entity, but rather as a process of working interactively with the semantic representation of the program, using text simply as one possible interface.

In particular, Smalltalk did not satisfy the properties stated by Tarski and Carnap as definitional of a formal language, discussed in Chapter 2. The first of these was that the basic signs in the language should be clearly described: this was not done for Smalltalk, and given the possibilities for non-textual communication with the system, it is not clear that it could have been done.

Similarly, the second condition, that the sentences of the language should be distinguished by purely structural means, was not satisfied. It was argued earlier that 'sentence' should be understood as denoting the linguistic unit expressing the speech act most important to the users of a language: indicative statements in the case of logic, and commands or programs in the case of conventional programming languages. In the Smalltalk system, as Kay stressed, the sole and unifying speech act was that of sending a message to an object. Certain textual forms for accomplishing this were specified, but these were not presented as being the only possibilities. As well as the possibility of non-textual messages, it would be quite possible within the Smalltalk metaphor for a user to send a garbled or meaningless message to an object: the effect of this would be defined by the object rather than by the syntax of the message.

In conclusion, then, the designers of Smalltalk do not appear to have thought of Smalltalk as a formal language, or to have made any attempt to present it in these terms. As this was a cornerstone of the Algol research programme, it is therefore possible to describe Smalltalk as marking a significant departure from the Algol paradigm, despite the influence of Simula on the language.

Smalltalk's computational model

The differences between Smalltalk and other languages are not simply to do with syntax, however, but extend to the general understanding of what computation is.

As described in Chapter 3, the principal motivation for the development of electronic digital computers was to automate calculation, and the canonical design that emerged, the so-called Von Neumann architecture, split the computer into a data store, and control and arithmetic units which processed data taken temporarily from the store. This architecture was reflected in the programming languages developed in the Algol tradition, which by 1970 were commonly viewed as consisting of features for expressing algorithms and a largely separate set of features for describing data structures. Within this tradition, programs were fundamentally seen as expressing algorithms, which in turn were understood as processes which carried out functional transformations on data.

Not all application areas fell neatly into this model, however: one which did not was the use of computers to carry out discrete event simulation, where the focus of interest lay in the computational process itself, or properties of it, rather than in the transformation between the initial and final states of the data presented to the program. As described earlier in this chapter, Simula 67 developed a way to support simulation within the context of the Algol research programme, a process which necessitated a number of extensions to Algol.

The designers of Smalltalk saw themselves as adopting the more radical approach of taking the notion of simulation expressed in Simula as the fundamental representation of computation. A Smalltalk system defines a simulated reality and rather than providing for the definition of isolated algorithms, the language provides a way for the user to interact with this reality [Shoch, 1979]. Therefore a semantic account based on functions does not seem likely to be the most natural way to understand the behaviour of a Smalltalk system.

It is notable that, like the Algol model, this design reflects aspects of contemporary computer architecture. Smalltalk was developed in the context of the Dynabook project, widely viewed as an originator of a type of ‘personal computing’ very different from traditional scientific computing. The Dynabook was intended to enable the user to interact simultaneously with a wide range of informational artifacts: documents, pictures, musical compositions and so on. Just as Pascal can be seen as reflecting the distinction between store and control in the von Neumann architecture, the design of Smalltalk can be seen as reflecting the conversational architecture of the Dynabook user interface.

Smalltalk and compositional semantics

A further way in which Smalltalk differs from conventional formal languages emerges in the relationship between syntax and semantics. In the metalogical scheme developed by Tarski and Carnap, the meaning of a sentence in a formal language stands in a functional relationship to its syntactic form. An interpretation in a language assigns a meaning to the smallest linguistic elements, and the meaning of larger expressions is defined in terms of the meanings of their component subexpressions.

The idea of message passing, and in particular the concept of dynamic binding, introduces difficulties into this scheme. Dynamic binding is associated with the notion of virtual quantities in Simula and was adopted as the default mechanism in Smalltalk. It provides a mechanism whereby the effect of sending a message cannot be predicted by the sender. Objects are accessed by means of references, but in both languages it is not always possible to tell from a reference exactly what kind of object is being referred to. The identity of the object being sent a message, then, is not in general known until the program is run, and the same message may invoke different behaviour on different occasions, depending on the history of the computation.

This means that the computational effect of an expression in Simula and Smalltalk cannot be told from a purely static inspection of the program text. It is only by running a program that its detailed behaviour can be known. This is an idea which has no counterpart in formal logic, where the meaning of an expression is entirely determined by its syntactic form. Dynamic binding introduces a new feature into object-oriented languages that appears to be incompatible with a key assumption of classical metalogic.

The programming process

Smalltalk had a novel idea of what the activity of programming consisted of, one in which the notion of inheritance was crucial. A Smalltalk program is not a self-contained linguistic entity which is compiled and run. Rather, the programmer works in the context of a pre-existing Smalltalk programming environment, itself written in Smalltalk, which provides support for both program development and execution. Programming is not viewed as an activity of constructing a discrete program, but rather as an activity of extending and modifying the environment, primarily using inheritance to reuse existing functionality. In such an environment, the notion of programming as a quasi-deductive activity can seem rather unnatural, and there is little if any evidence in the early Smalltalk literature of the concerns with program derivation or proving properties of programs that were characteristic of the Algol research programme.

It should be noted in passing that many aspects of the Smalltalk style of programming were also characteristic of Lisp programming environments, though in a less pronounced form. The contrast between this open, exploratory style of programming, and the more rigid, formal style of the Algol tradition is a striking feature of the history of programming.

8.7 Conclusions

By the early 1970s, programming language researchers had identified as a key issue the definition of linguistic structures that would support a unified treatment of data and algorithms. This chapter has described the development of two proposed solutions to this problem, namely the concept of abstract data types developed as part of the Algol research programme, and the approach to object-oriented programming embodied in Smalltalk.

Further, it was argued that Smalltalk marked a significant break with the Algol research programme, and in contrast with many of the languages developed in the preceding decade it owed little to the influence of logic. Not only were the inspiration and informal goals of the language quite different, emphasizing an interactive approach to programming and the use of computers, but the form of the language differed profoundly from those developed in the Algol tradition. The Smalltalk project was addressing some of the same issues as the Algol research programme, but proposed a quite distinct solution which owed little to the logic-influence approach characteristic of that programme.

Chapter 9

Conclusions

This chapter summarizes the main substantive and methodological conclusions of the thesis, makes a number of observations on the development of the field after the mid-1970s, and concludes by highlighting areas for possible future research.

9.1 The influence of logic

The primary aim of this thesis has been to provide an account of the ways in which mathematical logic was influential in the development of mainstream scientific and commercial programming languages. The account given recognizes, as have many others, that a crucial event in this development was the publication of the Algol 60 report [Naur et al., 1960]. The significance of this report is here explained, however, by proposing that it played the role of a concrete paradigm, in Kuhn's term, for what has been described in this thesis as the Algol research programme.

The key characteristics of the Algol research programme have been highlighted by making use of terminology introduced by Lakatos. For Lakatos, a research programme "consists of methodological rules: some tell us what paths of research to avoid (*negative heuristic*), and others what paths to pursue (*positive heuristic*)" [Lakatos, 1970, p. 132]. The negative heuristic tells researchers to preserve at all costs certain propositions, the "hard core" of the programme. For the Algol research programme, it was suggested in Chapter 5 that the hard core was roughly the proposition that programming languages should be understand to be formal languages in the sense established by mathematical logicians in the 1930s. This in turn was described, making use of Pickering's schematic account of conceptual innovation, as an example of *bridging* [Pickering, 1995].

By contrast, the positive heuristic of a research programme sets out the "research policy" of the programme, so that researchers will have a framework within which they can set their work, and which will save them from "becoming confused by the ocean of anomalies" [Lakatos, 1970, p. 135]. The positive heuristic of the Algol research programme was expounded most influentially by John McCarthy in the early 1960s [McCarthy, 1961, McCarthy, 1962]. A large part of what McCarthy suggested amounted to a programme for applying the metalinguistic framework of logic to the study of programming languages. Some of the details of this work were described in Chapters 6 and 7, and illustrated Kuhn's contention that a large part of normal science consists of puzzle solving rather than profound innovation. Pickering's description of this phase as one of *transcription*, where the well-understood ideas and techniques from one area are applied to a new area, reinforces this picture.

By the early 1970s, the Algol research programme had made significant progress and it was argued in Chapter 7 that many of its results were making their way to practical application in the form of 'structured programming'. In particular, this period saw the acceptance of particular forms of data and control structures and approaches to the methodology of program development that have remained central to the disciplines of computer science and software engineering ever since, and which represent central achievements of the Algol research programme in the period under study.

Paradigms and revolutions

The use of structural concepts such as 'paradigm' and 'research programme' in this account suggests further observations about the history of programming languages. For example, it could be argued that the Algol research programme was in fact the first paradigm within the field of programming language design, a claim based not on an isolated evaluation of the merits of the Algol report, but on the fact that after its publication the field acquired for the first time many of the characteristics of Kuhnian normal science, as described in Chapter 6. A consequence of adopting this position would be that the earlier work on automatic programming carried out in the 1950s would be described as being preparadigmatic. This is not to underemphasize the importance of earlier achievements, in particular Fortran, but draws attention to the fact that this work was not informed by a shared understanding of what the problems in the field were and on the best ways in which to make progress in solving them. The move towards automatic programming in the 1950s was driven by practical motives, including the desire to make the most economical use possible of the available machines, but innovation took the form of a number of small and largely independent initiatives, as described in Chapter 5, and it was only after 1960 that these began to coalesce into a coherent programme.

It could be objected that it is inappropriate to view the work of the 1950s as preparadigmatic, because there existed an existing paradigm for programming based on the use of machine code, a candidate concrete paradigm for which might be the textbook of Wilkes, Wheeler and Gill [Wilkes et al., 1951]. To some extent this must remain a question of judgement and interpretation, and there are facts, such as the initially negative reaction of many machine code programmers to Fortran, which bring to mind Kuhn's descriptions of the behaviour of adherents to an existing paradigm when confronted with a successor. However, it seems on the whole that the work on automatic programming was addressing issues and problems distinct from those relevant to machine code programming, and that it is better viewed as the preliminary to the formation of a new paradigm than as normal science in an established tradition of machine code programming.

Despite its influence and success, however, the Algol research programme did not encompass all subsequent work on programming languages. Chapter 8 described the early development of object-oriented programming and concluded that the Smalltalk project represented a new and independent development that, despite certain historical links, differed profoundly from approaches to program language design that were influenced by logic. It is outside the scope of this thesis to study in detail the subsequent development of object-oriented programming and the interaction between it and the logic-based tradition, but the following provisional remarks can be made.

Many widely-used programming languages of the present day, such as Java and C++, are described as being object-oriented, and owe a lot to the example of the ideas developed in Simula and Smalltalk [Stroustrup, 1994]. However, they do not differ as radically from Algol-like languages as Smalltalk did, and the currently dominant form of programming language can reasonably be described as a synthesis of the two approaches, as the following points suggest.

Firstly, the top-level structure of programs is based on the class concept evolved in the object-oriented tradition, not on the abstract data types of the Algol research programme, and the characteristically object-oriented features of inheritance and dynamic binding are widely used. Source code programs are structured as a set of class definitions, and an executing program is viewed as a network of intercommunicating objects, not as a single process.

However, the resemblance between contemporary programming languages and formal languages is stronger than in the case of Smalltalk. Programming in early versions of Smalltalk was a process of interacting with a complete system which included many aspects of what would now be classed as the computer's operating system, thus blurring the distinction between a program and its environment. Reuse and extension of existing code has replaced the Smalltalk model of an extensible programming environment, however, and programs are still largely understood to be fundamentally textual objects which are processed in various ways by a programming system which is in principle separate from the applications being written. As a result, the traditional metalogical distinctions can be applied to these languages, and research into, for example, the semantics of object-oriented languages has been able to make progress in a way that was difficult with Smalltalk.

Finally, it should be noted that contemporary programming languages include data and control structures that are clearly derived from the results of structured programming. These provide a layer of computational primitives which are used to define the classes that make up and object-oriented program. Like Simula, then, these languages are clear descendants of Algol-like languages.

Object oriented programming has frequently been described as a 'revolution', a description perhaps partly enabled by the frequent use of the Kuhnian term 'paradigm' to describe different approaches to programming language design. There does indeed seem to have been a significant change in the dominant class of programming languages, which until the late 1980s, consisted largely of languages which supported abstract data types. In a development reminiscent of the way in which structured programming constructs were introduced into older languages like Fortran, however, some of these languages later introduced object-oriented features, supporting the idea that object-orientation is now in fact the dominant approach.

If the adoption of object-oriented languages was a revolution, however, it appears to have been a conservative one, in the sense that many of the results from the previous paradigm have been carried across the revolutionary divide and preserved in the new paradigm. Specifically, as noted above, these results include the data and control structures of structured programming, and many of the metalinguistic assumptions of the Algol research programme. It has been suggested that such conservative revolutions characterize progress in mathematics and logic [Gillies, 1992], and it is therefore possible to speculate that the similar pattern of the object-oriented 'revolution' reflects the relationship that had been established by the Algol research programme between programming languages and logic.

It is possible to ask, however, if there are any substantive reasons why the adoption of object-orientation should have had a conservative character in which the new ideas were applied mostly to issues of large-scale program structure and not to all aspects of programming. One way to address this question would be trace the relationship between the belief, which became widespread in discussions of the 'software crisis', that the most significant problems for software engineering were those which arose in the development of large-scale systems, and the subsequent uptake of object-oriented ideas, not only in programming but also in program design. This, however, is a topic for future research.

A further question is the issue of why object-oriented languages have proved more successful than those based on abstract data types. A full answer to this question is outside the scope of thesis, but one possible factor is that object-orientation provides a better 'fit' with a significant range of applications than the simpler data abstraction model. For example, consider applications which run over a network of distributed computers: this scenario fits very naturally with Alan Kay's vision of a Smalltalk program being composed out of many objects, each with the capabilities of a computer. Further, since the widespread adoption of graphical user interfaces, programs are no longer in control of when input takes place, but are required to respond to unpredictable input from users. This again relates very naturally to the metaphor of objects responding to messages; in fact, as argued in Chapter 8, it is plausible that this was an important influence in the development of the ideas of object-oriented programming.

9.2 The nature of influence

This thesis has examined aspects of the influence a theoretical discipline, mathematical logic, has had on more practical activities, namely the design of programming languages and the construction of programs. Often, this direction of influence is described as the *application* of theoretical ideas to practice and is taken to ground a distinction between science and engineering. For example, in the 1980s a number of writers described how the logical and mathematical approach to software developed by the Algol research programme would enable the "craft" of programming to transform itself into a mature engineering discipline [Hoare, 1982, Shaw, 1990, for example].

Often, this process of application is seen as being unproblematic: for example, Chapter 3 considered that claims made by Davis and Mahoney that the invention of the computer could be characterized simply as the application of ideas from logic, and the computer as a 'byproduct', in Mahoney's term, of theoretical research in logic. However, it appears that a far less certain and more exploratory process took place than the simple term 'application' would suggest, and that within this process logical ideas were just one of many factors whose interplay led to the development of the computer. The close connection between the computer and logic appears, on the contrary, to have been established some years later, suggesting that the interaction between theory and application is not a causal process, but rather a question of interpretation, of a scientific community coming to see a new device in a particular way.

Similar points can be made about the relationship between logic and programming. Throughout the 1950s, there were several explicit attempts to apply logical ideas to programming, such as Turing's 'anticipation' of program proving, Elgot's use of formal language theory, and Hamblin's application of Łukasiewicz's ideas [Turing, 1949, Elgot, 1954, Hamblin, 1957]. However, in the absence of a more global understanding and acceptance of the role of logic, these pieces of work had little if any immediate influence. By contrast, once the Algol research program had become established in the 1960s, such applications of logic became routine. This suggests an answer to the question of how to explain the problematic time-lag identified by Jones between anticipations and the subsequent further development of similar ideas [Jones, 2003]: certain pieces of scientific work only gain their full significance when interpreted in the context of a research programme which shares their assumptions.

In the traditional view, application is seen as a separate stage that takes place after a research programme has delivered significant theoretical results, the results themselves not being substantially changed by their application. In this way, it was argued in Chapter 7 that the phenomenon of structured programming in the early 1970s can be seen precisely as the application of the ideas of the Algol research programme in practice. However, one striking feature of this process was the extent to which the theoretical ideas were modified: in particular, the importance of program proving was downplayed while new ideas about the management of software projects were treated as an integral part of the structured approach.

This phenomenon can be seen as related the third stage of Pickering's schema, *filling*, where results from the existing discipline do not provide a clear way forward and more open-ended work is required than in the transcription stage. Whereas logic had provided an approach to the design of data and control structures in languages, for example, its contribution to the practice of program development was more problematic. Informal top-down design became a popular practice, but formal program proving was not, and has not been, widely accepted. The belief that a proof-based approach to programming would guarantee the correctness of programs has been repeatedly criticized [De Millo et al., 1979, Fetzer, 1988, for example], for example, and a number of writers have commented on the lack of practical application of theoretical results [Arden, 1980, Mahoney, 1997, for example]. Proponents of the traditional model, such as Hoare, suggest that these problems can be addressed by further development of the theory, or a greater effort in education. By contrast, Pickering's scheme suggests the possibility that there may be limits to the extent to which a given theory can be straightforwardly applied in a particular area.

9.3 Methodological conclusions

The subject matter of this thesis belongs to what it usually characterized as internal history. However, as discussed in the introduction, one aim of the thesis was to explore the possibility of writing about such material while avoiding the methodological errors that traditional 'insider' history has been accused of. This section summarizes some of the ways in which this has been done, and the conclusions drawn.

Context and explanation

Insider history tends to describe historical episodes in terms of their relationship with the current state of knowledge, ignoring their historical context. An alternative tradition, associated with various approaches to the sociology of scientific knowledge, places emphasis instead on the external context of developments, and in particular social, political and economic factors. It was suggested in the introduction that such external factors might not be sufficient to explain certain internal features of a particular subject matter, and this thesis has accordingly tried to remain agnostic about what kinds of contextual factors might be explanatorily relevant in any particular case.

For example, in Chapter 2, Turing's machine table notation was examined in the context of contemporary logical work on computability. Accounts of Turing's work in the history of computing tend to stress its novelty, and to stress its role in the origins of modern computing. However, as argued in Chapter 2, there are many similarities between Turing's notation and that of recursive function theory and the λ-calculus, and drawing attention to these makes, it was suggested, better historical sense of Turing's work. In 1936, after all, Turing was making a contribution to the literature of mathematical logic, not to the then nonexistent subject of computer science.

In this case, then, related work in the theoretical discipline of mathematical logic provided a useful context in which to gain a better understanding of Turing's work. Chapter 3, by contrast, emphasized the importance of two other disciplines, namely computational mathematics and cybernetics, in the formation of the stable concept of an automatic digital computer described by von Neumann in 1945. As writers such as Paul Edwards have emphasized [Edwards, 1996], the Second World War provided a historical context which cannot be ignored in discussing the development of computers and the uses to which they were put, but this context is not sufficiently specific to explain the fine detail of proposals such as von Neumann's Draft Report.

External factors seem more directly relevant to the work described in Chapter 5: the principal motivation for the development of automatic programming in the 1950s was the need to make programming less laborious and time-consuming, hence enabling the anticipated demand for programming from industry and commerce to be met, and the emphasis on formula translation stemmed from the preponderance of scientific applications, itself attributable to the wartime origins of the computer. However, these factors are not, it was argued, sufficient to explain details such as the form of the mathematical expressions that were adopted in programming languages, and a more theoretical, 'internal' explanation was given for this.

As a final example of the possible range of relevant explanatory factors, it was suggested in Chapter 8 that aspects of the high-level structure of programming languages can be explained by the architecture of the machines on which the programming was to be carried out. This line of thought could be developed, for example, by considering the extent to which the area in which a given language was intended to be used, such as scientific or commercial applications, affected the design, included features and style of the resulting language.

In general, then, it can be concluded that a historical interest in the technical or internal details of a particular subject area does not preclude the possibility of giving a contextual account of particular episodes. However, obtaining an adequate understanding may in general involve a wider range of explanatory factors than are sometimes found in external histories.

The construction of new concepts

Insider accounts of technical invention often treat episodes of innovation as if they were indivisible moments of inspiration, not susceptible to analysis and explanation. This is a consequence of the Whiggish perspective which sees in the past only those aspects relevant to present work. In contrast, this thesis has emphasized the work involved in the construction of new concepts or techniques that may now seem to be obvious and unquestionable, the alternatives that were considered, and the reasons behind the choices that were made by the historical actors. A repeated pattern can be observed, in which a period of experimentation is followed by an episode of closure in which a standard solution is widely accepted. The reasons for which a particular solution is widely adopted differ from case to case, however.

This process appears at every stage of the historical story. Chapter 2 outlined the process by which a mathematical concept of effective computability emerged and gained acceptance, involving the interactions between the work of a number of logicians in the early 1930s. In this case, the provable equivalence of a number of widely different definitions appears to have been the deciding factor in generating closure.

A similar story can be told about the design of the computer embodied in von Neumann's Draft Report, as outlined in Chapter 3. The proposed EDVAC design appears to have won widespread acceptance very quickly, as it provided an effective solution to the problem of automatically programming electronic machines. The identification of computers of this type with Turing's universal machine concept, which is now often treated as axiomatic, took some time to become widely accepted, however. Furthermore, it was philosophical rather than technical arguments which seem finally to have made the difference in this case.

A similar extended, exploratory process is involved in the development of more technical details. Chapter 4 described how even such a fundamental feature of programming as performing two operations in sequence went through a period of negotiation before its final form was established; the solution adopted in this case was largely determined by the needs of the programmers of the machines, not by the intrinsic capabilities of the machines themselves.

A similar process to do with the types of formula that automatic translators would handle was described in Chapter 5. The interest in formula translation was prompted by a desire to widen the field of people who could program computers, but it was suggested that the form of the expressions handled was decided not by the mathematical needs of the users, but by the ease with which a particular class of formulas could be defined and processed.

This notion of the work required in conceptual innovation provides an explanation for the problem of 'blockages' noted in the introduction. These are episodes where a historical actor fails to make an inference or a discovery that with hindsight appears obvious or inevitable. Rather than simply accounting for such episodes as unaccountable failures, a more nuanced account of innovation enables us to recognize that even simple-looking innovations can require a complex and contingent process of work before their final form is established, and that in many cases this can only be achieved by the interaction and experience of many workers. To place responsibility on an individual for a lack of insight, or failure to make a particular move is in many cases to misunderstand the nature of the historical processes at work in technical innovation.

A striking example of this is the fact in the mid 1940s, neither von Neumann nor Turing included in their machine codes a single instruction to perform a conditional jump, despite being fully aware of the importance of this pattern to programming. In this case, the relevant general point that syntactic and semantic structures should match only became explicitly recognized with Dijkstra's work in the mid 1960s, after much experience in writing programs had been gained.

9.4 Directions for further work

Although this thesis has tried to avoid some of the familiar pitfalls in writing internal history, it does reflect traditional accounts in that it focuses on innovation rather the use of technology, in Edgerton's terms [Edgerton, 2006]. For example, in the account given in Chapter 5, the early 1950s were significant for the early development of autocodes, leading up to the development of Fortran. However, as Rosen pointed out, most programming in 1953 was being carried out on "the Card-Programmed Calculator, an ingenious mating of an Electromechanical Accounting Machine with an Electronic Calculating Punch" [Rosen, 1964]. Electronic computers were very thin on the ground, and for most programmers the use of autocodes would have seemed a remote and theoretical possibility. Similarly, in the early 1970s theoretical discussion of formal methods was being carried out against a background in which overwhelmingly the most widely used languages were still Fortran and Cobol [Rosen, 1972].

These observations suggest that there are a number of areas which would deserve more detailed examination in a more complete history of programming in this period. One significant omission is any consideration of the ways in which card processing systems were programmed to carry out complex computations, and the relationship between these techniques and those later developed for automatic calculators and stored-program machines. Punched-card technology was also of importance in the evolution of ideas about data structuring, and had been widely used in data processing applications since Hollerith's work during the 1890 US census [Austrian, 1982].

The relationships between different application areas and the programming techniques developed for them deserves more detailed attention than was possible in this thesis, which has focused primarily on scientific computing. As well as the main alternative areas of commercial programming and artificial intelligence, the 1950s and 1960s saw the development of a large number of special-purpose languages motivated by the perceived need to develop a language suitable for use in a restricted application area [Wexelblat, 1981].

A number of technical aspects of programming languages have only been mentioned in passing, but also deserve more detailed study. Significant omissions include concurrency and types and type theory, though in both these cases significant theoretical investigation only took place towards the end and after the period under study. It would also be worthwhile to study further the effect on programming language design of the perception in the late 1960s of a ‘software crisis’, and the subsequent construction and promulgation of a notion of software engineering intended to address the crisis.

As discussed in Chapter 8, object-oriented programming challenged some of the ideas of the Algol research programme, raising for example the question of whether all practical computation could be modelled on the mathematical notion of a function specifiable by its input and output characteristics, or whether the recursive structure of Algol 60’s block structure was sufficient for all needs. It would be worthwhile to study the ways in which logical models of programs and programming languages were refined in response to these questions. This in turn leads on to the question of the possible influence of programming language theory on logic: Gillies and Zheng have suggested that in general the interaction between two disciplines is a dynamic process, in which first one side dominates and then the other [Gillies and Zheng, 2001]. This thesis raises the interesting possibility that the influence of logic on programming discussed in this thesis might have been followed or accompanied by a period in which programming language research exerted a reciprocal influence on logic.

Finally, Chapter 3 described the way in which the new kind of computer gradually became understood in relation to Turing’s concept of the universal machine. It would be interesting to study in a similar way the history of the concept of the stored program machine. This is widely taken to be the defining feature of the machine architecture developed in 1945, but in fact the term was hardly used in the years up to 1950, and its salience was only gradually established. A related issue is the history of the technique of self-modifying code: initially recognized as a valuable practical technique, as described in Chapter 4, by the early 1950s this was thought to have much wider significance, for example in connection with the possibility of machine learning and thought. The autocodes of the 1950s excluded the possibility of writing self-modifying programs, however, and it found no place in the theoretical account of programming languages, based on a strict separation of syntax and semantics, given by the Algol research programme.

Bibliography

[Ackermann, 1928] Ackermann, W. (1928). Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen*, 93:118–133. Translated as “On Hilbert’s construction of the real numbers” in [van Heijenoort, 1967], pages 493–507.

[ACM, 1952] ACM (1952). *Proceedings of the 1952 ACM national meeting (Pittsburgh)*.

[ACM, 1966] ACM (1966). Proceedings of the ACM Programming Languages and Pragmatics Conference, San Dimas, California, August 8-12, 1965. *Communications of the ACM*, 9(3):137–232.

[Agar, 2003] Agar, J. (2003). *The Government Machine: A Revolutionary History of the Computer*. The MIT Press.

[Aiken, 1937] Aiken, H. H. (1937). Proposed automatic calculating machine. Reprinted in [Cohen and Welch, 1999], pages 9–29.

[Aiken, 1946] Aiken, H. H. (1946). The Automatic Sequence Controlled Calculator. In [Campbell-Kelly and Williams, 1985], pages 149–168. Lecture delivered 16 July, 1946.

[Aiken and Hopper, 1946] Aiken, H. H. and Hopper, G. M. (1946). The Automatic Sequence Controlled Calculator. *Electrical Engineering*, 65:384–391, 449–454, 522–528.

[Akera, 2001] Akera, A. (2001). Voluntarism and the fruits of collaboration: the IBM user group, Share. *Technology and Culture*, 42(4):710–736.

[Alt, 1948] Alt, F. L. (1948). A Bell Telephone Laboratories computing machine. *Mathematical Tables and other Aids to Computation*, III:1–13, 69–84.

[Anonymous, 1943] Anonymous (1943). ENIAC progress report. December 31, 1943. Quoted in [Stern, 1981].

[Anonymous, 1945] Anonymous (1945). Automatic high-speed computing: A progress report on the EDVAC. September 30, 1945. Quoted in [Metropolis and Worlton, 1980], p. 55, and [Aspray, 1990b], p. 38.

[Archibald, 1946] Archibald, R. C. (1946). Conference on advanced computation techniques. *Mathematical Tables and other Aids to Computation*, II(13):65–68.

[Arden, 1980] Arden, B. W., editor (1980). *What can be automated? The Computer Science and Engineering Research Study (COSERS)*. The MIT Press.

[Aspray, 1990a] Aspray, W., editor (1990a). *Computing Before Computers*. Iowa State University Press.

[Aspray, 1990b] Aspray, W. (1990b). *John von Neumann and the Origins of Modern Computing*. The MIT Press.

[Aspray and Burks, 1987] Aspray, W. and Burks, A. (1987). *Papers of John von Neuman on Computing and Computing Theory*, volume 12 of *Charles Babbage Institute Reprint Series for the History of Computing*. The MIT Press.

[Aspray Jr., 1980] Aspray Jr., W. F. (1980). *From Mathematical Constructivity to Computer Science: Alan Turing, John von Neumann, and the Origins of Computer Science in Mathematical Logic*. PhD thesis, University of Wisconsin-Madison.

[Austrian, 1982] Austrian, G. D. (1982). *Hermann Hollerith*. Columbia University Press.

[Babbage, 1864] Babbage, C. (1864). *Passages from the Life of a Philosopher*. London.

[Bachman, 1973] Bachman, C. W. (1973). The programmer as navigator. *Communications of the ACM*, 16(11):653–658.

[Backus, 1954] Backus, J. W. (1954). The IBM 701 Speedcoding system. *Journal of the Association for Computing Machinery*, 1(1):4–6.

[Backus, 1958] Backus, J. W. (1958). Automatic programming: properties and performance of FORTRAN systems I and II. In [National Physical Laboratory, 1959], pages 231–248.

[Backus, 1959] Backus, J. W. (1959). The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In [Unesco, 1960], pages 125–132.

[Backus, 1980] Backus, J. W. (1980). Programming in America in the 1950s — some personal impressions. In [Metropolis et al., 1980].

[Backus, 1981] Backus, J. W. (1981). ALGOL session: Transcript of question and answer session. In [Wexelblat, 1981], page 162.

[Backus and Heising, 1964] Backus, J. W. and Heising, W. P. (1964). FORTRAN. *IEEE Transactions on Electronic Computers*, EC-13(4):382–385.

[Baker, 1972a] Baker, F. T. (1972a). Chief programmer team management of production programming. *IBM System Journal*, 11(1):56–73.

[Baker, 1972b] Baker, F. T. (1972b). System quality through structured programming. In *AFIPS Proceedings of the 1972 Fall Joint Computer Conference*, volume 41 of *AFIPS Conference Proceedings*, pages 339–344. AFIPS Press.

[Barcan Marcus, 1962] Barcan Marcus, R. (1962). Interpreting quantification. *Inquiry*, 5:252–9.

[Barron et al., 1963] Barron, D. W., Buxton, J. N., Hartley, D. F., Nixon, E., and Strachey, C. (1963). The main features of CPL. *The Computer Journal*, 6(2):134–143.

[Basili, 1996] Basili, V. (1996). The role of experimentation in software engineering: Past, present and future. In *International Conference on Software Engineering*.

[Bauer, 1981] Bauer, F. L. (1981). Appendix 5 to [Naur, 1981]: Notes by F. L. Bauer. In [Wexelblat, 1981], pages 127–130.

[Bauer, 2000] Bauer, F. L. (2000). The Plankalkül of Konrad Zuse – revisited. In Rojas, R. and Hashagen, U., editors, *The First Computers—History and Architecture*, pages 277–293. The MIT Press.

[Bauer et al., 1957] Bauer, F. L. et al. (1957). Letter from GAMM members to Prof. John Carr, III, President of ACM. Reprinted in [Bemer, 1969], pages 160–161.

[Bauer and Wössner, 1972] Bauer, F. L. and Wössner, H. (1972). The “Plankalkül” of Konrad Zuse: A forerunner of today’s programming languages. *Communications of the ACM*, 15(1):678–685.

[Bemer, 1959] Bemer, R. W. (1959). Automatic programming systems. *Communications of the ACM*, 2(5):16.

[Bemer, 1969] Bemer, R. W. (1969). A politico-social history of Algol. In Halpern, M. I. and Shaw, C. J., editors, *Annual Review in Automatic Programming 5*, pages 152–237. Pergamon Press.

[Bemer, 1984] Bemer, R. W. (1984). Computing prior to FORTRAN. *Annals of the History of Computing*, 6(1):16–18.

[Benington, 1956] Benington, H. D. (1956). Production of large computer programs. In *Proceeding of the Symposium on Advanced Programming Methods for Digital Computers*, pages 15–28. Office of Naval Research.

[Bennett et al., 1952] Bennett, J. M., Prinz, D. G., and Woods, M. L. (1952). Interpretative subroutines. In [Forrester and Hamming, 1952], pages 81–87.

[Bergin and Gibson, 1996] Bergin, T. J. and Gibson, R. G., editors (1996). *History of Programming Languages – II*. ACM Press.

[Berkeley, 1949] Berkeley, E. C. (1949). *Giant Brains, or Machines that Think*. John Wiley & Sons, Inc.

[Berkeley, 1950] Berkeley, E. C. (1950). The relations between symbolic logic and large-scale calculating machines. *Science (New Series)*, 112(2910):395–399.

[Birnbaum, 1991] Birnbaum, L. (1991). Rigor mortis: a response to Nilsson's "Logic and artificial intelligence". *Artificial Intelligence*, 47(1–3):57–78.

[Bloch, 1947] Bloch, R. M. (1947). Mark I calculator. In [Harvard, 1947], pages 23–30.

[Bloch, 1999] Bloch, R. M. (1999). Programming Mark I. In [Cohen and Welch, 1999].

[Bloch et al., 1948] Bloch, R. M., Campbell, R. V. D., and Ellis, M. (1948). The logical design of the Raytheon computer. *Mathematical Tables and other Aids to Computation*, III(24):286–295.

[Bloor, 1976] Bloor, D. (1976). *Knowledge and Social Imagery*. The University of Chicago Press.

[Boehm, 1976] Boehm, B. W. (1976). Software engineering. *IEEE Transactions on Computers*, C-25(12):1226–1241.

[Boehm, 1984] Boehm, B. W. (1984). Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88.

[Böhm and Jacopini, 1966] Böhm, C. and Jacopini, G. (1966). Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371.

[Boole, 1854] Boole, G. (1854). *An Investigation of the Laws of Thought*. Macmillan.

[Booth, 1949] Booth, A. D. (1949). Relay computers. In [Cambridge University Mathematical Laboratory, 1950], pages 27–30.

[Booth, 1960] Booth, A. D. (1960). Opening address. In [Goodman, 1960], pages 1–7.

[Boring, 1946] Boring, E. G. (1946). Mind and mechanism. *American Journal of Psychology*, LIX(2):184. Quoted in [Edwards, 1996], page 188.

[Bosak et al., 1962] Bosak, R., Clippinger, R. F., Dobbs, C., Goldfinger, R., Jasper, R. B., Keating, W., Kendrick, G., and Sammet, J. E. (1962). An information algebra. *Communications of the ACM*, 5(4):190–204.

[Bowden, 1953] Bowden, B. V., editor (1953). *Faster Than Thought: a Symposium on Digital Computing Machines*. Sir Isaac Pitman & Sons, Ltd.

[Braffort and Hirschberg, 1963] Braffort, P. and Hirschberg, D., editors (1963). *Computer Programming and Formal Systems*. North-Holland Publishing Company.

[Briggs, 1946] Briggs, L. J. (1946). Impact of the war on science. *Electrical Engineering*, 65(1):8–10.

[Brooker et al., 1952] Brooker, R. A., Gill, S., and Wheeler, D. J. (1952). The adventures of a blunder. *Mathematical Tables and other Aids to Computation*, 6(38):112–113.

[Brooker and Wheeler, 1953] Brooker, R. A. and Wheeler, D. J. (1953). Floating operations on the EDSAC. *Mathematical Tables and other Aids to Computation*, 7(41):37–47.

[Burks, 1947] Burks, A. W. (1947). Electronic computing circuits of the ENIAC. *Proceedings of the I.R.E.*, 35(8):756–767.

[Burks et al., 1946] Burks, A. W., Goldstine, H. H., and von Neumann, J. (1946). Preliminary discussion of the logical design of an electronic computing instrument. Technical report, Institute of Advanced Study. Second edition (dated 2 September 1947) reprinted in [Aspray and Burks, 1987], pages 97–142.

[Butterfield, 1931] Butterfield, H. (1931). *The Whig Interpretation of History*. G. Bell and Sons, London.

[Callon, 1987] Callon, M. (1987). Society in the making: The study of technology as a tool for sociological analysis. In Bijker, W. E., Hughes, T. P., and Pinch, T. J., editors, *The Sociological Construction of Technological Systems*, pages 83–103. The MIT Press.

[Cambridge University Mathematical Laboratory, 1950] Cambridge University Mathematical Laboratory (1950). *Report of a Conference on High Speed Automatic Calculating Machines, 22-25 June 1949*. University Mathematical Laboratory, Cambridge. Reprinted in [Williams and Campbell-Kelly, 1989], pages 3–164.

[Campbell-Kelly, 1980] Campbell-Kelly, M. (1980). Programming the EDSAC: Early programming activity at the University of Cambridge. *Annals of the History of Computing*, 2(1):7–36.

[Campbell-Kelly, 2003] Campbell-Kelly, M. (2003). *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*. The MIT Press.

[Campbell-Kelly, 2007] Campbell-Kelly, M. (2007). The history of the history of software. *IEEE Annals of the History of Computing*, 29(4):40–51.

[Campbell-Kelly and Aspray, 1996] Campbell-Kelly, M. and Aspray, W. (1996). *Computer: A History of the Information Machine*. Basic Books.

[Campbell-Kelly and Williams, 1985] Campbell-Kelly, M. and Williams, M. R., editors (1985). *The Moore School Lectures: Theory and Techniques for Design of Electronic Digital Computers*, volume 9 of *Charles Babbage Institute Reprint Series for the History of Computing*. The MIT Press.

[Carnap, 1937] Carnap, R. (1937). *The Logical Syntax of Language*. Routledge & Kegan Paul Ltd.

[Carnap, 1939] Carnap, R. (1939). *Foundations of Logic and Mathematics*, volume I, number 3 of *International Encyclopedia of Unified Science*. The University of Chicago Press.

[Carnap, 1942] Carnap, R. (1942). *Introduction to Semantics*. Harvard University Press.

[Carpenter and Doran, 1977] Carpenter, B. E. and Doran, R. W. (1977). The other Turing machine. *The Computer Journal*, 20(3):269–279.

[Carpenter and Doran, 1986] Carpenter, B. E. and Doran, R. W., editors (1986). *A. M. Turing's ACE report of 1946 and other papers*, volume 10 of *Charles Babbage Institute Reprint Series for the History of Computing*. MIT Press.

[Ceruzzi, 1981] Ceruzzi, P. E. (1981). *The Prehistory of the Digital Computer, 1935–1945: A Cross-cultural Study*. PhD thesis, University of Kansas.

[Ceruzzi, 1983] Ceruzzi, P. E. (1983). *Reckoners*. Greenwood Press.

[Ceruzzi, 1997] Ceruzzi, P. E. (1997). Crossing the divide: Architectural issues and the emergence of the stored program computer, 1935–1955. *IEEE Annals of the History of Computing*, 19(1):5–12.

[Ceruzzi, 1998] Ceruzzi, P. E. (1998). *A History of Modern Computing*. MIT Press.

[Ceruzzi, 2001] Ceruzzi, P. E. (2001). A view from 20 years as a historian of computing. *IEEE Annals of the History of Computing*, 23(4):49–55.

[Chaitin, 2001] Chaitin, G. J. (2001). *Exploring Randomness*. Springer-Verlag.

[Cheatham, 1971] Cheatham, Jr., T. E. (1971). The recent evolution of programming languages. In [Freiman, 1972], pages 298–313.

[Cherry, 1950] Cherry, E. C. (1950). A history of the theory of information. In [Ministry of Supply, 1950], pages 22–43.

[Church, 1932] Church, A. (1932). A set of postulates for the foundation of logic. *The Annals of Mathematics (2nd Series)*, 33(2):346–366.

[Church, 1936] Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363.

[Cleave, 1960] Cleave, J. P. (1960). Application of formula translation to automatic coding. In [Goodman, 1960], pages 81–92.

[CODASYL Data Base Task Group, 1969] CODASYL Data Base Task Group (1969). Data base task group report to the CODASYL programming language committee. Technical report, General Electric Research and Development Center.

[Codd, 1970] Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.

[Cohen, 1999] Cohen, I. B. (1999). *Howard Aiken: Portrait of a Computer Pioneer*. The MIT Press.

[Cohen and Welch, 1999] Cohen, I. B. and Welch, G. W., editors (1999). *Makin' Numbers: Howard Aiken and the Computer*. The MIT Press.

[Colilla and Sams, 1962] Colilla, R. A. and Sams, B. H. (1962). Information structures for processing and retrieving. *Communications of the ACM*, 5(1):11–16.

[Copeland, 2004a] Copeland, B. J. (2004a). Computable numbers: A guide. In [Copeland, 2004b].

[Copeland, 2004b] Copeland, B. J., editor (2004b). *The Essential Turing*. Oxford University Press.

[Curry, 1929] Curry, H. B. (1929). An analysis of logical substitution. *American Journal of Mathematics*, 51(3):363–384.

[Dahl et al., 1972] Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R. (1972). *Structured Programming*. Academic Press.

[Dahl et al., 1968] Dahl, O.-J., Myhrhaug, B., and Nygaard, K. (1968). Common base language. Technical Report S-2, Norwegian Computing Centre.

[Dahl and Nygaard, 1965] Dahl, O.-J. and Nygaard, K. (1965). SIMULA – a language for programming and description of discrete event systems. Introduction and user's manual. Technical Report 11, Norwegian Computing Centre. (5th edition (1967) available at `http://www.edelweb.fr/Simula`. Accessed May 4, 2008.).

[Dahl and Nygaard, 1966] Dahl, O.-J. and Nygaard, K. (1966). SIMULA—an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678.

[Dahl and Nygaard, 1968] Dahl, O.-J. and Nygaard, K. (1968). Class and subclass declarations. In Buxton, J. N., editor, *Simulation Programming Languages*, pages 158–174. North-Holland Publishing Company.

[Dasgupta, 1991] Dasgupta, S. (1991). *Design Theory and Computer Science*. Cambridge University Press.

[Davis, 1988] Davis, M. (1988). Influences of mathematical logic on computer science. In [Herken, 1988].

[Davis, 2000] Davis, M. (2000). *The Universal Computer*. W. W. Norton & Company.

[De Millo et al., 1979] De Millo, R. A., Lipton, R. J., and Perlis, A. J. (1979). Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280.

[Department of Defense, 1983] Department of Defense (1983). *Reference Manual for the Ada Programming Language*. United States Department of Defense. ANSI/MIL-STD-1815A-1983.

[Department of Scientific and Industrial Research, 1946] Department of Scientific and Industrial Research (1946). A.C.E. the automatic computing machine. *Electronic Engineering*, 18(12):372–373.

[Diehm, 1952] Diehm, I. C. (1952). Computer aids to code checking. In [Forrester and Hamming, 1952], pages 19–21.

[Dijkstra, 1962a] Dijkstra, E. W. (1962a). An attempt to unify the constituent concepts of serial program execution. In [International Computation Center, 1962], pages 237–251.

[Dijkstra, 1962b] Dijkstra, E. W. (1962b). Some meditations on advanced programming. In [Popplewell, 1963], pages 535–538.

[Dijkstra, 1963] Dijkstra, E. W. (1963). On the design of machine independent programming languages. In Goodman, R., editor, *Annual Review in Automatic Programming, 3*, pages 27–42. Pergamon Press.

[Dijkstra, 1965] Dijkstra, E. W. (1965). Programming considered as a human activity. In [Kalenich, 1965], pages 213–217.

[Dijkstra, 1968a] Dijkstra, E. W. (1968a). A constructive approach to the problem of program correctness. *BIT*, 8:174–186.

[Dijkstra, 1968b] Dijkstra, E. W. (1968b). Go to statement considered harmful. *Communications of the ACM*, 11(3):147–8.

[Dijkstra, 1968c] Dijkstra, E. W. (1968c). The structure of the “THE”-multiprograming system. *Communications of the ACM*, 11(5):341–346.

[Dijkstra, 1969a] Dijkstra, E. W. (1969a). EWD249: Notes on structured programming. Unpublished manuscript, available at `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF` (accessed May 4, 2008). Circulated as [Dijkstra, 1970], and published with additional material as [Dijkstra, 1972].

[Dijkstra, 1969b] Dijkstra, E. W. (1969b). Structured programming. In Buxton, J. N. and Randell, B., editors, *Software Engineering Techniques: Report on a conference sponsored by the NATO Science Committee. Rome, Italy, 27th to 31st October 1969*, pages 84–88. NATO Science Committee.

[Dijkstra, 1970] Dijkstra, E. W. (1970). Notes on structured programming (second edition). Technical Report 70-WSK-03, Department of Mathematics, Technical University Eindhoven, The Netherlands.

[Dijkstra, 1972] Dijkstra, E. W. (1972). Notes on structured programming. In [Dahl et al., 1972], pages 1–82.

[Donaldson, 1973] Donaldson, J. R. (1973). Structured programming. *Datamation*, 19(12):52–54.

[Earley, 1971] Earley, J. (1971). Toward an understanding of data structure. *Communications of the ACM*, 14(10):617–627.

[Eckert et al., 1945] Eckert, J. P., Mauchly, J. W., and Warren, S. R. (1945). PY summary report No. 1. March 31, 1945. Quoted in [Aspray, 1990b], p. 38.

[Eckert, 1944] Eckert, Jr., J. P. (1944). Disclosure of magnetic calculating machine. Typescript dated 29 January, 1944. Reprinted in [Lukoff, 1979], pages 207–209.

[Eckert, 1946] Eckert, Jr., J. P. (1946). A preview of a digital computing machine. In [Campbell-Kelly and Williams, 1985], pages 109–126. Lecture delivered 15 July, 1946.

[Edgerton, 2006] Edgerton, D. (2006). *The Shock of the Old: Technology and global history since 1900*. Profile Books.

[Edwards, 1996] Edwards, P. N. (1996). *The Closed World: Computers and the Politics of Discourse in Cold War America*. The MIT Press.

[Elgot, 1954] Elgot, C. C. (1954). On single vs. triple address computing machines. *Journal of the Association for Computing Machinery*, 1(3):119–123.

[Elgot and Robinson, 1964] Elgot, C. C. and Robinson, A. (1964). Random-access stored-program machines, an approach to programming languages. *Journal of the Association for Computing Machinery*, 11(4):365–399.

[Feldman, 1966] Feldman, J. A. (1966). A formal semantics for computer languages and its application in a compiler-compiler. *Communications of the ACM*, 9(1):3–9.

[Felton, 1960] Felton, G. E. (1960). Assembly, interpretive and conversion programs for PEGASUS. In [Goodman, 1960], pages 32–57.

[Fetzer, 1988] Fetzer, J. H. (1988). Program verfication: the very idea. *Communications of the ACM*, 31(9):1048–1063.

[Floyd, 1964] Floyd, R. W. (1964). The syntax of programming languages—a survey. *IEEE Transactions on Electronic Computers*, EC-13(4):346–353.

[Floyd, 1967] Floyd, R. W. (1967). Assigning meanings to programs. In Schwartz, J. T., editor, *Mathematical Aspects of Computer Science*, volume XIX of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society.

[Floyd, 1971] Floyd, R. W. (1971). Toward interactive design of correct programs. In [Freiman, 1972], pages 7–10.

[Forrester and Hamming, 1952] Forrester, J. W. and Hamming, R. W., editors (1952). *Proceedings of the 1952 ACM national meeting (Toronto)*.

[Frege, 1879] Frege, G. (1879). *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Halle. Translated into English as "*Begriffsschrift*, a formula language, modeled upon that of arithmetic, for pure thought" in [van Heijenoort, 1967], pages 1–82.

[Freiman, 1972] Freiman, C. V., editor (1972). *Information Processing 71: Proceedings of the IFIP Congress 71*. North-Holland Publishing Company.

[Fritz, 1994] Fritz, W. B. (1994). ENIAC—a problem solver. *IEEE Annals of the History of Computing*, 16(1):25–45.

[Gandy, 1988] Gandy, R. (1988). The confluence of ideas in 1936. In [Herken, 1988], pages 55–111.

[Garwick, 1964] Garwick, J. V. (1964). The definition of programming languages by their compilers. In [Steel, 1966], pages 139–147.

[Gerhart and Yelowitz, 1976] Gerhart, S. L. and Yelowitz, L. (1976). Observations of fallibility in applications of modern programming methodologies. *IEEE Transactions on Software Engineering*, SE-2(3):195–207.

[Gill, 1951] Gill, S. (1951). The diagnosis of mistakes in programmes on the EDSAC. *Proceedings of the Royal Society of London (Series A)*, 206:538–554.

[Gill, 1953] Gill, S. (1953). Getting programmes right. In *Automatic Digital Computation*, pages 289–292. National Physical Laboratory, HMSO, London. Reprinted in [Williams and Campbell-Kelly, 1989], pages 209–498.

[Gill, 1959] Gill, S. (1959). Current theory and practice of automatic programming. *The Computer Journal*, 2(3).

[Gillies, 1992] Gillies, D., editor (1992). *Revolutions in Mathematics*. Oxford University Press.

[Gillies and Zheng, 2001] Gillies, D. and Zheng, Y. (2001). Dynamic interactions with the philosophy of mathematics. *Theoria*, 16(3):437–459.

[Gilmore, 1963] Gilmore, P. C. (1963). An abstract computer with a Lisp-like machine language without a label operator. In [Braffort and Hirschberg, 1963], pages 71–86.

[Giloi, 1997] Giloi, W. K. (1997). Konrad Zuse's Plankalkül: The first high-level, "non von Neumann" programming language. *IEEE Annals of the History of Computing*, 19(2):17–24.

[Gödel, 1931] Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198. English translation in [Gödel, 1986], pages 145–195. Page references to this translation.

[Gödel, 1934] Gödel, K. (1934). On undecidable propositions of formal mathematical systems. Mimeographed lecture notes, taken by Stephen C. Kleene and J. Barkley Rosser. Reprinted in [Gödel, 1986], pages 346–369.

[Gödel, 1986] Gödel, K. (1986). *Collected Works, Volume I. Publications 1929–1936* (S. Feferman et al., eds). Oxford University Press.

[Goldberg and Kay, 1976] Goldberg, A. and Kay, A. (1976). Smalltalk-72 instruction manual. Technical Report SSL 76-6, Xerox Palo Alto Research Center.

[Goldstine, 1972] Goldstine, H. H. (1972). *The Computer from Pascal to von Neumann*. Princeton University Press.

[Goldstine and Goldstine, 1946] Goldstine, H. H. and Goldstine, A. (1946). The Electronic Numerical Integrator and Computer (ENIAC). *Mathematical Tables and other Aids to Computation*, II(15):97–110.

[Goldstine and von Neumann, 1946] Goldstine, H. H. and von Neumann, J. (1946). On the principles of large scale computing machines. Unpublished. Reproduced in [Aspray and Burks, 1987], pages 317–348.

[Goldstine and von Neumann, 1947] Goldstine, H. H. and von Neumann, J. (1947). Planning and coding problems for an electronic computing instrument, Part II, Volume 1. Technical report, Institute of Advanced Study. Reprinted in [Aspray and Burks, 1987], pages 151–222.

[Goldstine and von Neumann, 1948] Goldstine, H. H. and von Neumann, J. (1948). Planning and coding problems for an electronic computing instrument, Part II, Volume 3. Technical report, Institute of Advanced Study. Reprinted in [Aspray and Burks, 1987], pages 286–306.

[Good, 1951] Good, I. J. (1951). Discussion contribution. In [Manchester University, 1951], page 192.

[Goodenough and Gerhart, 1975] Goodenough, J. B. and Gerhart, S. L. (1975). Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173.

[Goodman, 1960] Goodman, R., editor (1960). *Annual Review in Automatic Programming, I. Papers read at the Working Conference on Automatic Programming of Digital Computers held at Brighton, 1–3 April 1959*. Pergamon Press.

[Gordon, 1961] Gordon, G. (1961). A general purpose systems simulation program. In *Proceedings of the Eastern Joint Computer Conference*, pages 87–104.

[Gorn, 1961] Gorn, S. (1961). Some basic terminology connected with mechanical languages and their processors. *Communications of the ACM*, 4(8):336–339.

[Gorn, 1962] Gorn, S. (1962). Theory of mechanical languages. *Communications of the ACM*, 5(1):62.

[Gorn, 1964] Gorn, S. (1964). Summary remarks and general discussion, ACM Mechanical Languages Workshop, August 1963. *Communications of the ACM*, 7(2):133–136.

[Green et al., 1959] Green, J., Shapiro, R. M., Helt, Jr., F. R., Franciotti, R. G., and Theil, E. H. (1959). Remarks on ALGOL and symbol manipulation. *Communications of the ACM*, 2(9):25–27.

[Hamblin, 1957] Hamblin, C. L. (1957). Computer languages. *The Australian Journal of Science*, 20(5):135–139.

[Hamblin, 1958] Hamblin, C. L. (1958). GEORGE IA and II: A semi-translation programming scheme for DEUCE. Programming and operation manual. Unpublished report, University of New South Wales.

[Hartree, 1946a] Hartree, D. R. (1946a). Letter to the *Times*.

[Hartree, 1946b] Hartree, D. R. (1946b). The ENIAC, an electronic computing machine. *Nature*, 158(4015):500–506.

[Hartree, 1949] Hartree, D. R. (1949). *Calculating Instruments and Machines*. The University of Illinois Press.

[Harvard, 1947] Harvard (1947). *Proceedings of a Symposium on Large-scale Digital Calculating Machinery, 7–10 January 1947*, volume XVI of *The Annals of the Computation Laboratory of Harvard University*. The Navy Department Bureau of Ordnance and Harvard University, Harvard University Press.

[Harvard, 1951] Harvard (1951). *Proceedings of a Second Symposium on Large-scale Digital Calculation Machinery, 13–16 September 1949*, volume XXVI of *The Annals of the Computation Laboratory of Harvard University*. The Navy Department Bureau of Ordnance and Harvard University, Harvard University Press.

[Hashagen et al., 2002] Hashagen, U., Keil-Slawik, R., and Norberg, A. L., editors (2002). *History of Computing: Software Issues*. Springer.

[Heims, 1980] Heims, S. J. (1980). *John von Neumann and Norbert Wiener: From Mathematics to the Technologies of Life and Death*. The MIT Press.

[Henderson and Snowdon, 1972] Henderson, P. and Snowdon, R. (1972). An experiment in structured programming. *BIT*, 12:38–53.

[Herken, 1988] Herken, R., editor (1988). *The Universal Turing Machine: A Half-Century Survey*. Oxford University Press.

[Hilbert, 1926] Hilbert, D. (1926). Über das Unendliche. *Mathmatische Annalen*, 95:161–190. Translated as "On the infinite" in [van Heijenoort, 1967], pages 367–392.

[Hilbert and Ackermann, 1928] Hilbert, D. and Ackermann, W. (1928). *Grundzüge der theoretischen Logik*. Springer-Verlag.

[Hoare, 1968] Hoare, C. A. R. (1968). Record handling. In Genuys, F., editor, *Programming Languages*, pages 291–347. Academic Press.

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

[Hoare, 1971] Hoare, C. A. R. (1971). Proof of a program: FIND. *Communications of the ACM*, 14(1):39–45.

[Hoare, 1972a] Hoare, C. A. R. (1972a). A note on the for statement. *BIT*, 12:334–341.

[Hoare, 1972b] Hoare, C. A. R. (1972b). Notes on data structuring. In [Dahl et al., 1972], pages 83–173.

[Hoare, 1982] Hoare, C. A. R. (1982). Programming is an engineering profession. Technical Report PRG-27, Programming Research Group, Oxford University.

[Hoare, 1996] Hoare, C. A. R. (1996). How did software get so reliable without proof? In Gaudel, M.-C. and Woodcock, J., editors, *FME'96: Industrial Benefit and Advances in Formal Methods. Proceedings of the Third International Symposium of Formal Methods Europe*, volume 1051 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag.

[Hobbes, 1651] Hobbes, T. (1651). *Leviathan*. Andrew Crooke, London.

[Hodges, 1983] Hodges, A. (1983). *Alan Turing: The Engima*. Burnett Books with Hutchinson.

[Holmevik, 1994] Holmevik, J. R. (1994). Educating the machine: A study in the history of computing and the construction of the SIMULA programming language. Technical Report 22, University of Trondheim, Centre for Technology and Society.

[Hopper, 1952] Hopper, G. M. (1952). The education of a computer. In [ACM, 1952], pages 243–249.

[Hopper, 1959] Hopper, G. M. (1959). Automatic programming: present status and future trends. In [National Physical Laboratory, 1959], pages 155–194.

[Huskey, 1948] Huskey, H. D. (1948). The status of high-speed digital computing systems. *Mechanical Engineering*, 70(12):975–978.

[Huskey, 1951] Huskey, H. D. (1951). Semiautomatic instruction on the Zephyr. In [Harvard, 1951], pages 83–90.

[Hyman, 1990] Hyman, R. A. (1990). Whiggism in the history of science and the study of the life and work of Charles Babbage. *Annals of the History of Computing*, 12(1):62–67.

[IBM, 1956] IBM (1956). *Programmer's Reference Manual: The FORTRAN Automatic Coding System for the IBM 704 EDPM*. International Business Machines Corporation, 590 Madison Ave., New York 22, N.Y. Applied Science Division and Programming Research Dept., Working Committee: J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, H. L. Herrick, R. A. Hughes, L. B. Mitchell, R. A. Nelson, R. Nutt, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller.

[IBM, 1958] IBM (1958). *Reference Manual: FORTRAN II for the IBM 704 Data Processing System*. International Business Machines Corporation, 590 Madison Ave., New York 22, N.Y.

[Ingalls, 1978] Ingalls, D. H. H. (1978). The Smalltalk-76 system design and implementation. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 9–16.

[International Computation Center, 1962] International Computation Center (1962). *Symbolic Languages in Data Processing: Proceedings of the Symposium organized and edited by the International Computation Centre, Rome, March 26-31, 1962*. Gordon and Breach.

[Irons, 1961] Irons, E. T. (1961). A syntax directed compiler for ALGOL 60. *Communications of the ACM*, 4(1):51–55.

[Isaac, 1952] Isaac, E. J. (1952). Machine aids to coding. In [Forrester and Hamming, 1952], pages 17–19.

[Jeffress, 1951] Jeffress, L., editor (1951). *Cerebral Mechanisms in Behavior*. Wiley-Interscience.

[Jones, 2003] Jones, C. B. (2003). The early search for tractable ways of reasoning about programs. *IEEE Annals of the History of Computing*, 25(2):26–49.

[Kalenich, 1965] Kalenich, W. A., editor (1965). *Information Processing 1965: Proceedings of IFIP Congress 65*. Spartan Books, Inc.

[Katz, 1957] Katz, C. (1957). Systems of debugging automatic coding. In *Automatic Coding: Proceedings of the Symposium held January 24-25, 1957 at the Franklin Institute in Philadelphia*, Journal of the Franklin Institute, Monograph No. 3, pages 17–27.

[Kay and Goldberg, 1977] Kay, A. and Goldberg, A. (1977). Personal dynamic media. *Computer*, 10(3):31–41.

[Kay, 1996] Kay, A. C. (1996). The early history of Smalltalk. In [Bergin and Gibson, 1996], pages 511–579.

[Kleene, 1935a] Kleene, S. C. (1935a). A theory of positive integers in formal logic. Part I. *American Journal of Mathematics*, 57(1):153–173.

[Kleene, 1935b] Kleene, S. C. (1935b). A theory of positive integers in formal logic. Part II. *American Journal of Mathematics*, 57(2):219–244.

[Kleene, 1936a] Kleene, S. C. (1936a). General recursive functions of natural numbers. *Mathematische Annalen*, 112(5):727–742.

[Kleene, 1936b] Kleene, S. C. (1936b). λ-definability and recursiveness. *Duke Mathematical Journal*, 2(2):340–353.

[Knuth, 1970] Knuth, D. E. (1970). Von Neumann's first computer program. *Computing Surveys*, 2(4):247–260.

[Knuth, 1974] Knuth, D. E. (1974). Structured programming with go to statements. *Computing Surveys*, 6(4):260–301.

[Knuth and McNeley, 1964] Knuth, D. E. and McNeley, J. L. (1964). SOL—a symbolic language for general-purpose systems simulation. *IEEE Transactions on Electronic Computers*, EC–13(4):401–408.

[Knuth and Trabb Pardo, 1980] Knuth, D. E. and Trabb Pardo, L. (1980). The early development of programming languages. In [Metropolis et al., 1980].

[Kuhn, 1962] Kuhn, T. S. (1962). *The Structure of Scientific Revolutions*. The University of Chicago Press.

[Kuhn, 1968] Kuhn, T. S. (1968). The history of science. In Sills, D. L., editor, *International Encyclopedia of the Social Sciences*, volume 14, pages 74–83. The Macmillan Company & The Free Press.

[Lakatos, 1967] Lakatos, I., editor (1967). *Problems in the Philosophy of Mathematics*. Amsterdam: North-Holland.

[Lakatos, 1970] Lakatos, I. (1970). Falsification and the methodology of scientific research programmes. In Lakatos, I. and Musgrave, A., editors, *Criticism and the Growth of Knowledge*, pages 91–196. Cambridge University Press.

[Lakatos, 1971] Lakatos, I. (1971). History of science and its rational reconstructions. In Buck, R. C. and Cohen, R. S., editors, *P. S. A. 1970 Boston Studies in the Philosophy of Science*, volume 8, pages 91–135. Dordrecht: Reidel. Reprinted as chapter 2 of [Lakatos, 1978].

[Lakatos, 1978] Lakatos, I. (1978). *Philosophical papers. Vol. I: The methodology of scientific research programmes*. Cambridge University Press.

[Landin, 1964] Landin, P. J. (1964). The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320.

[Landin, 1965a] Landin, P. J. (1965a). A correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Communications of the ACM*, 8(2):89–101.

[Landin, 1965b] Landin, P. J. (1965b). A correspondence between ALGOL 60 and Church's lambda-notation: Part II. *Communications of the ACM*, 8(3):158–165.

[Landin, 1966] Landin, P. J. (1966). The next 700 programming languages. *Communications of the ACM*, 9(3):157–166.

[Laning Jr. and Zierler, 1954] Laning Jr., J. H. and Zierler, N. (1954). A program for translation of mathematical equations with Whirlwind I. Engineering Memorandum E-364, Instrumentation Laboratory, Massachusetts Institute of Technology.

[Larman and Basili, 2003] Larman, C. and Basili, V. R. (2003). Iterative and incremental development: A brief history. *Computer*, 36(6):47–56.

[Larvor, 1998] Larvor, B. (1998). *Lakatos: An Introduction*. Routledge: London and New York.

[Ledgard, 1974] Ledgard, H. F. (1974). The case for structured programming. *BIT*, 14:45–57.

[Lewis, 1918] Lewis, C. I. (1918). *A Survey of Symbolic Logic*. University of California Press.

[Liskov, 1973] Liskov, B. (1973). Report of session on structured programming. *ACM SIGPLAN Notices*, 8(9):5–10.

[Liskov and Zilles, 1974] Liskov, B. and Zilles, S. (1974). Programming with abstract data types. *ACM SIGPLAN Notices*, 9(4):50–59.

[Liskov and Zilles, 1975] Liskov, B. H. and Zilles, S. N. (1975). Specification techniques for data abstractions. *IEEE Transactions on Software Engineering*, SE-1(1):7–19.

[Lombardi, 1960] Lombardi, L. (1960). Theory of files. In *Proceedings of the Eastern Joint Computer Conference*, pages 137–141.

[Lucas, 1972] Lucas, P. (1972). Formal definition of programming languages and systems. In [Freiman, 1972], pages 291–297.

[Lucas and Walk, 1969] Lucas, P. and Walk, K. (1969). On the formal description of PL/I. *Annual Review in Automatic Programming*, 6(3):105–182.

[Lukoff, 1979] Lukoff, H. (1979). *From Dits to Bits: A personal history of the electronic computer*. Robotics Press, Portland, Oregon.

[Mahoney, 1988] Mahoney, M. S. (1988). The history of computing in the history of technology. *Annals of the History of Computing*, 10:113–125.

[Mahoney, 1989] Mahoney, M. S. (1989). Cybernetics and information technology. In Olby, R. C., editor, *Companion to the History of Modern Science*, chapter 34. Routledge, Chapman and Hall.

[Mahoney, 1997] Mahoney, M. S. (1997). Computer science: The search for a mathematical theory. In Krige, J. and Pestre, D., editors, *Science in the Twentieth Century*, pages 617–634. Harwood Academic Publishers.

[Mahoney, 2000] Mahoney, M. S. (2000). Software as science—science as software. In [Hashagen et al., 2002], pages 25–48.

[Manchester University, 1951] Manchester University (1951). *Manchester University Computer, Inaugural Conference, July 1951*. Reprinted in [Williams and Campbell-Kelly, 1989], pages 165–206.

[Manna and Waldinger, 1971] Manna, Z. and Waldinger, R. J. (1971). Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165.

[Manna and Waldinger, 1978] Manna, Z. and Waldinger, R. J. (1978). The logic of computer programming. *IEEE Transactions on Software Engineering*, SE-4(3):199–229.

[Marcus and Akera, 1996] Marcus, M. and Akera, A. (1996). Exploring the architecture of an early machine: the historical relevance of the ENIAC machine architecture. *IEEE Annals of the History of Computing*, 18(1):17–24.

[Markowitz et al., 1963] Markowitz, H. M., Hauser, B., and Kerr, H. W. (1963). *SIMSCRIPT—A Simulation Programing Language*. Prentice-Hall, Inc.

[Martin, 1993] Martin, C. D. (1993). The myth of the awesome thinking machine. *Communications of the ACM*, 36(4):120–133.

[Masani et al., 1987] Masani, N., Randell, B., Ferry, D. K., and Saeks, R. (1987). The Wiener memorandum on the mechanical solution of partial differential equations. *Annals of the History of Computing*, 9(2):183–197.

[Mauchly, 1942] Mauchly, J. W. (1942). The use of high speed vacuum tubes for calculating. Unpublished memorandum, quoted in [Stern, 1981], page 56.

[Mauchly, 1946] Mauchly, J. W. (1946). Code and control II: Machine design and instruction codes. In [Campbell-Kelly and Williams, 1985], pages 453–461. Lecture delivered 9 August, 1946.

[Mauchly, 1947] Mauchly, J. W. (1947). Preparation of problems for EDVAC-type machines. In [Harvard, 1947], pages 203–207.

[Mauchly, 1949] Mauchly, J. W. (1949). Suggested form for “BINAC BRIEF CODE”. Unpublished notes, printed in [Schmitt, 1988], pages 17–18.

[McCarthy, 1960] McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195.

[McCarthy, 1961] McCarthy, J. (1961). A basis for a mathematical theory of computation, preliminary report. In *Proceedings of the Western Joint Computer Conference*, pages 225–238.

[McCarthy, 1962] McCarthy, J. (1962). Towards a mathematical science of computation. In [Popplewell, 1963], pages 21–28.

[McCarthy, 1963a] McCarthy, J. (1963a). A basis for a mathematical theory of computation. In [Braffort and Hirschberg, 1963], pages 33–70. Corrected and extended version of [McCarthy, 1961].

[McCarthy, 1963b] McCarthy, J. (1963b). General discussion. Quoted in [Gorn, 1964].

[McCarthy, 1964] McCarthy, J. (1964). A formal description of a subset of ALGOL. In [Steel, 1966], pages 1–12.

[McCarthy, 1965] McCarthy, J. (1965). Problems in the theory of computation. In [Kalenich, 1965], pages 219–222.

[McCarthy, 1981] McCarthy, J. (1981). History of LISP. In [Wexelblat, 1981], pages 173–185.

[McCracken, 1973] McCracken, D. (1973). Revolution in programming: An overview. *Datamation*, 19(12):50–52. Reprinted in [Yourdon, 1979].

[McCracken and Jackson, 1982] McCracken, D. D. and Jackson, M. A. (1982). Life cycle concept considered harmful. *ACM SIGSOFT Software Engineering Notes*, 7(2):29–32.

[McCulloch, 1948] McCulloch, W. S. (1948). Contribution to discussion following [von Neumann, 1948]. In [Jeffress, 1951], pages 32–41.

[McCulloch and Pitts, 1943] McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133.

[Metropolis et al., 1980] Metropolis, N., Howlett, J., and Rota, G.-C., editors (1980). *A History of Computing in the Twentieth Century*. Academic Press.

[Metropolis and Worlton, 1980] Metropolis, N. and Worlton, J. (1980). A trilogy of errors in the history of computing. *Annals of the History of Computing*.

[Miller, 1949] Miller, J. C. P. (1949). Remarks on checking. In [Cambridge University Mathematical Laboratory, 1950], pages 123–124.

[Mills, 1976] Mills, H. D. (1976). Software development. *IEEE Transactions on Software Engineering*, SE-2(4):265–273.

[Mills, 1986] Mills, H. D. (1986). Structured programming: Retrospect and prospect. *IEEE Software*, 3(6):58–66.

[Ministry of Supply, 1950] Ministry of Supply (1950). *Symposium on Information Theory: Report of Proceedings*. Ministry of Supply, London.

[Misa, 2007] Misa, T. J. (2007). Understanding 'how computing has changed the world'. *IEEE Annals of the History of Computing*, 29(4):52–63.

[Mooers, 1946] Mooers, C. N. (1946). Code and control IV: Example of a three-address code and the use of "stop order tags". In [Campbell-Kelly and Williams, 1985], pages 465–484. Lecture delivered 12 August, 1946.

[Morris, 1938] Morris, C. W. (1938). *Foundations of the Theory of Signs*, volume I, number 2 of *International Encyclopedia of Unified Science*. The University of Chicago Press.

[Morris Jr., 1973] Morris Jr., J. H. (1973). Protection in programming languages. *Communications of the ACM*, 16(1):15–21.

[National Physical Laboratory, 1959] National Physical Laboratory (1959). *Mechanisation of Thought Processes: Proceedings of a Symposium held at the National Physical Laboratory on 24th, 25th, 26th and 27th November 1958*. National Physical Laboratory, HMSO, London.

[Naur, 1959] Naur, P., editor (1959). *ALGOL-Bulletin no. 1*. Regnecentralen, Copenhagen.

[Naur, 1966] Naur, P. (1966). Proof of algorithms by general snapshots. *BIT*, 6:310–316.

[Naur, 1969] Naur, P. (1969). Programming by action clusters. *BIT*, 9:250–258.

[Naur, 1981] Naur, P. (1981). The European side of the last phase of the development of ALGOL 60. In [Wexelblat, 1981], pages 92–139.

[Naur et al., 1960] Naur, P., Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., van Wijngaarden, A., and Woodger, M. (1960). Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314.

[Naur and Randell, 1969] Naur, P. and Randell, B., editors (1969). *Software Engineering: Report on a conference sponsored by the NATO Science Committee. Garmisch, Germany, 7th to 11th October 1968*. Scientific Affairs Division, NATO, Brussels 39, Belgium.

[Newell and Simon, 1956] Newell, A. and Simon, H. A. (1956). The logic theory machine: A complex information processing system. *IRE Transactions on Information Theory*, IT-2(3):61–79.

[Newell and Tonge, 1960] Newell, A. and Tonge, F. M. (1960). An introduction to Information Processing Language V. *Communications of the ACM*, 3(4):205–211.

[Newman, 1949] Newman, M. H. A. (1949). General principles of the design of all-purpose computing machines. *Proceedings of the Royal Society of London (Series A)*, 195:271–274. Record of a discussion held on March 4, 1948.

[Nygaard and Dahl, 1981] Nygaard, K. and Dahl, O.-J. (1981). The development of the SIMULA languages. In [Wexelblat, 1981], pages 439–480.

[Oettinger, 1952] Oettinger, A. G. (1952). Programming a digital computer to learn. *The Philosophical Magazine*, 43, Seventh Series:1243–1263.

[Parnas, 1972] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058.

[Patterson, 1949] Patterson, G. W. (1949). Logical syntax and transformation rules. In [Harvard, 1951], pages 125–133.

[Peláez, 1999] Peláez, E. (1999). The stored-program computer: Two conceptions. *Social Studies of Science*, 29(3):359–389.

[Perlis and Samelson, 1958] Perlis, A. J. and Samelson, K. (1958). Preliminary report—International Algebraic Language. *Communications of the ACM*, 1(12):8–22.

[Pickering, 1995] Pickering, A. (1995). *The Mangle of Practice: Time, Agency and Science*. The University of Chicago Press.

[Pickering, 2002] Pickering, A. (2002). Cybernetics and the mangle: Ashby, Beer and Pask. *Social Studies of Science*, 32(3):413–437.

[Popplewell, 1963] Popplewell, C. M., editor (1963). *Information Processing 1962: Proceedings of IFIP Congress 62*. North-Holland Publishing Company.

[Post, 1936] Post, E. L. (1936). Finite combinatory processes—formulation 1. *The Journal of Symbolic Logic*, 1(3):103–105.

[Pratt, 1987] Pratt, V. (1987). *Thinking Machines: the Evolution of Artificial Intelligence*. Basil Blackwell.

[Prinz, 1952] Prinz, D. G. (1952). Robot chess. *Research: Science and its Application in Industry*, 5:261–266.

[Puyen and Vauquois, 1960] Puyen, J. and Vauquois, B. (1960). A propos d'un langage universel. In [Unesco, 1960], pages 132–137.

[Rabinowitz, 1962] Rabinowitz, I. N. (1962). Report on the algorithmic language FORTRAN II. *Communications of the ACM*, 5(6):327–337.

[Radin and Rogoway, 1965] Radin, G. and Rogoway, H. P. (1965). NPL: Highlights of a new programming language. *Communications of the ACM*, 8(1):9–17.

[Randell, 1972] Randell, B. (1972). On Alan Turing and the origins of digital computers. In Meltzer, B. and Michie, D., editors, *Machine Intelligence 7*, pages 3–20. Edinburgh University Press.

[Randell, 2000] Randell, B. (2000). Facing up to faults. *The Computer Journal*, 43(2):95–106.

[Redmond and Smith, 1980] Redmond, K. C. and Smith, T. M. (1980). *Project Whirlwind: The History of a Pioneer Computer*. Digital Press.

[Renwick, 1949] Renwick, W. (1949). The E.D.S.A.C. demonstration. In [Cambridge University Mathematical Laboratory, 1950], pages 21–26.

[Robinson, 1960] Robinson, C. (1960). Automatic programming on DEUCE. In [Goodman, 1960], pages 111–126.

[Robinson et al., 1998] Robinson, H., Hall, P., Hovenden, F., and Rachel, J. (1998). Postmodern software development. *The Computer Journal*, 41(6):363–375.

[Rochester and Goldfinger, 1964] Rochester, N. and Goldfinger, R. (1964). The special issue on computer languages—Editorial. *IEEE Transactions on Electronic Computers*, EC-13(4):343.

[Rope, 2007] Rope, C. (2007). ENIAC as a stored-program computer: A new look at the old records. *IEEE Annals of the History of Computing*, 29(4):82–87.

[Rosen, 1964] Rosen, S. (1964). Programming systems and languages: A historical survey. In *Proceedings of the 1964 Spring Joint Computer Conference*, volume 25 of *AFIPS Conference Proceedings*, pages 1–15. Spartan Books, Inc., Baltimore, Md., Cleaver-Hume Press, London.

[Rosen, 1967] Rosen, S. (1967). *Programming Systems and Languages*. McGraw-Hill, Inc.

[Rosen, 1972] Rosen, S. (1972). Programming systems and languages 1965–1975. *Communications of the ACM*, 15(7):591–600.

[Rosenbloom, 1950] Rosenbloom, P. C. (1950). *The Elements of Mathematical Logic*. Dover Publications, Inc.

[Rosenblueth et al., 1943] Rosenblueth, A., Wiener, N., and Bigelow, J. (1943). Behaviour, purpose and teleology. *Philosophy of Science*, 10(1):18–24.

[Ross, 1961] Ross, D. T. (1961). A generalized technique for symbol manipulation and numerical calculation. *Communications of the ACM*, 4(3):147–150.

[Ross and Rodriguez, 1963] Ross, D. T. and Rodriguez, J. E. (1963). Theoretical foundations for the computer-aided design system. In *Proceedings of the 1963 Spring Joint Computer Conference*, volume 23 of *AFIPS Conference Proceedings*, pages 305–322. Spartan Books, Inc., Baltimore, Md., Cleaver-Hume Press, London.

[Rosser, 1984] Rosser, J. B. (1984). Highlights of the history of the lambda-calculus. *Annals of the History of Computing*, 6(4):337–349.

[Ryder and Hailpern, 2007] Ryder, B. and Hailpern, B., editors (2007). *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. ACM Press, New York.

[Samelson, 1981] Samelson, K. (1981). Appendix 7 to [Naur, 1981]: Comments by K. Samelson, 1978 December 1. In [Wexelblat, 1981], pages 131–134.

[Sammet, 1962] Sammet, J. E. (1962). Basic elements of COBOL 61. *Communications of the ACM*, 5(5):237–253.

[Sammet, 1969] Sammet, J. E. (1969). *Programming Languages: History and Fundamentals*. Prentice-Hall, Inc.

[Sammet, 1972] Sammet, J. E. (1972). Programming languages: History and future. *Communications of the ACM*, 15(7):601–610.

[Sammet, 1981] Sammet, J. E. (1981). Organization of the conference. In [Wexelblat, 1981], pages xvii–xx. From the ACM SIGPLAN History of Programming Languages Conference, June 1–3, 1978.

[Schick and Wolverton, 1978] Schick, G. J. and Wolverton, R. W. (1978). An analysis of competing software reliability models. *IEEE Transactions on Software Engineering*, SE-4(2):104–120.

[Schmitt, 1988] Schmitt, W. F. (1988). The UNIVAC SHORT CODE. *Annals of the History of Computing*, 10(1):7–18.

[Schönfinkel, 1924] Schönfinkel, M. (1924). Über die Bausteine der mathematischen Logik. *Mathematische Annalen*. Translated as “On the building blocks of mathematical logic” in [van Heijenoort, 1967], pages 357–366.

[Shannon, 1938] Shannon, C. E. (1938). A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers*, 57:713–723.

[Shannon, 1950] Shannon, C. E. (1950). Programming a computer for playing chess. *The Philosophical Magazine*, 41, Seventh Series:256–275. Paper first presented at National IRE Convention, March 9, 1949.

[Shannon, 1953] Shannon, C. E. (1953). Computers and automata. *Proceedings of the I.R.E*, 41(10):1234–1241.

[Shapiro, 1997] Shapiro, S. (1997). Splitting the difference: The historical necessity of synthesis in software engineering. *IEEE Annals of the History of Computing*, 19(1):20–54.

[SHARE, 1958a] SHARE (1958a). News and notices. *Communications of the ACM*, 1(3):17.

[SHARE, 1958b] SHARE (1958b). News and notices. *Communications of the ACM*, 1(4):16–17.

[Shaw, 1990] Shaw, M. (1990). Prospects for an engineering discipline of software. *IEEE Software*, 7(6):15–24.

[Sheridan, 1957] Sheridan, P. B. (1957). The Fortran automatic coding system. In *Summer Institute for Symbolic Logic, Cornell University, 1957*, pages 452–3. American Mathematical Society.

[Sheridan, 1959] Sheridan, P. B. (1959). The arithmetic translator-compiler of the IBM FORTRAN automatic coding system. *Communications of the ACM*, 2(2):9–21.

[Shoch, 1979] Shoch, J. F. (1979). An overview of the programming language Smalltalk-72. *ACM SIGPLAN Notices*, 14(9):64–73.

[Skolem, 1923] Skolem, T. (1923). Begründung der elementaren Arithmetik durch die rekurrierende Denkweise ohne Anwendung scheinbarer Veränderlichen mit unendlichem Ausdehnungsbereich. *Skrifter utgit av Videnskapsselskapet i Kristiania, I. Matematisk-naturvidenskabelig klasse*, 6:1–38. Translated as “The foundations of elementary arithmetic established by means of the recursive mode of thought, without the use of apparent variables ranging over infinite domain” in [van Heijenoort, 1967], pages 302–333.

[Soare, 1996] Soare, R. I. (1996). Computability and recursion. *The Bulletin of Symbolic Logic*, 2(3):284–321.

[Steel, 1966] Steel, T. B., editor (1966). *Formal Language Description Languages for Computer Programming*. North-Holland Publishing Company.

[Steel, 1961] Steel, Jr., T. B. (1961). UNCOL: The myth and the fact. In Goodman, R., editor, *Annual Review in Automatic Programming, 2*, pages 325–344. Pergamon Press.

[Stern, 1980] Stern, N. (1980). John von Neumann's influence on electronic digital computing, 1944–1946. *Annals of the History of Computing*, 2(4):349–362.

[Stern, 1981] Stern, N. (1981). *From ENIAC to UNIVAC: An Appraisal of the Eckert-Mauchly Computers*. Digital Press.

[Stevens et al., 1974] Stevens, W., Myers, G., and Constantine, L. (1974). Structured design. *IBM Systems Journal*, 13(2):115–139.

[Strachey, 1962] Strachey, C. (1962). Contribution to panel discussion on metasyntactic and metasemantic languages. In [International Computation Center, 1962], pages 99–110.

[Strachey, 1964] Strachey, C. (1964). Towards a formal semantics. In [Steel, 1966], pages 198–220.

[Strachey, 1967] Strachey, C. (1967). Fundamental concepts in programming languages. Unpublished lecture notes, International Summer School in Computer Programming, Copenhagen. Published as [Strachey, 2000].

[Strachey, 2000] Strachey, C. (2000). Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13:11–49.

[Strachey and Wilkes, 1961] Strachey, C. and Wilkes, M. V. (1961). Some proposals for improving the efficiency of ALGOL 60. *Communications of the ACM*, 4(11):488–491.

[Stroustrup, 1994] Stroustrup, B. (1994). *The Design and Evolution of C++*. Addison-Wesley.

[Tarski, 1933] Tarski, A. (1933). Pojęcie prawdy w językach nauk dedukcyjnych. Warsaw. Translated with additions into German as [Tarski, 1935], and then into English as "The concept of truth in formalized languages" in [Tarski, 1983], pages 152–278.

[Tarski, 1935] Tarski, A. (1935). Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, I:261–405.

[Tarski, 1936a] Tarski, A. (1936a). Grundlegung der wissenschaftlichen Semantik. In *Actes du Congrès International de Philosophie Scientifique*, pages 1–8.

[Tarski, 1936b] Tarski, A. (1936b). O ugruntowaniu naukowej semantyki. *Przegląd Filozoficzny*, 39:50–57. Translated into German as [Tarski, 1936a], and then into English as "The establishment of scientific semantics" in [Tarski, 1983], pages 401–408.

[Tarski, 1983] Tarski, A. (1983). *Logic, Semantics, Metamathematics*. Hackett Publishing Company, second edition.

[Taylor, 1960] Taylor, A. (1960). The FLOW-MATIC and MATH-MATIC automatic programming systems. In [Goodman, 1960], pages 196–206.

[Tennent, 1976] Tennent, R. D. (1976). The denotational semantics of programming languages. *Communications of the ACM*, 19(8):437–543.

[Todd, 1974] Todd, J. (1974). John von Neumann and the National Accounting Machine. *SIAM Review*, 16(4):526–530.

[Turing, 1936] Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Second Series*, 42:230–265.

[Turing, 1946] Turing, A. M. (1946). Proposal for development in the mathematics department of an automatic computing engine (ACE). Technical report, National Physical Laboratory, Teddington, UK. Reprinted in [Carpenter and Doran, 1986], pages 20–105.

[Turing, 1947] Turing, A. M. (1947). Lecture to the London Mathematical Society on 20 February 1947. In [Carpenter and Doran, 1986], pages 106–124.

[Turing, 1948] Turing, A. M. (1948). Intelligent machinery. Technical report, National Physical Laboratory. Reprinted in [Copeland, 2004b].

[Turing, 1949] Turing, A. M. (1949). Checking a large routine. In [Cambridge University Mathematical Laboratory, 1950], pages 70–72.

[Turing, 1950a] Turing, A. M. (1950a). Computing machinery and intelligence. *Mind*, 59:433–60.

[Turing, 1950b] Turing, A. M. (1950b). Discussion on Dr. E. Slater's paper on "Statistics for the chess computer and the factor of mobility". In [Ministry of Supply, 1950], pages 198–200.

[Turing, 1951] Turing, A. M. (1951). Discussion contribution. In [Manchester University, 1951], page 192. Reprinted in [Williams and Campbell-Kelly, 1989], pages 165–206.

[Ulam, 1980] Ulam, S. M. (1980). Von Neumann: The interaction of mathematics and computing. In [Metropolis et al., 1980], pages 93–99.

[Unesco, 1960] Unesco (1960). *Proceedings of the International Conference on Information Processing, Unesco, Paris 15-20 June, 1959*. Unesco, Unesco (Paris), R. Oldenbourg (Munich) and Butterworths (London).

[van der Poel, 1986] van der Poel, W. L. (1986). Some notes on the history of ALGOL. In Zemanek, H., editor, *A Quarter Century of IFIP*, pages 373–392. Elsevier Science Publishers B.V.

[van Heijenoort, 1967] van Heijenoort, J. (1967). *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press.

[van Wijngaarden, 1962] van Wijngaarden, A. (1962). Generalized ALGOL. In [International Computation Center, 1962], pages 409–419.

[von Neumann, 1945] von Neumann, J. (1945). First draft of a report on the EDVAC. Technical report, Moore School of Electrical Engineering, Unversity of Pennsylvania. Reprinted as [von Neumann, 1993] with corrections by Michael D. Godfrey.

[von Neumann, 1948] von Neumann, J. (1948). The general and logical theory of automata. In [Jeffress, 1951], pages 1–41.

[von Neumann, 1993] von Neumann, J. (1993). First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75.

[Wang, 1957] Wang, H. (1957). A variant to Turing's theory of computable numbers. *Journal of the Association for Computing Machinery*, 4(1):63–92.

[Wegner, 1972] Wegner, P. (1972). Operational semantics of programming languages. *ACM SIGPLAN Notices*, 7(1):128–141.

[Wegner, 1976] Wegner, P. (1976). Programming languages—the first 25 years. *IEEE Transactions on Computers*, C-25(12):1207–1225.

[Wegstein, 1960] Wegstein, J. H. (1960). Algorithms: Announcement. *Communications of the ACM*, 3(2):73.

[Weiss, 1993] Weiss, E. A. (1993). Obituaries: Peter Bernard Sheridan. *IEEE Annals of the History of Computing*, 15(1):69–70.

[Wells, 1980] Wells, M. B. (1980). Reflections on the evolution of algorithmic language. In [Metropolis et al., 1980].

[Wexelblat, 1981] Wexelblat, R. L., editor (1981). *History of Programming Languages*. Academic Press. From the ACM SIGPLAN History of Programming Languages Conference, June 1–3, 1978.

[Wheeler, 1950] Wheeler, D. J. (1950). Programme organization and initial orders for the EDSAC. *Proceedings of the Royal Society of London (Series A)*, 202:573–589.

[Wheeler, 1952] Wheeler, D. J. (1952). The use of sub-routines in programmes. In [ACM, 1952], pages 235–236.

[Whitehead and Russell, 1910] Whitehead, A. N. and Russell, B. (1910). *Principia Mathematica*, volume I. Cambridge University Press.

[Wiener, 1940] Wiener, N. (1940). Memorandum on the mechanical solution of partial differential equations. Printed in [Masani et al., 1987].

[Wiener, 1945] Wiener, N. (1945). Letter to Arturo Rosenblueth, January 24, 1945. Quoted in [Heims, 1980], pages 185–6.

[Wiener, 1948] Wiener, N. (1948). *Cybernetics*. The Technology Press, John Wiley & Sons.

[Wilkes, 1949] Wilkes, M. V. (1949). The design of a practical high-speed computing machine. the EDSAC. *Proceedings of the Royal Society of London (Series A)*, 195:274–279. Record of a discussion held on March 4, 1948.

[Wilkes, 1951a] Wilkes, M. V. (1951a). Automatic calculating machines. *Journal of the Royal Society of Arts*, C(4862):56–90.

[Wilkes, 1951b] Wilkes, M. V. (1951b). Can machines think? *The Spectator*, 6424:177–178.

[Wilkes, 1952] Wilkes, M. V. (1952). Pure and applied programming. In [Forrester and Hamming, 1952], pages 121–124.

[Wilkes, 1953a] Wilkes, M. V. (1953a). Can machines think? *Proceedings of the I.R.E*, 41(10):1230–1234.

[Wilkes, 1953b] Wilkes, M. V. (1953b). The use of a 'floating address' system for orders in an automatic digital computer. *Proceedings of the Cambridge Philosophical Society*, 49(1):84–89.

[Wilkes et al., 1951] Wilkes, M. V., Wheeler, D. J., and Gill, S. (1951). *The Preparation of Programs for an Electronic Digital Computer*. Addison-Wesley Press, Inc.

[Wilkins, 1668] Wilkins, J. (1668). *An Essay Towards a Real Character and a Philosophical Language*. Sa: Gellibrand and John Martyn, for the Royal Society, London.

[Williams, 1951] Williams, F. C. (1951). The University of Manchester computing machine. In [Manchester University, 1951], pages 171–177.

[Williams and Campbell-Kelly, 1989] Williams, M. R. and Campbell-Kelly, M., editors (1989). *The Early British Computer Conferences*, volume 14 of *Charles Babbage Institute Reprint Series for the History of Computing*. The MIT Press.

[Wirth, 1971a] Wirth, N. (1971a). Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227.

[Wirth, 1971b] Wirth, N. (1971b). The programming language Pascal. *Acta Informatica*, 1:35–63.

[Wirth, 1974] Wirth, N. (1974). On the composition of well-structured programs. *Computing Surveys*, 6(4):247–259.

[Wirth and Hoare, 1966] Wirth, N. and Hoare, C. A. R. (1966). A contribution to the development of ALGOL. *Communications of the ACM*, 9(6):413–432.

[Wisdom, 1951] Wisdom, J. O. (1951). The hypothesis of cybernetics. *The British Journal for the Philosophy of Science*, II(5):1–24.

[Woodger, 1960] Woodger, M. (1960). An introduction to ALGOL 60. *The Computer Journal*, 3(2):67–75.

[Wulf et al., 1971] Wulf, W. A., Russell, D. B., and Haberman, A. N. (1971). BLISS: A language for systems programming. *Communications of the ACM*, 14(12):780–790.

[Yourdon, 1979] Yourdon, E. N., editor (1979). *Classics in Software Engineering*. Yourdon Press.

[Zemanek, 1966] Zemanek, H. (1966). Semiotics and programming languages. *Communications of the ACM*, 9(3):139–143.

[Zilles, 1973] Zilles, S. N. (1973). Procedural abstraction: A linguistic protection technique. *ACM SIGPLAN Notices*, 8(9):142–146.

[Zuse, 1948] Zuse, K. (1948). Über den Allgemeinen Plankalkül als Mittel zur Formalierung schematisch-kombinativer Aufgaben. *Archiv der Mathematik*, 1(6):441–449.

[Zuse, 1993] Zuse, K. (1993). *The Computer—My Life*. Springer-Verlag.