History of Computing

Mark Priestley

A Science of Operations

Machines, Logic and the Invention of Programming

Springer

History of Computing

The *History of Computing* series publishes high-quality books which address the history of computing, with an emphasis on the 'externalist' view of this history, more accessible to a wider audience. The series examines content and history from four main quadrants: the history of relevant technologies, the history of the core science, the history of relevant business and economic developments, and the history of computing as it pertains to social history and societal developments.

Titles can span a variety of product types, including but not exclusively, themed volumes, biographies, 'profile' books (with brief biographies of a number of key people), expansions of workshop proceedings, general readers, scholarly expositions, titles used as ancillary textbooks, revivals and new editions of previous worthy titles.

These books will appeal, varyingly, to academics and students in computer science, history, mathematics, business and technology studies. Some titles will also directly appeal to professionals and practitioners of different backgrounds.

Author guidelines: springer.com > Authors > Author Guidelines

For other titles published in this series, go to www.springer.com/series/8442

Mark Priestley

A Science of Operations

Machines, Logic and the Invention of Programming

Springer

Dr. Mark Priestley
London
UK
m.priestley@gmail.com
url: http://www.markpriestley.net

ISSN 2190-6831 e-ISSN 2190-684X
ISBN 978-1-84882-554-3 e-ISBN 978-1-84882-555-0
DOI 10.1007/978-1-84882-555-0
Springer London Dordrecht Heidelberg New York

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2011921403

Cover design: deblik

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

It has long been recognized that there is a relationship between logic and computer programming. Since the 1940s many logicians, including Alan Turing and John von Neumann, and computer scientists such as Edsgar Dijkstra and John McCarthy, have drawn attention to the connection between the two disciplines and commented on its importance. During the formal methods boom of the 1980s and 1990s it was common to read assertions to the effect that programming could be reduced to a quasi-logical process of theorem proving and formula derivation.

However, despite having a background in both programming and logic, I found something elusive about such assertions. My practical experience of programming had led me to understand it as a very different type of activity from the formal manipulations of logical proof. With the intention of resolving this uncertainty, I embarked upon a Ph.D. to study the ways in which the development of programming languages had been influenced by logic. This book is based on the resulting thesis, although it has a wider focus and marks a further stage in my growing understanding of the connection between the two areas.

Broadly speaking, I have come to understand this connection not as a *fact* about the two disciplines, but as something more like a *decision* made by identifiable historical actors as to how the new discipline of programming should be understood and structured. A key event in this process was the creation of the Algol language in the years around 1960. The motivation for this decision has very deep roots, however, reaching back to ideas about machines and machinic processes that date from the very beginning of the scientific revolution.

From this starting point, the book gives an account of the history of what we now call programming. It is useful, however, to view this not simply as *computer* programming, but to think more generally of attempts to define the steps involved in computations and other information-processing activities in such a way that they could be performed by machines, or at least by humans mimicking the behaviour of machines. From this perspective, the history of programming is distinct from the history of the computer, despite the close relationship between the two in the twentieth century.

The story told in this book stops in the 1970s, at the point where object-oriented programming emerged as a successor, or alternative, to structured programming. This is not an arbitrary cut-off point; rather, I believe that structured programming marks the completion of a particular historical trajectory, and that object-orientation, despite its rich inheritance from existing practices, is something quite different, or at least something which has deep roots in approaches and disciplines other than logic. Untangling these roots is part of a different project, however, and the history of how programming and programming languages developed after the 1970s is part of a different story from the one told here.

There are several things this book is not, therefore. Although it contains much historical material, it does not pretend to be a complete history of programming languages, and still less a history of the computer. What it does try to do is to tell a particular story about these historical developments and to show one way in which the history of programming languages can be set in a wider context and seen as an integral part of a development which, ultimately, is central to the project started by the scientific revolution in the seventeenth century.

The history of computing has recently been expanding its focus from technical details and embracing wider narratives and historical contexts. These encouraging developments have been principally evident in studies of commercial and military applications of computing and their societal impacts. If I had a single hope for this book, it would be that it might contribute to a similar process in the more technical aspects of the subject and to inspire computer scientists and historians of science and technology to see programming not as a isolated technical field. but as an interesting and important part of general intellectual history.

Acknowledgements I would like particularly to thank my supervisor Donald Gillies and external examiner John Tucker, without whose enthusiastic advice, support and encouragement this book would never have seen the light of day.

London, UK Mark Priestley

Contents

Chapter 1
Introduction

Since its development at the end of the Second World War, the electronic digital computer has been widely seen as a revolutionary technological innovation. Within the space of 50 years, the computer became part of everyday life to an extent that was unforeseen by many of its developers. Computers form the infrastructure of business and commerce worldwide; desktop, portable and mobile computers are ubiquitous, used for both work and leisure, small embedded computers are found in virtually all complex machinery, and more and more people are connecting to the worldwide network of the Internet. For many years, it has been a commonplace to refer to a 'computer age' or 'computer revolution'.

The very ubiquity of computers might make us reflect, and ask how it is that one device can fulfil such a dazzling array of roles. The answer, of course, is that the computer is a *universal* device, in a sense made precise by the British mathematician Alan Turing in the 1930s. Unlike, say, a washing machine, which is designed to perform one specific function, a computer has the potential to perform an unlimited range of tasks. For example, one and the same machine might be used to process text or financial data, to act as a communication device across the global networks, to play games, music or movies, or to simulate other devices.

Computers possess this flexibility because of the great range of *programs* that can be run on them. A program is a set of instructions telling a computer how to perform a particular task: the flexibility of the computer is therefore limited only by the ingenuity of its programmers in describing complex activities in a way that can be interpreted by the machine. In fact, a better way of looking at the situation is to notice that computers perform only the single task of carrying out the instructions in a program: universality is not an intrinsic property of computers, but is derived from the range of programs that can be written for them.

Programs are often referred to as *software*, as opposed to the electronic *hardware* that makes up the computer itself. This terminology marks a basic distinction: whereas computers are physical devices, programs are linguistic, or logical, entities. A program can be thought of as a text, and the conventions governing how program instructions should be expressed so as to be interpretable by a computer are thought of as defining a programming *language*.

M. Priestley, *A Science of Operations*, History of Computing,
DOI 10.1007/978-1-84882-555-0_1,

Both the programs themselves and also the languages in which they have been written have been addressed by historians of programming. Early work focused on the details and evolution of programming languages, and this work is documented in a number of books and conference proceedings.[1] From about 1980, however, professional historians began to take an interest in the history of computing, and one effect of this was an increased emphasis on the social and economic context surrounding the technical developments. Among other things, this contributed to a move of historical focus away from accounts of programming languages to a more general attempt to write the history of software.[2]

This book has a slightly different aim, namely to describe the history of the idea of a programming language, rather than that of the individual languages themselves, to consider the different forms that notations for expressing programs have taken in the course of their evolution. This history is seen as part of a larger story about the development of ideas about machinery and mechanical approaches to language.

The analogy between programs and machines was made explicit by Turing, who was concerned to offer an account of what was known as *effective computability*, or the extent to which humans can carry out mental processes, such as calculation, in a mechanical way. He defined a class of very simple information-processing devices, now known as Turing machines, each of which performed a single computational task. Rather than building these machines, however, he developed a notation, known as *machine tables*, to describe what they did. One of these machines, the *universal* machine, played a significant role in Turing's theory. The distinguishing feature of the universal machine was that it could simulate the behaviour of any other machine, if only it was provided with the table for the machine to be simulated.[3]

The tables interpreted by the universal machine are equivalent to the programs run on a modern-day computer, of course. To reinterpret Turing's insight in modern terminology we might say that programs are descriptions of information-processing machines. We do not have to build these machines, however, because the universal machine is capable of simulating the behaviour of them all.

Understanding this relationship between programs and machines opens up the possibility of widening the scope of the history of programming to include episodes involving the construction and use of certain information-processing machines that predate the invention of the electronic digital computer. This in turn suggests a wider perspective, which sees the history of programming as part of the broader field of the history of machinery. Ideas about programming have evolved in parallel with developments in machinery, and in particular the development of machines meant to carry out programmed instructions.

[1] Well-known early books on the history of programming languages include those by Saul Rosen (1967) and Jean Sammet (1969). A subsequent series of conferences organized by the ACM has documented the history of many individual languages (Wexelblat 1981; Bergin and Gibson 1996; Ryder and Hailpern 2007).

[2] See the book by Martin Campbell-Kelly (2003) for a good example of this approach.

[3] Turing's ideas, and the machine table notion, are covered in detail in Chap. 4.

In this book, the history of programming is seen to stand at the intersection of the two fields of machinery and language, and in particular to be closely related to attempts to give a mechanical account of language on the one hand, and a linguistic account of machines on the other. The starting point for the story told here will be the machines and the associated concept of the mechanical which were developed at the start of the nineteenth century, partly in response to the increasing prominence of machines in the early industrial revolution. The remainder of this chapter will consider the background of this work in a debate about machines and languages that dates from the very start of the scientific revolution.

1.1 Minds, Method and Machines

In 1620, Francis Bacon published a collection of texts which were intended to form part of the *Instauratio magna*, a multi-volume work in which he called for a complete overhaul of the ways in which scientific research was carried out, and a "wholesale *Renewal* of the sciences, arts and all human learning, raised on proper foundations".[4] The most complete part of the work was the second, the *Novum Organum*, in which Bacon described his ideas about scientific method.

Bacon saw a great difference between the state of the sciences and that of the "mechanical arts". The sciences appeared to make little or no progress, and to be stuck in a traditional Aristotelian approach, characterized by Bacon as involving never-ending verbal debates which led to no new knowledge or useful discovery. By contrast, the mechanical arts "as if they were partaking of a certain breath of life, grow and get better by the day",[5] a condition he attributed to the fact that they were based on observation and experimentation.

Nevertheless, progress in the mechanical arts appeared rather haphazard. Bacon attributed this to the fact that they lacked a definite method which could guide the process of inquiry. He drew an analogy between physical and intellectual labour: just as the productive capacity of the human body can be greatly enlarged by the use of appropriate tools, so too could the capacity of the human mind. He described two kinds of assistance in particular.

Firstly, it appeared that the mind needed the guidance of a method to direct it in its inquiries. In the very first sentence of the *Instauratio*, Bacon wrote that "the human intellect was the author of its own difficulties", by not making use of available remedies. He recommended that "from the very start, the mind should not be left to itself, but constantly controlled; and the business done, as it were, by machines".[6] For empirical investigations, Bacon suggested that a form of induction be

[4]Bacon (1620). Originally written in Latin, this work has been variously translated as *The Great Instauration* (Rees and Wakely 2004) and *The Great Renewal* (Jardine and Silverthorne 2000), and the second part as *The New Organon*.

[5]Bacon (1620), preface to the *Instauratio magna*. Translated by Rees and Wakely (2004), p. 13.

[6]Bacon (1620), preface to the *Novum Organum*. Translated by Jardine and Silverthorne (2000), p. 28.

used. The account he gave of induction was rather different from the way in which it was later understood, but it was nevertheless clearly distinguished from the deductive approaches adopted by Aristotelians, which centred on the use of the syllogism. In the field of mathematics, Bacon explicitly appealed to the idea of machinery to emphasize the importance of the help that a suitable method could give to the understanding:

> For I recall that when you have a machine to hand a demonstration in mathematics is easy and transparent, but that without one everything seems obscure and more subtle than it actually is.[7]

In the empirical, observation-based science envisaged by Bacon, however, the senses needed assistance as much as the intellect. He proposed that this assistance could be provided by making use of experiments. Experiments could be more subtle than the senses, and increase their power. Part of this increased power came from the fact that experiments were consciously designed to "confine and harass" nature, to create situations and reactions that went beyond what could be achieved by unaided human capabilities. In making this proposal, Bacon was drawing upon the success of the mechanical arts: his idea was, essentially, that experiments were mechanical aids to the senses.

Bacon's vision of a new approach to scientific enquiry, therefore, was informed in two distinct ways by mechanical metaphors. The workings of the mind itself were to be governed by rules by the application of which knowledge could be systematically produced, and not left to emerge by chance. In this endeavour, it was to be assisted by experiments which would draw upon, and further develop, the existing practice of the mechanical arts.

1.2 Language and Science

An important part of Bacon's proposals was the unmasking of the illusions, or idols, which interfered with, or blocked, the relationship between mind and world, and so hampered the intellect's access to truth. Bacon described four categories of illusion: the idols of the tribe, or those illusions arising from human nature; the idols of the cave, those arising from individual peculiarities; the idols of the marketplace, those attributable to language; and finally the idols of the theatre, the illusions caused by adherence to the doctrines of past philosophies.

Of these, Bacon regarded the idols of the marketplace, the illusions and errors caused by language, as the most problematic: because of the role that language plays in shaping thought, it was harder to rid the mind of these than of the other types of idol. The idols of the marketplace were of two types. The first was the use of "the names of things which do not exist", and the second "the names of things which do exist but are muddled, ill-defined, and rashly and roughly abstracted from

[7] Bacon (1620), Plan of the Work. Translated by Rees and Wakely (2004), p. 43.

the facts".[8] Bacon pointed out that many intellectual debates ended in controversies about the meaning of words, and even suggested that "it would wiser (following the custom and practice of the mathematicians) to reduce these controversies to order by beginning with definitions".[9] However, he recognized that things were not so simple in empirical science, where words could not always be defined in terms of other words but had, at some point, to connect with things outside language.

The common source of the idols of the marketplace lay in the failure of language to correspond exactly with the world: words could either fail to refer to anything, or have a confused definition which muddled together aspects of more than one type of thing. Reflection on these problems throughout the seventeenth century led to a change of ideas about language, and in particular about the relationship between science and language.[10] The new science emphasized the role of observation and experiment, and characterized earlier work as sophistical and over-concerned with argument and disputation caused by the shortcomings of natural language. It was felt that reliance on language obscured a clear view of the truths of nature that were accessible to simple observation.

It was of course recognized that science could not function without language, or at least without some means to communicate and share observations and results with scientists in one's own and other countries. This recognition led many scientists and philosophers to consider ways of reforming natural language or of developing artificial languages that would be more suitable for scientific work. Two approaches were common: the first attempted to reform natural language and to remove the problems that Bacon had identified, while the second drew inspiration directly from the symbolic language of mathematics.

Universal Languages If the root of the problem was the failure of the terms in natural languages to correspond precisely to the objects that science was uncovering in the world, one attractive path to a solution was to devise a 'scientific language' whose terminology would be more exact.

In the first half of the seventeenth century, a number of attempts along these lines were made to develop what was sometimes called a "real character", based on a complete categorization of all the objects and concepts that could be referred to in scientific discourse. Some proposals followed the route of giving new or modified definitions to words, while others invented complex sets of new symbols to represent scientific concepts.

On the basis of this precise vocabulary, a "philosophical language" could then be defined which would be suitable for clear and unambiguous communication on any subject whatsoever. One particularly comprehensive and ambitious proposal was the *Essay Towards a Real Character and a Philosophical Language* written by John Wilkins, Bishop of Chester and one of the founders of the Royal Society.[11]

[8] Bacon (1620), aphorism 60. Translated by Rees and Wakely (2004), pp. 93–94.

[9] Bacon (1620), aphorism 59. Translated by Rees and Wakely (2004), p. 93.

[10] A general discussion of this topic can be found in Jones (1932).

[11] Wilkins (1668).

A similar, if more programmatic, proposal was put forward by Leibniz, who hoped to create what he called a *characteristica universalis*. This was intended to be a universal language which would be adequate to represent the whole of human thought. Leibniz stressed the structured nature of his characteristic: he envisaged that it would define a set of elementary concepts from combinations of which all complex propositions could be expressed.[12]

A second, more original, part of Leibniz's ideas was his proposal for a *calculus ratiocinator*, a calculus or algebra which would make explicit the particular forms of 'reckoning' applied in reasoning. He made some progress in the development of such a calculus, formalizing, for example, the idempotency of logical addition, in which it differs from the numerical operation. This proposal is often seen as a forerunner of later developments in mathematical logic.

Similar ideas to those of Leibniz had been put forward by the philosopher Thomas Hobbes. In Hobbes' account of language, "general" or "universal" names signified collections, or "parcels", of objects, and "consequence" was a relationship between names corresponding to the relation of inclusion between these collections. He explicitly compared reasoning with numerical computation, as follows:

> When a man *Reasoneth*, he does nothing else but conceive a summe total, 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.[13]

Symbolic Language None of the various proposals for a new symbolic language of science put forward at this time achieved any significant level of use, however. In this same period, however, mathematical language was undergoing a transformation of the way in which in understood symbols and their meaning and role.[14]

This innovation is often associated with the name of Francois Viète, who in 1591 published a book entitled *Introduction to the Analytic Art*. In place of the words and abbreviations drawn from natural languages that had been used up to that point, Viète used arbitrary letters to stand for the quantities in equations. He represented unknown quantities by the vowels A, E, I, O and U, and used consonants to represent known quantities. This second innovation in particular allowed a level of abstraction that had not previously been available, and allowed Viète to find general solutions of equations more easily than before. According to Pycior, this new symbolism had the effect of focusing the attention of mathematicians on the structure of formulae.

The symbolical style was enthusiastically adopted in England partly thanks to the efforts of William Oughtred, whose textbook *The Key of the Mathematicks* was published in 1631. Oughtred endorsed Viète's innovation, arguing that the use of symbols enabled mathematicians to grasp situations more quickly, and enabled them

[12] Lewis (1918).

[13] Hobbes (1651), Chap. 5.

[14] The remainder of this section is largely based on the account given by Helena Pycior (1997).

to reason more generally. He introduced a wide range of new symbols, including those for operations and relationships of quantity.

The concision and clarity of the symbolic style recommended it to the scientific reformers. Thus Bacon recommended that disputes caused by language could be alleviated by following the practice of the mathematicians, and Leibniz's *calculus ratiocinator* attempted to capture common patterns of reasoning in symbols.

1.3 The Age of Machinery

At the beginning of the nineteenth century, the effects of the industrial revolution in England were becoming increasingly apparent. One particularly prominent symptom was the changes brought about by the increasing use of machinery in all areas of industry. As well as the invention and use of new types of machines in many areas, machines were widely seen to be bringing about great social changes, particularly in the growth of the factory system and the associated movement of population from the countryside to the new and expanding cities of the industrial north.

The scale and significance of the external and material changes brought about by these developments were widely recognized by contemporary commentators. There was also a wider cultural effect, in that an increasing familiarity with machines provided metaphors and enabled modes of thought that affected developments in other areas perhaps far removed from manufacturing industry. In his essay *Signs of the Times*, published anonymously in 1829, Thomas Carlyle considered that the present age could best be described as "the age of machinery", and wrote that "men are grown mechanical in head and in heart, as well as in hand".[15] Carlyle's essay is of interest, as it attempts to give a general account of the cultural meaning of the mechanical, rather than a straightforward description or criticism of technological innovations.

Carlyle characterized the mechanical by contrasting it with what he saw as its opposite: he saw it as external rather than internal, concerned with procedures and methods for bringing about certain ends, rather than a more intuitive emphasis on the ends themselves. In the social and political arenas he saw the mechanical in the proliferation of societies, meetings and periodicals by means of which people were enabled to work together to achieve their purposes, a tendency contrasted with the less planned and more inspirational approach which he detected in the past. In the arts, he saw the genius of the great figures of the past being replaced by the creation of academies; in education, the inspirational effect of a great teacher being replaced by systems of instruction, turning it into "a secure, universal, straightforward business, to be conducted in the gross, by proper mechanism, with such intellect as comes to hand". In summary, "everything has its cunningly designed implements, its preestablished apparatus; it is not done by hand, but by machinery".

[15] Carlyle (1829).

Of particular interest is Carlyle's description of an increasing mechanization in abstract thought. He described contemporary trends in philosophy as moving from an intuitive, meditative approach to one concerned more with argument, and the tracing out of causes and effects. Associated with this was a decline in importance of the philosophy of mind in favour of an emphasis on properties of matter. In a striking image, Carlyle wrote that "philosophers ... stand among us not to do, not to create anything, but as a sort of Logic-mills to grind out the true causes and effects of all that is done and created". A similar situation prevailed in mathematics, where the same industrial metaphor was developed at greater length:

> their calculus, differential and integral, is little else than a most cunningly constructed arithmetical mill; where the factors being put in, are, as it were, ground into the product, under cover, and without other effort on our part than steady turning of the handle.

For Carlyle, the basic characteristic of the mechanical approach to a particular subject was an interest in the arrangement and combination of its elements in such a way as to bring about certain ends in a predictable and controllable manner. It was an analytical approach, laying open and making visible the operative structure of things; it required no application of insight or intuition, but rather an appeal to ingenuity.

1.4 The Mechanization of Mathematical Language

At the beginning of the nineteenth century, a long-running debate about the relative merits of the geometrical and algebraic approaches to mathematics again became prominent.

Playfair's Paradox Prior to the nineteenth century, mathematics was considered to be a science of *quantity*. It was common to present mathematical arguments in the form of geometrical demonstrations in which arbitrary quantities were represented by particular line segments and angles. These representations were understood to be quantities themselves, and so arguments using them were guaranteed not to exceed the limits of what was true of quantity in general. Mathematicians reasoned about the connections between complex ideas about quantity, and geometrical notation simply made these connections visible.

The increasing use of algebraic techniques, however, called into question many of these traditional assumptions. Algebra used arbitrary symbols to represent both quantities and operations, and it appeared to many observers that this enabled reason to somehow outstrip or come adrift from its subject matter. According to the Scottish mathematician John Playfair, "the analyst continues to reason about the characters after nothing is left which they can possibly express ... obscurity and paradox must of necessity ensue".[16] In particular, unrestrained algebraic manipulation introduced and made extensive use of negative and imaginary numbers, such as $\sqrt{-1}$. Such

[16] Playfair (1778), p. 319.

numbers did not conform to existing notions of quantity: for example, they could not be thought of as representable by the length of a line segment. As a result, they were frequently referred to as *impossible quantities*.

Expressions for impossible quantities were therefore often taken to be meaningless, because they denoted nothing: this in turn raised the question of what sense could be made of formulae in which they occurred. A common view was that such expressions acted as a diagnostic sign of inconsistency or impossibility in the stated conditions of a problem. The occurrence of $\sqrt{-1}$, for example, in a solution meant that the problem was in some way ill-stated, or admitted of no solution.

However, the situation was more problematic than this, because in many cases the use of impossible quantities in mathematical arguments seemed to lead to correct and useful conclusions. As Playfair put it:

> Here then is a paradox which remains to be explained. If the operations of this imaginary arithmetic are unintelligible, why are they not also useless?[17]

Playfair's explanation of this phenomenon appealed to a notion of analogy: he considered a proof of certain properties of the circle and noted that if the impossible expression $\sqrt{-1}$ was replaced throughout by $\sqrt{1}$, the result was an unobjectionable demonstration of the equivalent properties of a hyperbola. He took the acceptability of the second proof as a warrant of the validity of the reasoning in the proof that used impossible expressions, appealing to a principle of analogy between "the subjects of investigation".

Woodhouse's Formalism Twenty years later, Playfair's arguments were criticized by Robert Woodhouse, a fellow of Caius College in Cambridge, who went on to put forward his own solution of the paradox. Woodhouse believed that "there can be neither paradoxes nor mysteries inherent and inexplicable in a system of characters of our own invention, and combined according to rules, the origin and extent of which we can precisely ascertain",[18] and wished to provide a stronger justification of the use of impossible quantities in mathematical reasoning than simply an appeal to analogy.

According to Woodhouse, the truth of an algebraic formula such as

$$(a+b)(c+d) = ac + ad + bc + bd$$

derived from "observations made on individual objects". In this case, the relevant observations concerned the properties of the addition and multiplication of real quantities. The use of letters conferred the advantage of generality: the equation will be valid whatever quantities the letters are taken to stand for. However, when the same operations are performed with imaginary quantities, giving for example

$$\left(a + b\sqrt{-1}\right)\left(c + d\sqrt{-1}\right) = ac + ad\sqrt{-1} + bc\sqrt{-1} - bd,$$

[17] Playfair (1778), p. 321.

[18] Woodhouse (1801), p. 93.

there no longer appeared to be any justification for its truth, because we have no knowledge of the properties of the imaginary quantities, or even reason to believe in their existence.

In Woodhouse's view, expressions containing symbols for imaginaries were to be understood as extensions of similar expressions involving only real symbols. Thus the equality of the two terms in the second equation above was not demonstrated, but postulated. In his words, they were "not proved equivalent, but put so". The justification for this procedure lay in the fact that the properties of the operators involved were being preserved. Thus,

> it is to be understood, that ... $x\sqrt{-1} - x\sqrt{-1}$ is put equal 0, not by bringing $\sqrt{-1}$ under the predicament of quantity, and making it the subject of arithmetical computation, but by giving to + and − their proper signification when used with real quantities, and then they designate reverse operations.[19]

In other words, rather than justifying particular manipulations by appealing to the properties of the objects being manipulated, Woodhouse appealed to the properties of the operators being used: "the relations between the symbols of general terms were to be established by giving the true meaning to the connecting signs, which indicate not so much the arithmetical computation of quantities, as certain algebraical operations".[20]

In parallel with this approach, Woodhouse gave a novel account of mathematical reasoning. He cited the traditional Lockean description of a valid step in reasoning as one which preserved the "agreement of ideas", but immediately rephrased this in terms of propositions rather than ideas, thus radically changing its meaning. For Woodhouse, reasoning was concerned more with the transformation of symbols than the agreement of ideas, "each proposition being converted into its next, by changing the combination of signs that represent it, into another shewn to be equivalent to it".[21]

Playfair had noted the apparent tendency of algebraic reasoning to outstrip sense, and viewed it as a problem that required explanation. Woodhouse, by contrast, seems to have viewed it as a demonstration of the power of algebraic techniques, and used it to motivate a novel account of mathematical language in which emphasis was placed on the formal properties of operators rather than the intuited properties of quantities, and on the equivalence of propositions rather than the agreement of ideas.

The Analytical Society In 1812, a number of Cambridge students formed the so-called Analytical Society. The principal mission of the society was to encourage the dissemination of continental approaches to mathematics at Cambridge. Charles Babbage was a founder member of this society, and other notable members included John Herschel, the son of the renowned astronomer William Herschel, and George

[19] Woodhouse (1801), p. 99.

[20] Woodhouse (1802), p. 86.

[21] Woodhouse (1801), p. 107.

Peacock. The first meeting of the society took place in rooms in Caius College, and Woodhouse's work was certainly known to them, but Woodhouse himself does not appear to have taken an active part in the society, whose members were all students of the University.

In addition to its general mathematical interests and activities, the Analytical Society devoted a fair amount of attention to methodological issues of notation and language in mathematics. Their conclusions are presented in the preface to the Memoirs of the Analytical Society, published anonymously but in fact co-written by Babbage and Herschel. They described the "peculiar province of mathematical analysis", as opposed to other branches of mathematics, as being the examination of "the varied relations of necessary truth", and took a very positive view of its achievements: the techniques of analysis enabled mathematicians to pursue "trains of reasoning, which, from their length and intricacy, would resist for ever the unassisted efforts of human sagacity".[22]

This success was attributed to the "accurate simplicity" and concision of the language of analysis, and its methodological approach. According to Babbage and Herschel, "by separating the difficulties of a question" analysis made it possible to reduce complex problems to simpler ones, which could be, if not solved, at least clearly identified as outstanding problems.

The Society in general, and Babbage in particular, were struck by the advantages that could be gained by the use of a suitable symbolic notation. One of the most important ambitions of the society was to replace Newton's fluxional notation for the calculus, used in Britain, with the Leibnizian notation used on the continent; the assertion was made that the adoption of an unsuitable notation had greatly retarded the development of mathematics in Britain.

The particular benefits of analytical notation derived firstly from the fact that its symbols were precisely defined, and thereby given a meaning which did not change, and secondly from the advantage that symbols gave in abbreviating and making accessible longs chains of reasoning. To explain this, Babbage and Herschel drew an explicit analogy with machinery:

> It is the spirit of this symbolic language, by that mechanical tact (so much in unison with all our faculties) which carries the eye at a glance through the most intricate modification of quantity, to condense pages into lines, and volumes into pages.[23]

In particular, they highlighted "the happy idea of defining the result of every operation, that can be performed on quantity, by the general term of function, and expressing this generalization by a characteristic letter" as an innovation that both generalized and simplified the notation, and so suggested possibilities for further research as well as providing the means to describe that research.[24]

[22] Babbage and Herschel (1813), p. i.

[23] Babbage and Herschel (1813), pp. i–ii.

[24] Babbage and Herschel (1813), p. xiv.

Babbage's Philosophy of Analysis In the light of his importance in the history of computing, it is worthwhile here to examine Babbage's ideas about analysis and mathematical notation in more detail. The Analytical Society was itself short lived, but Babbage remained in contact with some of its members, Herschel and Peacock in particular, for many years, and he returned to consider the issues of notation and methodology, working for a number of years on an unpublished manuscript titled *The Philosophy of Analysis*.[25]

Babbage described the application of analysis to a problem as consisting of three stages. The first consisted "in translating the proposed question into the language of analysis", the second in "comprehending the system of operations necessary to be performed, in order to resolve that analytical question into which the first stage had transformed the proposed one", and lastly "retranslating the results of the analytical process into ordinary language".[26]

It is clear from the examples that he gave that by the 'language of analysis' Babbage meant the symbolic language of algebra and the calculus. The strategic aim of his mathematical work was to demonstrate the superiority of the algebraic approach over the geometrical. A geometrical solution involved drawing diagrams representing the problem, and reasoning directly on the basis of these diagrams, thus keeping an intuitive connection with the reality of the problem. By contrast, analysis translated the problem into the more abstract language of algebra, and the steps of the subsequent argument could not be directly compared with the problem until they were 'retranslated'.

Much of the success of analysis was attributed to the nature of the notation used, and in particular to "the accurate simplicity of its language". In part Babbage is referring here to the straightforward semantics of symbols that are introduced by means of definitions. After its introduction, a symbol "can neither convey, nor excite any idea foreign to its original definition". Babbage later expanded on this thought, explaining that symbols have, by and large, one meaning, which is originally given by a simple definition. This leads to a lack of ambiguity which makes the operations of reason transparent and checkable and, by reducing the burden on memory, leads to a greater speed in mental operation.

A further important property of analytical notation, according to Babbage, is its concision. This helps by reducing the load on memory, thus making it easier to exercise the judgement in deciding on the validity of a particular step in a train of reasoning.

A final point worth noting here is that, in his mathematical writings, Babbage frequently commented on the importance of the distinction between operations and quantity, picking up an earlier point about the utility of functional notation. In a popular account of mathematical notation, published in 1830, he stated this most clearly:

[25] Dubbey (1978).

[26] Babbage (1827), p. 346.

> There are in analysis two great divisions of symbols—those which denote quantity and those which denote operations.[27]

Peacock's Algebra In 1830, Peacock published *A Treatise on Algebra*, which gave a comprehensive and very influential statement of the new view of algebra, one which was profoundly influenced by the discussions of the Analytical Society.[28] Peacock drew a fundamental distinction between 'arithmetical' and 'symbolical' algebra. Arithmetical algebra was a generalization of arithmetic: symbols replaced numbers, but the significance of expressions was still given purely in terms of their numerical interpretation. Thus an expression such as $a - b$ only made sense in the case where a was greater than b.

By contrast, in symbolical algebra symbols were considered to be quite general, capable of representing "all species of quantity", including negative and imaginary numbers. Operations were defined by their laws of combination: thus rather than being thought of as representing the concrete numerical operations of addition and subtraction, the operations denoted by $+$ and $-$ were defined to be inverses of each other.

The laws of combination of the operations were suggested by the properties of the arithmetic operations, thus guaranteeing the applicability of symbolical algebra to arithmetic. It was assumed that the same laws would apply generally, an assumption known as the principle of the permanence of equivalent forms. This principle was applied to extend the applicability of an expression like $a^m \times a^n = a^{m+n}$ from whole numbers to negative numbers, say, and also to give an interpretation to an expression like a^{-n}, by observing that $a^{-n} \times a^n = a^0 = 1$.

By 1830, then, the debates about the use of impossible quantities in algebra had led to the working out of a radical new account of algebra, which saw it as the science of the laws of symbolic combination or, as Peacock put it, "the science of general reasoning by symbolical language".[29] Two aspects of this view are of particular importance here.

Firstly, Peacock viewed algebra as being primarily a language of *operations*: the quantities or objects being operated on were represented by symbols, but nothing was assumed about their nature or properties. The general applicability of operations to all quantities was assumed, and subsequent development was governed by the laws of combination of the operations.

Secondly, Peacock held that algebraic demonstrations were legitimate even in the absence of an interpretation of the symbols involved. Only the final expressions in a demonstration needed to be interpreted, the intermediate stages being justified on the grounds that they were "algebraically necessary". This was Peacock's approach to Playfair's paradox, one which appears to sidestep it rather than provide a resolution; this is indicative perhaps of how the climate of debate had changed since Playfair wrote. Peacock's view is very similar to Babbage's account of the analytical method,

[27] Babbage (1830), p. 398.

[28] Peacock (1830).

[29] Peacock (1830), p. 1.

highlighting the commonality of assumptions shared by the former members of the Analytical Society. Peacock did not rule out the possibility of providing geometrical interpretations of impossible quantities, but his view of symbolical algebra meant that they were not needed to justify the properties of algebraic operations.

The Machine Metaphor From the very beginning of this debate, a comparison was drawn between the operations performed by the mathematician who carried out symbolic manipulations, and the operations of machines. Such observations were often accompanied by comments bemoaning the apparently diminished place left for the employment of intuition and intelligence in mathematical work. Playfair, for example, considering the manipulation of formulae which contained impossible expressions, protested:

> Is investigation an art so mechanical, that it may be conducted by certain manual operations? Or is truth so easily discovered, that intelligence is not necessary to give success to our researches?[30]

The opposition perceived here between 'mechanical' operations and those requiring intelligence was echoed by Woodhouse:

> During the operation of these quantities, it is said, all just reasoning is suspended, and the mind is bewildered by exhibitions that resemble the juggling tricks of mechanical dexterity.[31]

Woodhouse, in line with his overall view, is in general less critical of mechanical approaches than Playfair, though he also issued a warning against "mathematicians, neglecting to exercise mental superintendance, [being] too prone to trust to mechanical dexterity".[32]

Babbage made similar comparisons between mental and mechanical processes, but without any trace of this kind of criticism. Instead, reflecting perhaps the spirit of Carlyle's age of machinery, he pointed out the assistance that the mechanical, or algebraic, approach could give to the productivity of the mathematician. Writing with Herschel, he stated that

> It is the spirit of this symbolic language, by that mechanical tact (so much in unison with all our faculties) which carries the eye at one glance through the most intricate modifications of quantity, to condense pages into lines, and volumes into pages.[33]

and he later referred to "the almost mechanical nature of many of the operations of Algebra" as something which "certainly contributes greatly to its power".[34] This line of thought has been pursued by William Ashworth, who writes of "the attempt of John Herschel and Charles Babbage to discipline the human mind and speed up the operations of intelligence through a philosophy of algebraic analysis".[35]

[30] Playfair (1778), p. 321.

[31] Woodhouse (1801), p. 90.

[32] Woodhouse (1801), p. 119.

[33] Babbage and Herschel (1813).

[34] Babbage (1827), p. 333.

[35] Ashworth (1996).

Babbage, of course, was to attempt more than this: he would take the metaphor of the machine literally, designing and endeavouring to build calculating engines which combined the new, mechanical philosophy of algebra with the physical power made available by the machine-based industry of the industrial revolution, as described in the following chapter.

Chapter 2
Babbage's Engines

Although Babbage was a significant figure in English mathematics at the start of the nineteenth century, he is now principally remembered for his work on the design and development of a series of what he described as mechanical calculating engines. He devoted a large amount of time and money to these projects, but unfortunately none of them was ever fully completed.

The earliest machine, the Difference Engine, was intended to compute and print accurate mathematical and navigational tables. In 1822, Babbage completed a small prototype of this machine, which demonstrated the feasibility of his ideas. This was displayed and widely commented on, and the subsequent development of a full-scale machine was funded by the British Government. Progress was slower than expected, however, and was further hindered by Babbage's occasionally problematic relationship with the engineers he employed to work on the construction. By about 1830, work had come to a standstill, despite the investment of a considerable amount of Babbage's own money in the project. Government funding was not renewed, and the Difference Engine was never completed.

In the early 1830s, when he was no longer occupied with work on the Difference Engine, Babbage conceived of a more powerful and flexible machine, which became known as the Analytical Engine. Over a period of years, he produced a large number of drawings and associated specifications for this machine, but he never seriously considered undertaking its construction.

Late in life, he revisited his earlier ideas, developing the design for a new and simplified difference engine, using insights gained from his work on the Analytical Engine. This machine became known as the Difference Engine No. 2, though again, Babbage did not succeed in completing the construction of the engine.

Babbage's machines, and in particular the Analytical Engine, are often portrayed as isolated precursors of the modern computer, and Babbage as an anachronistic genius who had the misfortune to live at a time when technology was insufficiently developed to enable him to fully implement his vision. When viewed in context, however, a rather different picture emerges. Babbage engaged whole-heartedly with the scientific, industrial and even political life of his time, and this chapter describes how his calculating engines can be seen in a fuller historical context.

M. Priestley, *A Science of Operations*, History of Computing,
DOI 10.1007/978-1-84882-555-0_2, © Springer-Verlag London Limited 2011

2.1 The Division of Mental Labour

From the onset of the Industrial Revolution, observers were struck by the increases in productivity that were brought about by the application of machinery and the factory system to the manufacture of goods of all sorts. In *The Wealth of Nations*, published in 1776, the Scottish philosopher and economist Adam Smith attempted to explain the great discrepancies observable between the levels of production in different countries. The starting point of his analysis was the idea of the division of labour, or breaking down the overall task of manufacturing a product into a number of simple processes, and the allocation of these processes to different workers within a factory.

Famously, Smith took as his primary example of the division of labour the trade of pin-making, and he described how as many as 18 different processes were involved in the manufacture of pins. These ranged from fundamental operations, such as cutting the wire used to make the pins to the length and thickness required, to less obvious tasks such as inserting the completed pins into paper holders before they could be sold. He took as an example a 'small manufactory' of ten workers which produced 48,000 pins per day, and by estimating that a single worker carrying out all the required operations could make at most 20 pins in a day, concluded that the division of labour had increased the productivity of the workers in the pin-making trade by a factor of at least 240.

Smith attributed the effect of the division of labour to three major causes. Firstly, he observed that a workman specializing in one simple process would become more skilled and so quicker in carrying it out than someone for whom it was one of many different tasks to be carried out during a day's work. A further economy of time was obtained by having each workman perform the same task continually, thus saving the time lost in switching from one type of work to another. Finally, in an observation that is particularly relevant to the concerns of this chapter, Smith pointed out that once an activity had been broken down by the division of labour, it was often possible to develop machinery that would assist or replace human workers in the execution of the simple processes involved, increasing the ease and efficiency with which the work was carried out.

Babbage whole-heartedly agreed with Smith's view of the division of labour: he devoted two chapters to it in his book *The Economy of Machinery and Manufactures*, describing it as "perhaps the most important principle on which the economy of a manufacture depends". In addition to the factors listed by Smith, Babbage believed he had identified a further important principle to explain the cheapness of articles manufactured through the division of labour, namely that "the master manufacturer, by dividing the work to be executed into different processes, each requiring different degrees of skill or force, can purchase exactly that precise quantity of both which is necessary for each process".[1] He also gave a typically detailed account of the pin-making trade in England and France, with details of the productivity and wages of the workers in each of the processes into which it was divided.

[1] Babbage (1835b), pp. 175–176.

Babbage went beyond Smith, however, and considered the effect of applying the division of labour not only to manufacturing processes but also to activities which involved mental rather than physical labour, and in particular, to the calculation of mathematical tables. His principal example was the production of a large set of tables recently undertaken in France under the direction of the engineer Gaspard Riche de Prony.[2]

De Prony's tables were produced as part of the reform of weights and measures instigated in the 1790s by the French revolutionary government. The reform was intended to introduce the decimal system wherever possible, and one application of this principle was a proposal to divide a right angle into 100, rather than 90, degrees. This system had been used in the survey of the Paris meridian that led to the definition of the new unit of length, the metre, and in 1794 it was decided that new tables of trigonometrical functions and their logarithms were required which were adapted to this new angular measure. The calculation of these tables was entrusted to de Prony, then the director of the Bureau de Cadastre, an organization concerned with the production of accurate land surveys required for the purposes of taxation.

De Prony had been instructed to ensure that the tables "left nothing to desire with respect to exactitude" and would be "the vastest and most imposing monument to calculation that had ever been executed or even conceived". The tables required an unprecedented amount of calculation, and when complete would contain several hundred thousand figures, calculated to an accuracy of between 12 and 22 decimal places. Clearly, a project of this magnitude would require careful organization, and according to his own account, de Prony explicitly drew upon Smith's ideas as an inspiration for the organization of his workforce: "I came across the chapter where the author treats of the division of work ... I conceived all of a sudden the idea of applying the same method to the immense work with which I had been burdened, and to manufacture logarithms as one manufacture pins".[3]

In order to apply the division of labour to computation, de Prony turned to a mathematical technique known as the method of differences. This method enabled complex logarithmic and trigonometric functions to be calculated by employing only the much simpler operations of addition and subtraction. Suppose, for example, that it is required to calculate the values of the formula $f(x) = x^2 + x + 41$. A few values of this formula are given in the second column of Table 2.1.

The *first differences*, shown in the column headed Δ_1, are found by calculating the difference between two adjacent values of the function. So the first figure in this column is found by computing $f(2) - f(1) = 47 - 43 = 4$. In a similar way, the *second differences*, shown in the column headed Δ_2 are found by computing the difference between adjacent values in the first difference column. It can be seen that in the case the second differences are all the same, a property that is always true for polynomial formulae in which the highest power is 2.

We can use this property to reverse the process and compute the next value of the formula using only addition. The fourth entry in the Δ_2 column will be 2; this

[2] Descriptions of this project have been given by Grattan-Guinness (1990) and Daston (1994).

[3] de Prony (1824), quoted in Grattan-Guinness (1990).

Table 2.1 Application of the method of differences to the function $x^2 + x + 41$

x	$f(x)$	Δ_1	Δ_2
1	43	4	2
2	47	6	2
3	53	8	2
4	61	10	
5	71		

can be added to the entry in the Δ_1 column to give the fifth Δ_1 value, which is 12; this can in turn be added to the last value in the $f(x)$ column to give the value 83. It can be checked by direct computation that the value of the formula for $x = 6$ is in fact 83, but the method of differences has allowed us to work this out without performing any multiplications. In a similar way, provided that the first number in each column is known, all the values of the function can be tabulated. Subtraction will be required as well as addition if negative numbers appear in the table.

The application of the method of differences is not always as straightforward as in the example above, but provided that some preliminary mathematical work is done, the method can be used to calculate the logarithmic and trigonometric functions that de Prony was concerned with to any required degree of accuracy. To bring this about, De Prony divided his workforce into three sections.

The first section consisted of mathematicians who derived the formulae that would be used to calculate the required functions. This work required a considerable degree of mathematical expertise; on the other hand, the number of workers required in the first section was relatively small, and they carried out no numerical work.

From these formulae, the workers in the second section derived the more detailed information required to compute the results using the method of differences. Unlike the simple example given above, logarithmic and trigonometrical functions do not have a constant final difference. If a sufficient number of differences are computed, however, the final difference will be constant over significant ranges of values of x. The job of the second section was to work out what these ranges were and to compute, for each range, the values in the top row of the table, including the constant last difference.

The results were handed on to the workers in the third section in the form of sheets "ruled with fifty horizontal lines ... and divided into a number of vertical columns according to the number of orders of differences which had to be written. The topmost line of each of these sheets reproduced the numbers determined by the calculators of the second section, and thus served as a point of departure".[4] The workers in the third section could then complete the sheets by performing only additions and subtractions, as explained above. Once completed, the sheets were passed back to the second section for checking; this could be done without repeating the detailed calculations.

[4] Lefort (1858), p. 127.

The workers in the third section were instructed in how to complete the sheets. This would have involved learning the order in which the intermediate results were to be calculated. The actual additions and subtractions were carried out on loose sheets which were discarded once the results had been transcribed onto the sheets prepared by the second section.[5] The third section, therefore, had little, if any, scope for the exercise of judgement or initiative.

Babbage later made an explicit comparison between the organization of de Prony's calculations and the typical organization of "a cotton or silk mill, or any similar establishment". Furthermore, he observed that the work of the third section might "almost be termed mechanical", and referred to a time "when the completion of a calculating engine shall have produced a substitute for the whole of the third section of computers".[6] As he makes clear in a footnote, this is a reference to his project to build a 'difference engine' which occupied much of his time, effort and money during the 1820s. This comment also provides a nice illustration of the way in which Babbage's work exemplifies Smith's third principle, according to which the introduction of new forms of machinery is a consequence of employing a suitable division of labour.

2.2 The Difference Engine

In Babbage's own accounts, the invention of the Difference Engine closely followed Smith's principle whereby the simple processes resulting from the division of labour became susceptible to mechanization. As early as 1813, Babbage and Herschel had made a case for the introduction of a suitable division of labour into computation:

> The ingenious analyst who has investigated the properties of some curious function, can feel little complaisance in calculating a table of its numerical values; nor is it for the interest of science, that he should *himself* be thus employed, though perfectly familiar with the method of operating on symbols; he may not perform extensive arithmetical operations with equal facility and accuracy; and even should this not be the case, his labours will at all events meet with little remuneration.[7]

In his autobiography, Babbage recounted an anecdote suggesting that the idea of mechanical calculation had occurred to him at this time.[8] However, it was not until several years later that he seriously entertained the idea of building a machine which would perform calculations. In 1821, he and Herschel were reflecting on the experience of overseeing a large computation in which "the calculations … were distributed among several computers", thus sparing Babbage and Herschel from "that wearisomeness and disgust, which always attend the monotonous repetition of arithmetical operations". However, they still found the preliminary calculations

[5] Lefort (1858), p. 134.

[6] Babbage (1835b), p. 195.

[7] Babbage and Herschel (1813), p. viii.

[8] Babbage (1864).

and subsequent comparison and verification of the results "a considerable trial of the patience of those who superintend them". As a result of this frustration, "it was suggested by one of us, in a manner which certainly at the time was not altogether serious, that it would be extremely convenient if a steam-engine could be contrived to execute calculations for us; to which it was replied that such a thing was quite possible",[9] and in Babbage's account, it was this casual suggestion which sparked in him a serious consideration of the possibility of mechanical computation.

In the following year, Babbage gave a similar account of the origin of the machine in a letter to Humphry Davy, then the president of the Royal Society:

> The intolerable labour and fatiguing monotony of a continued repetition of similar arithmetical calculations, first excited the desire, and afterwards suggested the idea, of a machine, which, by the aid of gravity or any other moving power, should become a substitute for one of the lowest operations of human intellect.[10]

In the same letter, he gave a description of de Prony's project, and stated explicitly the intended scope of his machine: "If the persons composing the second section, instead of delivering the numbers they calculate to the computers of the third section, were to deliver them to the engine, the whole of the remaining operations would be executed by machinery".[11]

Babbage did not view the Difference Engine as simply a calculator, however, but rather as an attempt to mechanize a complete process, namely the production of mathematical tables. Tables were by far the most important mathematical aid in use at the time: those in use ranged from simple tables of products to those tabulating complex trigonometric functions and, as in the project entrusted to de Prony, were often produced in both natural and logarithmic forms. The areas in which tables were used also varied widely: a particularly significant market was the production of astronomical tables, which found a very practical application to the maritime navigation on which international trade depended.

As well as the saving of labour, Babbage repeatedly stressed a further benefit to be hoped for from the mechanization of calculation, namely the avoidance of errors. It was well known that all published tables contained errors, and the detection and correction of errors was an important aspect of the work of calculation. Errors could arise not only in the process of computation, but also in the subsequent printing process, and Babbage placed as much emphasis on the mechanization of typesetting as on that of calculation. His overall vision was of a machine divided into two parts, each dealing with one of these fundamental processes.

> In order to produce printed tables free from error I proposed the engine should be able to calculate any tables whatever, and that it should produce a stereotype plate of the computed results … This view of the subject naturally divided the engine into two great branches: one part must make the calculations, another must produce the plates.[12]

[9] Babbage (1822c), p. 15.

[10] Babbage (1822b), p. 3.

[11] Babbage (1822b), p. 10.

[12] Babbage (1822c), p. 16.

Table 2.2 A schematic calculating sheet equivalent to the Difference Engine

x	$f(x)$	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6
1	7	…	…	…	…	…	…
…	…	…	…	…	…	…	…

In order to be able to produce "any tables whatever", Babbage needed a simple yet general mathematical method and, like de Prony and many table-makers before him, he turned to the method of differences. As well as its generality, this method recommended itself to Babbage because of "the great uniformity which it would necessarily introduce into all [the machine's] parts … The whole difficulty was now reduced to that of forming a machine which should add or subtract, as might be required, any number of differences".[13]

The basic idea behind the calculating part of the engine was to provide a physical representation of one row of the calculating sheets used by de Prony. Babbage planned to build a machine which would calculate with six orders of differences and up to 20 digits in each number stored; if de Prony had organized his calculations on a similar scale, the sheets used would have looked something like Table 2.2.

Babbage presented the design of the Difference Engine as proceeding through a number of stages; at each stage he showed how a complex operation could be analyzed in terms of simpler ones, which could then be recombined to produce the desired overall effect.[14]

To complete such a table, a computer would start from the right, adding the last difference to the preceding column, then adding the resulting number to the column to its left, and so on until the next function value had been obtained. Babbage's first step of analysis, therefore, was to observe that if he could devise a mechanism for adding one number to another, this mechanism could be replicated as often as necessary to add the successive differences across a row in the sheet.

However, each number in the table was itself made up of a number of digits; Babbage further observed that a mechanism that could add one digit to another could simply be repeated as often as required to add together all the digits in a number. This plan was slightly complicated by the necessity of dealing with *carries*, when the sum of two digits was greater than 9 and required a further increment to the digit in the next place. Babbage's strategy at this point was to separate these two fundamental operations: the overall design of the engine is therefore based on the repetition of two basic mechanisms, one to add together two digits, and the other to deal with any resulting carry. Babbage saw an analogy between the structure of the machine and that of arithmetic itself:

> In fact, the parts of which it consists are few but frequently repeated, resembling in this respect the arithmetic to which it is applied, which, by the aid of a few digits often repeated, produces all the wide variety of number.[15]

[13] Babbage (1822c), p. 17.

[14] The following description is based on the account given in Babbage (1822c).

[15] Babbage (1822b), pp. 6–7.

It is not necessary to consider the mechanical details of the engine here, but the overall organization of the machine is of some interest. Individual digits were represented by wheels mounted on an axle. The numerals 0 to 9 were inscribed on the circumference of some of the wheels and positioned in such a way that the number stored on the axle could be read off at the front of the machine. Babbage later referred to the assemblage consisting of a figure wheel and its associated machinery as a 'cage'. A complete number was represented by a set of cages mounted in a column on a single axle.

Adjacent columns represented the numbers stored in a row of a calculating sheet. Addition was performed by a rotation of the axis to which a number was fixed. By means of various contrivances, this would cause the digits represented on the moving axle to be added to those on its stationary neighbour, while at the same time the machine would record the various carries that were required. The operation of adding the carries to the resulting sum was performed in a separate step, once the rotation of the moving axle was completed.

The action of the various parts of the machine was to be coordinated in such a way that a number of additions could be performed simultaneously. In one step, the numbers stored on the columns representing the even differences would be added to the adjacent columns; then the columns would be geared differently, and the odd differences added to the even columns. In a description of the engine written after construction had halted, the journalist Dionysius Lardner described the functioning of the engine in poetical style:

> There are two systems of mechanical action continually flowing from bottom to top; and two streams of similar action constantly passing from the right to the left. The crests of the first system of adding waves fall upon the last difference, and upon every alternate one proceeding upwards whilst the crests of the other system touch upon the intermediate differences. The first stream of carrying action passes from right to left along the highest row and every alternate row, whilst the second stream passes along the intermediate rows.[16]

This nicely evokes the way in which the calculating part of the engine embodied the complexity of the method of differences, despite being built out of relatively simple components. By contrast, the design for the printing part of the engine was worked out in much less detail, and its details need not be considered here.

The difficulties that Babbage encountered, or created, in trying to construct a working difference engine have been described elsewhere, and it is not necessary to recount them in detail here. A prototype engine was built in 1822, computing only two rather than the proposed six orders of differences. Its application was therefore rather limited, but it served for Babbage as a demonstration of the feasibility of his ideas, and was displayed at his house. He used it to construct tables of square and triangular numbers, and tabulations of formulae such as $x^2 + x + 41$. With the support of the Royal Society and some financial assistance from the Government, construction of the full-size engine started in 1823 and occupied much of Babbage's time for several years. After a number of financial disagreements with his engineer, construction was suspended in 1833 and never restarted. Babbage made appeals to

[16] Lardner (1834), p. 298.

successive Prime Ministers for additional funding, but none was forthcoming, and in 1842 the Government finally decided to withdraw support from the machine. The completed portions of the machine ended up in the South Kensington Museum.

2.3 The Meanings of the Difference Engine

The Difference Engine was not, of course, the first mechanical calculator. However, earlier machines such as those of Pascal and Leibniz had only mechanized single arithmetical operations such as addition and multiplication, where the user would supply the numbers to be operated on and the machine would calculate the result.

By contrast, Babbage's engine was intended to automate several aspects of a complex process, namely the production of scientific and navigational tables by means of the method of differences. In contrast to the earlier machines, it would be *productive*: as it implemented a complete algorithm and not just a single operation, large numbers of results could be produced from a relatively small amount of input data. Furthermore, not only was it inspired by the division of labour apparent in the manual production of tables, but it also embodied a further division in the distinction between the calculating and printing parts of the machine.

The scale and novelty of Babbage's proposal generated a lot of interest in the Difference Engine; in the 1820s, Babbage had a small prototype set up in his house and demonstrated it to many interested visitors. The engine was widely perceived as having more significance than as a simple aid to calculation; rather, it was seen to be a novel application of the processes of mechanization that were clearly visible elsewhere in society.

The Mechanization of the Mental As discussed in Chap. 1, machines were used in many areas of manufacturing and industry by the 1820s, and their introduction had profound social and practical implications. Machine performance surpassed that of humans in many physical tasks, leading to great changes in the nature of work. Among the benefits claimed for mechanization were that it increased the power that a worker could apply to a task, and also enabled work to be carried out faster and more accurately than before.

Even more fundamental than these practical advantages, however, was the fact that the Difference Engine appeared to increase the scope of what could potentially be mechanized. In 1825, in an address given when Babbage received the gold medal of the Astronomical Society, Henry Colebrooke observed that hitherto "mechanical devices have substituted machines for simpler tools or for bodily labour", before commenting:

> But the invention, to which I am adverting, comes in place of mental exertion: it substitutes mechanical performance for an intellectual process: and that performance is effected with celerity and exactness unattainable in ordinary methods.[17]

[17] Colebrooke (1825), p. 509.

As Babbage later put it in a letter to the Prime Minister, the Duke of Wellington, the engine represented "the first conversion of mental into mechanical processes".[18] The fact that Babbage had disparaged calculation as "one of the lowest operations of the human intellect" does not alter the fact that this marks a significant extension of the reach of the mechanical.

Despite the novelty of the application area, however, Babbage and his associates appeared to align mental with physical labour, at least in terms of the benefits that mechanization could be expected to deliver.

Economy In common with other mechanical innovations, the Difference Engine was expected to deliver greater productivity at lower costs. The bulk of de Prony's workforce was made up of the calculators in the third section, and it was precisely this section that would be made redundant by Babbage's engine. Babbage also pointed out that the automatic printing facilities would result in "the whole work of the compositor being executed by the machine, and the total suppression of that most annoying of all literary labour, the correction of the errors of the press".[19]

A key factor in this projected saving was the fact that the operator would only have to supply a small amount of initial data, from which the engine would be able to calculate a significant portion of a table along with all its intermediate differences. For Colebrooke, this distinguished Babbage's proposal from the earlier machines of Pascal and Leibniz where the individual numbers to be operated on had to be set on the machine before each operation. The Difference Engine was not just a calculator, but encoded a complete mathematical process; furthermore, the constant repetition of the simple operations required to perform the calculation made the process a suitable one to be carried out by the unvarying behaviour of a machine.

The ability of the Difference Engine and related machines to automate an entire arithmetical process, rather than just single operations, also promised to remove an obstacle that was impeding the progress of mathematics in some areas. For example, Colebrooke referred to certain equations of Lagrange "which involve operations too tedious and intricate for use, and which must remain without efficacy, unless some mode be devised of abridging the labour or facilitating the means of its performance".[20]

Avoidance of Error Even more than its productivity, the capacity of the engine to deliver tables that were more reliable than those produced manually was stressed by Babbage who wrote that "[t]he quantity of errors from carelessness in correcting the press, even in tables of the greatest credit, will scarcely be believed".[21]

This theme was given great prominence by Lardner, who gave a compendious summary of the tables currently in use, but commented that their usefulness was

[18] Babbage (1834), p. 4.

[19] Babbage (1822b), p. 10.

[20] Colebrooke (1825), p. 512.

[21] Babbage (1822b), p. 5.

limited by their "want of numerical correctness", [22] reiterating Babbage's point that the engine was designed to remove not only errors of calculation, but also those introduced during the processes of typesetting and printing.

Mathematical Innovation The purpose of the difference engine was to express a familiar mathematical procedure in machinery. In a curious reversal of this process, Babbage was led to investigate the mathematical properties of some novel functions that had been suggested to him by a consideration of some modifications that could be made to the machine. In 1822, he explained to the Astronomical Society how he had been led to this discovery.[23]

The engine was designed to compute tables which had a constant last difference. Many important functions, particular trigonometrical functions, did not have this property, but could nevertheless be approximated over suitably chosen intervals by functions which did have a constant difference. A disadvantage of this procedure was that, at the boundaries between these intervals, the constant difference in use would have to be manually altered to the value appropriate for the next interval. Babbage considered that this could be a significant source of error, and therefore looked for ways to avoid this procedure.

For some functions, he found that there was a mathematical relationship between the value to be calculated and one of the later differences. For example, it can be shown by simple algebraic manipulation that

$$\Delta_2 \sin(x) = K \sin(x + 1),$$

for a constant value K. In other words, the value of $\sin(x + 1)$ can be calculated from the second difference of the preceding value of the function. Although it seems rather circular, this in fact means that the sine function can be tabulated if an initial value $\sin_1$ and first difference $\Delta_1 \sin_1$ are given, even though it has no column of constant differences. The calculations proceed as follows (where for clarity $\sin_1$ is written in place of $\sin(1)$, and so on):

$$\begin{aligned} \sin_2 &= \sin_1 + \Delta_1 \sin_1, \\ \Delta_2 \sin_1 &= K \sin_2, \\ \Delta_1 \sin_2 &= \Delta_1 \sin_1 + \Delta_2 \sin_1, \\ \sin_3 &= \sin_2 + \Delta_1 \sin_2, \\ &\dots \end{aligned}$$

Babbage observed that if this idea were to be implemented, one requirement would be the ability to pass values between columns, in this case from the result column to the second difference column. He considered ways in which this could be done, and in fact the portion of the engine that was actually assembled in 1832 included mechanism which provided the ability to transfer a single digit between these two columns.[24]

[22] Lardner (1834), p. 283.

[23] Babbage (1822a).

[24] Collier (1970), pp. 111–112.

Babbage then noticed that connecting the columns of the engine in this way would enable it to "produce tables of a new species altogether different from any with which I was acquainted".[25] As his first example of such a table, he considered one where the second difference was equal to the units figure of a number already computed. This is, of course, a table that could be easily have been computed by the machinery assembled in 1832.

Babbage proceeded to analyze these new functions, and succeeded, with some difficulty, in deriving formulae that would generate the same numerical sequences. He commented later that this episode provided the first ever "example of analytical enquiries, suggested and rendered necessary by the progress of machinery adapted to numerical computation".[26] He was further surprised to discover that one of the functions he investigated was related to enquiries that he had made years previously in the course of an investigation into the problem of describing knight's tours on a chessboard.

A further example of a table that Babbage claimed the engine would be able to compute was a series of cube numbers, "subject to this condition, that whenever the number 2 occurred in the tens' place, that and all the succeeding cubes should be increased by ten".[27] This example appears to introduce a new requirement, namely the ability for the machine to detect the occurrence of a 2 and to react accordingly. Unfortunately, Babbage did not give a detailed explanation of how the engine might have performed this task.

Babbage attached great significance to these discoveries,[28] even giving a detailed example of the phenomenon in *The Economy of Machinery and Manufactures*, an unexpected detail in a book intended for the general reader.[29] He was struck in particular by the contrast between the ease with which such new tables could be mechanically computed and the difficulty of finding an analytical solution. This led him to reflect that the ability to perform mechanical calculation could produce useful results even when no theoretical account was given, and also that "one of the first effects of machinery adapted to numbers, has been to lead us to surmount new difficulties in analysis".[30]

2.4 The Mechanical Notation

In the course of his early work on the design of the Difference Engine, Babbage reflected on his working practices, and formulated a explicit account of the process

[25] Babbage (1822a), p. 312.

[26] Babbage (1826b), p. 217.

[27] Babbage (1823), p. 125.

[28] Baily (1823), p. 419, describes Babbage as considering that the "mechanical contrivances" embodied in the engine were of "a secondary kind" compared to these theoretical results.

[29] Babbage (1835b), p. 198.

[30] Babbage (1823), p. 127.

of "mechanical invention", a process which he broke down into three main stages. Given a description of what the proposed machine was intended to do, and taking advantage of any natural divisions such as that between calculation and printing in the Difference Engine, Babbage recommended that the inventor start with what appeared to be the most difficult part. Then,

> [a] kind of analysis of it must be made, and it will be subdivided into a number of different movements, some of which must be executed simultaneously, others in succession; some actions must take place at regular, others at irregular, periods.[31]

Some means to implement each of these individual movements should then be found, and they should all be assembled into a complete design without regard to elegance or mechanical efficiency. At this stage, the only requirement was "that supposing them all executed with perfect accuracy and supposing no flexure nor friction in the materials the machine would do its work". The inventor could then begin the second stage of the process, which "ought to consist almost entirely of simplification", and Babbage gave a number of detailed heuristics which could be applied to help spot ways of simplifying complex mechanical designs. Once the design had been simplified as much as possible, the third stage of Babbage's method involved selecting, or if necessary developing, the mechanisms to implement the new, simplified design.

To indicate the level of simplification that could be hoped for at the second stage, Babbage stated that his initial design for the prototype engine had consisted of 96 wheels and 24 axes, but that after simplification he was able to reduce this to 18 wheels and three axes. Lardner gave a similar account some years later of an instance where Babbage had managed dramatically to reduce the number of revolutions of a particular axis required to perform a specific task, indicating that simplification could be applied to the functioning as well as to the structure of the mechanism.

In the course of this work, Babbage found that the traditional method of using drawings to describe machinery was inadequate. A drawing could only represent the state of a machine at one instant, and so provided little assistance in understanding the sequences of movements involved in a complex mechanism or in working out the appropriate timing of the movements of its interacting parts. Babbage rejected as impractical the idea of producing a series of drawings of successive states of the machine, and believed that natural language was too verbose and ambiguous to be used. Instead, "being convinced from experience of the vast power which analysis derives from the great condensation of meaning in the language it employs", he decided "to have recourse to the language of signs".[32]

Unlike existing symbolic languages such as that of algebra, the notation Babbage developed was partly graphical: the machine to be analyzed was represented by a two-dimensional diagram containing various textual annotations. A diagram began with an enumeration of the moving parts of the machine in question, which were labelled to enable easy cross-reference with the drawings of the machine. No attempt

[31] Babbage (1822c), p. 24.

[32] Babbage (1826a), p. 251.

was made in the notation to describe the actual form of the parts, although a number of their properties could be listed, such as their velocity. The names of the parts were listed in a row across the top of the diagram, and each part had a vertical 'indicating line' below it.

Arrows were drawn between the indicating lines to show a second major feature of mechanisms, namely the connections between parts by means of which motion was communicated from one to another. Different forms of arrow showed alternative types of connection, indicating whether motion was communicated by attaching one part to another, by friction, by a ratchet-driven mechanism, or a variety of other means.

Most importantly, the notation showed the "succession of the movements which take place in the working of the machine". Babbage assumed that a machine's action was periodic, and that after a certain period of time it would return to its initial state and the sequence of movements would be repeated. The movements of each part were shown by means of various symbols written next to the indicating lines. Thus reading down the indicating line for a part would show its motion through one cycle of machine operation, whereas reading horizontally across all the indicating lines would show the motions of all the parts at a given instant. As Lardner later put it, "it exhibits in a map, as it were, that which every part of the machine is doing at each moment of time".[33]

Lardner devoted several pages of his article to the notation, giving examples of its utility and power. He described it as an aid to "invention and discovery", a claim that had also been made when symbolic notation was introduced into algebra two centuries earlier. Lardner himself drew attention to this comparison, writing that "[w]hat algebra is to arithmetic, the notation we now allude to is to mechanism".[34] He stressed its applicability to the second, simplifying stage of Babbage's method: if the complexity of the whole machine was represented in easily manipulable signs, it would help the inventor to identify possible modifications and simplifications that would otherwise have remained obscure. Finally, in an observation which throws an interesting light on contemporary understanding of the relationships between machines and society, he claimed that the notation could be applied to describe the organization of "an extensive factory, or any great public institution, in which a vast number of individuals are employed, and their duties regulated (as they generally are or ought to be) by a consistent and well-digested system".[35]

Babbage continued to use and further develop the notation while working on the Analytical Engine, but he wrote little else about it and it never became widely known or applied. Nevertheless, it is of interest as a novel attempt to apply a symbolic approach to the design and description of complex processes.

[33] Lardner (1834), p. 313.

[34] Lardner (1834), p. 315.

[35] Lardner (1834), p. 319.

2.5 The Analytical Engine

It is misleading to talk about Babbage's 'Analytical Engine' as if it was a single, definite artefact: the term refers to a projected machine which Babbage described in a series of drawings and other documents over a period of many years, during which time his plans naturally changed considerably. Babbage published no description of the machine until the appearance of his autobiography, and the machine was never built, though a model of certain parts of it was constructed late in Babbage's life.

The Origins of the Engine Babbage referred to a new engine in 1834, in a letter to the Prime Minister, the Duke of Wellington. Babbage explained that difficulties he had encountered in the construction of the Difference Engine had led to him having no access to the engineering drawings of that machine for two years. When he regained possession of the drawings, Babbage said, he "immediately began a re-examination and criticism of every part. The result of this, and of my increased knowledge, has been the contrivance of a totally new engine possessing much more extensive powers, and capable of calculations of a nature far more complicated".[36] However, he gave no specific details of the new machine, and in fact seemed rather pessimistic about his chances of success in persuading the British Government to support its construction.

This re-examination began with a reconsideration of the mechanism necessary to transfer numbers from one column to another. Babbage had examined this ten years earlier, when considering how transcendental functions could be computed on the difference engine, but had put the matter aside while concentrating on the practical problems of building the machine. He now referred to this technique as enabling the machine to "eat its own tail", and settled on a scheme where the columns would be arranged around a large central wheel, enabling numbers to be transferred between them easily.[37]

Computation of functions by this new method would also require the ability to handle negative as well as positive numbers, and to perform multiplications. Both of these requirements went beyond the capabilities of the difference engine, and so Babbage turned his attention to the design of machinery to carry out multiplication and division. This led to two significant developments. Firstly, in order to minimize the time taken, Babbage came up with more efficient methods for adding numbers, using in particular a technique known as *anticipatory carrying*. The mechanism for implementing this was rather extensive, however, and rather than making multiple copies of it for each number column, Babbage was led to the idea of separating the parts of the machine that carried out arithmetical operations from those which simply stored numbers.

Secondly, he came up with what he later called a "tentative" process for carrying out division, by repeatedly subtracting multiples of the divisor from the dividend.

[36] Babbage (1834), p. 6.

[37] This account is largely taken from the account of Babbage's notebook entries summarized by Collier (1970), pp. 116–140.

This process should come to an end when a subtraction caused the remaining value of the dividend to become negative. Babbage designed a mechanism which would enable the engine to detect when this had happened, undo the last subtraction, and proceed to the next part of the calculation.

At this point, towards the middle of 1835, Babbage mentioned the new engine in a letter to the Belgian statistician Quetelet. He described it as being "capable of having 100 variables (or numbers susceptible of change) placed upon it" and being able to perform a range of arithmetical operations on any of these variables. The operations Babbage lists are addition, subtraction, multiplication and division of two numbers, extracting the square root of a number, and reducing a number to zero. By these means, Babbage claimed, "if $f(v_1, v_2, \ldots, v_n), n < 120$, be any given formula which can be formed by addition, multiplication, subtraction, division, or extraction of square root the engine will calculate the numerical value of f",[38] and he went on to give examples of some situations where this capability could be used.

This suggests that Babbage viewed the primary purpose of the new engine as being the evaluation of algebraic formulae. However, he had at this stage no easy way of controlling the great variety of operations that would be required in the evaluation of any significant computation. In 1836, however, he came up with the idea of using punched cards to control the progress of the machine, as Jacquard had done earlier in the century in the development of his automated loom.

This turned out to be last major innovation in the design of the new engine, and in 1837 Babbage described its structure and functioning in a lengthy but incomplete manuscript entitled *On the mathematical powers of the calculating engine*.[39] As the title suggests, Babbage was more concerned to describe the capabilities of the proposed machine than the details of its construction, but nevertheless he did not give a clear statement of its purpose. However, the final section of the manuscript, on the use of the machine, is headed "Of computing the numerical value of algebraic formulae", and the discussion and examples given strongly support the idea that at this period Babbage saw this as the primary application for the machine.

The Structure of the Analytical Engine Despite inevitable later revisions, most of the features described in the manuscript of 1837 remained constant during the years that Babbage worked on the Analytical Engine.[40] Like the Difference Engine, at the top level it was divided into two major components.

> The calculating part of the engine may be divided into two portions:
> First. The *mill* in which all operations are performed.

[38] Babbage (1835a), pp. 12, 13. The apparent discrepancy between the number of variables stored in the machine and the number of function arguments is in Babbage's text.

[39] Babbage (1837b).

[40] According to Allan Bromley, who carried out extensive research on Babbage's unpublished design notations for the engine, the design that emerged around 1838 was the basis for Babbage's refinements and elaborations in the subsequent ten years (Bromley 1982). Bromley's paper gives a detailed physical description of the engine, including the engaging observation that the completed machine would "have been about the size and weight of s small railway locomotive".

> Secondly. The *store* in which all the numbers are originally placed and to which the numbers computed by the engine are returned.[41]

At one level, the top-level structure of the engine and the terminology used by Babbage to describe it can be seen as reflecting a division of labour similar to that employed in the cotton industry, where the raw material was worked on in the mill and the resulting fabrics were removed from and replaced in a storehouse. However, a deeper motivation comes from Babbage's philosophical views on the notation of mathematics, which took as fundamental the distinction between operations and variables, and saw the evaluation of a formula as a question of applying the required operations to the numbers provided. Babbage's distinction between the mill and the store, on this view, simply makes concrete this theoretical distinction. As Lovelace put it some years later,

> [i]n studying the action of the Analytical Engine, we find that the peculiar and independent nature of the considerations which in all mathematical analysis belong to *operations*, as distinguished from *the objects operated upon* and from the *results* of the operations performed upon those objects, is very strikingly defined and separated.[42]

The store consisted of a number of *figure axes*, also known as *variables*, each of which was capable of storing one number. Each axis consisted of 40 *figure wheels* mounted vertically on a single axle. Each wheel was marked with the digits 0 to 9, and could be moved to the desired position by hand, thus allowing manual entry of numbers into the store. A additional wheel recorded the sign of the number stored on the axis; this wheel also had ten positions, of which the even positions were marked with a plus sign and the odd positions with a minus sign. Each axis was identified by a fixed label or variable number, such as V_1, which was displayed above the axis. In addition, below the figure wheels each axis had

> a small square frame [...] in which may be inserted a card to be changed according to the nature of the calculation directed. On this card is written that particular variable or constant of the formula to be computed whose numerical coefficient and sign are expressed on the wheels above it.[43]

If, for example, it was necessary to calculate the value of the formula $\sqrt{a^2 + x^2}$ from the values of a and x, it might be decided to place the values of a and x on the variable axes V_2 and V_3, respectively, and the final value on V_6. In this case, the symbols a, x and $\sqrt{a^2 + x^2}$, using the mathematical notation relevant to the problem at hand, would be written on cards and placed beneath the variables V_1, V_2 and V_6. Clearly, this technique had no effect on the operation of the machine, and was purely intended as an aid to its human users.

Numbers could be transferred from the store to the mill, where they would be operated upon, and the results of these operations could then be transferred back to the store. The engine was designed to perform arithmetical operations, and in particular addition, subtraction, multiplication and division. Babbage occasionally

[41] Babbage (1837b), p. 15.

[42] Lovelace (1843), p. 692.

[43] Babbage (1837b), p. 23.

referred to other possible operations, such as the extraction of roots, but described only the four basic operations in detail.

The detailed progress of an operation such as the addition of two numbers was controlled by cylinders known as *barrels*. Studs were attached to the barrels in a number of vertical columns, and as the barrels rotated, the studs caused different parts of the mechanism of the mill to be brought into action, in a manner similar to that employed in a music box or barrel organ. The barrels were capable of more than simply controlling a sequence of actions, however, and various mechanisms enabled the mill to detect and respond to situations that might arise in the course of carrying out an operation.

The most important such situation was when a larger number was subtracted from a smaller, giving rise to a negative result. This particular event was known as *running up*; it was, in effect, a carry that, because of the numerical representation used, ran off the end of the figure axes, hence the name. The machine contained "a lever on which the *running up* warning acts and this lever governs many parts of the engine according as the circumstances demand".[44] This feature gave the engine the capacity to detect and react to certain events that took place during the course of a computation by affecting the way in which barrel selected the next column of studs to be applied to the mechanism.[45]

The course of a computation was determined by the sequence of operations that was to be performed. Operations were selected by means of *operation cards*, which would be presented in order to the mill. These cards were to be perforated pieces of pasteboard or thin metal similar to those used in Jacquard's automated loom. A card would be pressed against an array of levers; the unperforated positions in the card would engage a particular set of levers, which would then control the progress of the specified operation. As Babbage commented, "by arranging a string of cards with properly prepared holes any series of orders however arbitrary and however extensive may be given through the intervention of these levers".[46]

The process of reading an operation card and carrying out the specified operation was carried out under the control of the barrels. When an operation was requested, the numbers to be operated on would be fetched from the store and placed on two figure axes in the mill; on completion of the operation, the result would be returned to the store. In the same way as operation cards selected the operations that were to be performed, *variable cards* selected the quantities that would be operated on. At the start of an operation, the mill would request numbers from the store; the numbers selected would be those on the figure axes specified by the next variable cards to be presented to the store. At the conclusion of an operation, the result would be placed on a figure axis specified by the next variable card.

---

[44] Babbage (1837b), p. 17.

[45] Allen Bromley describes in detail the conditional operations implemented within the mill, and gives Babbage the credit for originating "the whole concept of a conditional sequence of operations in a machine, and in particular of a conditional dependence on the outcome of previous actions of the machine" (Bromley 1982).

[46] Babbage (1837b), p. 20.

The independence of the operation and variable cards is a notable feature of the design of the Analytical Engine. The cards controlled the operations of separate parts of the machine, the mill and the store, respectively. Because the control levers differed, the two types of card were to be of different sizes and to have different patterns of perforation. An operation card could be used to operate on different data at different times, as the numbers used on each occasion would be determined solely by the variable cards that were presented to the store.

Babbage saw that particular operations and sequences of operations might need to be carried out repeatedly in the course of a computation. He therefore proposed that special cards, known as *combinatorial cards*, could be inserted in the sequence of operation cards. The function of the combinatorial cards was:

> To govern the *repeating apparatus* of the operation and of the variable cards and thus to direct at certain intervals the return of those cards to given places and to direct the number and nature of the repetitions which are to be made by those cards.[47]

Subsequent entries in Babbage's notebooks described a set of special counting wheels which would record the number of times operations should be repeated. When a combinatorial card was encountered, the number on these wheels would be reduced by one and the operation cards backed up as far as required. When the number on the wheels reached zero, the combinatorial card would be ignored and the computation would proceed with the next operation card. Babbage's first idea involved backing up the operation cards to the beginning, but he later proposed that the cards could be backed up to a predetermined spot in the sequence, or by a certain number of cards.[48]

A different further method of repeating operations, suggested in the early 1840s, was for each operation card to contain an index number which would specify how many times the operation on that card should be carried out before the next operation card was read.[49] The sole purpose of this suggestion seems to have been to minimize the number of operation cards that needed to be prepared for a calculation.

A further type of cards were the *number cards*; these were perforated in such a way as to make it possible to transfer a number from the card to a figure axis. An advantage of a number card as opposed to the manual entry of a number on an axis was the possibility of reusing the card whenever necessary. Babbage also considered that the cards could be used in cases where a calculation exceeded the number of variables provided in the store, and the use of number cards in a calculation was therefore to be controlled by the variable cards.[50]

The mechanical description of the engine concluded with a number of proposed auxiliary devices, such as a card punch which would enable numbers to be punched onto the number cards in the first place. Other planned output devices included an apparatus for printing results on paper or copperplate, and Babbage also mentioned

[47] Babbage (1837b), p. 21.

[48] The relevant notebook entries have been summarized by Collier (1970), pp. 196–200.

[49] See the references listed by Collier (1970), p. 199.

[50] Babbage (1837b), p. 27.

the possibility of producing a graphical representation of results by means of a curve drawing apparatus.

The Mathematical View In the 1837 manuscript, Babbage gave few details of how the engine would be used. He seems to have envisaged a loose division of labour between the "employer of the engine", who would specify the operation and variable cards to be used for a computation along with any required constants, and a number of "attendants" and a "superintendent" who would actually operate the engine.[51] As well as simple sequences of operations, it was proposed that users could order the repetition of groups of operations using the combinatorial cards, but Babbage does not seem to have envisaged at this point that users could specify alternative sequences. The running up mechanism provided conditional execution at certain points in the *internal* implementation of operations, in particular division, but this capability was not made available to the user.

In 1840, Babbage travelled to Turin, where he presented his ideas to a number of Italian scientists. Following this visit, the mathematician L.F. Menabrea wrote an account of the engine, which was published in 1842. A translation of this article, which was written in French, was published by Ada Lovelace in the following year. Lovelace added some extensive notes of her own to the translation. Both Menabrea and Lovelace had discussed the engine at length with Babbage, and their writings form an important source of information about Babbage's ideas. In particular, they included a number of detailed descriptions of the operations that would be required to perform various computational tasks, derived from examples that Babbage had developed between 1837 and 1840, but never published. These examples included the tabulation of polynomials and iterative formulae, and the solution of simultaneous equations using Gaussian elimination.[52]

Unlike Babbage, Menabrea and Lovelace paid little attention to the mechanical details of the Engine, focusing instead on those aspects of it that would be important to someone wanting to use the completed machine to perform calculations. Lovelace characterized this situation by drawing a distinction between the "mechanical" and "mathematical" views of the engine. The mathematical view was that of someone "taking for granted that mechanism is able to perform certain processes, but without attempting to explain *how*", and describing instead "the manner in which analytical laws can be so arranged and combined as to bring every branch of that vast subject within the grasp of the assumed powers of mechanism".[53] She emphasized that the two views were complementary, and commented that "the same mind might not be likely to prove equally profound or successful in both".

Menabrea and Lovelace gave just enough mechanical detail about the engine to enable the reader to understand how it might be used to perform calculations. Rather than going into the details of how division was performed, for example, Menabrea wrote that "we must limit ourselves to admitting that the first four operations of

[51] Babbage (1837b), p. 52.

[52] Bromley (1982), p. 215.

[53] Lovelace (1843), p. 700.

arithmetic … can be performed in a direct manner through the intervention of the machine". The general pattern for the execution of an arithmetical operation was the following:

> When two numbers have been thus written on two distinct columns, we may propose to combine them arithmetically with each other, and to obtain the result on a third column.[54]

To carry out an operation it was necessary first to configure the mill to perform that operation, and then to transfer the required numbers from the store to the mill; when the operation was completed, the answer should be transferred back to the store. Menabrea explained how operation and variable cards were used to carry out these tasks automatically. However, when giving detailed examples of calculations, Menabrea followed the example of Babbage's unpublished work and rather than describing the actual cards required, he listed the operations carried out, in a "three address" notation of a kind that would become familiar a century later.

It is worth noting, however, that this notation simplifies the actual behaviour of the engine. In order to deal with the complexities of adding and subtracting signed numbers, Babbage distinguished what he called the 'algebraic' sign of a number from its 'accidental' sign. In the formula $P - Q$, for example, the algebraic signs of P and Q are $+$ and $-$, respectively; however, their accidental signs could be $+$ or $-$, depending on whether the corresponding numbers in the store were positive or negative. Depending on the combination of these signs, the actual operation to be performed might differ from that specified on an operation card: in the formula above, if the accidental sign of Q was negative, for example, its absolute value would be added to that of P rather than subtracted from it.

Whereas the tabular notation suggests that addition and subtraction operations were specified by a single card, Babbage had actually stated that:

> For the processes of addition and subtraction two operation cards only are necessary. One of these is required for each quantity, and assigns to it the algebraic sign of the quantity.[55]

This approach allowed formulae such as $-P - Q$ to be handled in a consistent way, and also allowed repeated sums to be formed without moving partial answers back and forth between the mill and the store, making a potential saving in computation time.

For his first example, Menabrea described the operations required to calculate the value of x in the following pair of simultaneous equations:

$$mx + ny = d,$$
$$m'x + n'y = d'.$$

The series of operations specified by the cards together with the results of each operation were given in the tabular format shown in Table 2.3; the formula used to calculate x is displayed in the bottom right-hand cell of the table. Each line in this table represents a single operation and shows the two variables holding the numbers

[54] Menabrea (1842), p. 676.

[55] Babbage (1837b), p. 49.

Table 2.3 A computation on the Analytical Engine

Number of the operations	Operation cards	Variable cards		Progress of the operations
	Symbol of operation	Columns on which operations are performed	Columns which receive results	
1	×	$V_2 \times V_4 =$	$V_8 \ldots$	$= dn'$
2	×	$V_5 \times V_1 =$	$V_9 \ldots$	$= d'n$
3	×	$V_4 \times V_0 =$	$V_{10} \ldots$	$= n'm$
4	×	$V_1 \times V_3 =$	$V_{11} \ldots$	$= nm'$
5	−	$V_8 - V_9 =$	$V_{12} \ldots$	$= dn' - d'n$
6	−	$V_{10} - V_{11} =$	$V_{13} \ldots$	$= n'm - nm'$
7	÷	$\frac{V_{12}}{V_{13}} =$	$V_{14} \ldots$	$= x = \frac{dn'-d'n}{n'm-nm'}$

to be operated on as well as the variable that would receive the calculated result. The final column shows the mathematical meaning of the result variable, using the notation of the original problem. These are the formulae that would be written on the cards beneath the columns V_8 to V_{14}.

To execute this calculation on the Analytical Engine, a set of operation cards would have to be derived from the list of operations given in the second column of the table and a set of variable cards from the entries in the third and fourth columns. There was no need, however, to reconfigure the mill when the same operation was repeated, and therefore no need for a sequence of operation cards requesting the same operation. Menabrea observed that the number of operation cards required could be reduced if the machine included "an apparatus which shall, after the first multiplication, for instance, retain the card which relates to this operation, and not allow it to advance so as to be replaced by another one, until after this same operation shall have been four times repeated".[56] To reflect this, he used a revised tabular format, which included an additional column numbering the operation cards that were physically required.[57] He did not give any details of the postulated counting mechanism, however, nor do the tables include an explicit statement of the number of times any particular operation card is to be used.

A second modification to the notation concerned the reuse of numbers stored on the variable columns. The normal behaviour of the engine was to erase a variable when the number on it was transferred to the store. After operation 1 in Table 2.3, for example, the number stored on variable V_4 would be erased and would no longer be available for use by operation 3. It was possible to use an alternative form of variable card, however, which specified that when a number was transferred to the mill it would also be preserved in the store. Menabrea therefore extended the notation with an additional column headed "indication of the new columns on which the variables

[56] Menabrea (1842), p. 679.

[57] Menabrea (1842), p. 681.

are written"; for the first operation, the entry in this column would have read "V_2 on V_2, V_4 on V_4", indicating that both columns V_2 and V_4 kept the number stored on them after it was called into the mill for the multiplication.[58]

Menabrea gave further examples of calculations in which the variable columns were taken to represent, not simply the numbers in an algebraic formula, but the coefficients of the terms in a series such as $\Sigma_i \cos^i x$. This use of the engine, although it required a different approach to the planning of a calculation, did not require any additional features of the engine or notation.

As well as the ability to repeat cycles of operations, the need for the engine to be able to perform different sequences of operations in different circumstances was recognized. Menabrea introduced this requirement by considering "certain functions which necessarily change in nature when they pass through zero or infinity", and he described how the machine might stop and ring a bell to summon an operator when this happened. However:

> If this process has been foreseen, then the machine, instead of ringing, will so dispose itself as to present to the new cards which have relation to the operation that is to succeed the passage through zero and infinity. These new cards may follow the first, but may only come into play contingently upon one or other of the two circumstances just mentioned coming into place.[59]

To illustrate this capability, Menabrea considered the evaluation of terms of the form ab^n, where the number of multiplications to be performed depends on the value of the exponent n. He described how the value of n could be placed on a certain "registering-apparatus" and reduced by one each time a multiplication took place. The engine would detect when this value reached zero, and "pursuing its course of operations, will order the product of b^n by a".[60]

In a second example, he considered a calculation where it was required to tell when the two expressions $m + q$ and $n + p$ were equal. Menabrea wrote that:

> For this purpose, the cards may order $m + q$ and $n + p$ to be transferred to the mill, and there subtracted one from the other; if the remainder is nothing [...] the mill will order other cards to bring to it the coefficients Ab and Ba, that it may add them together.

However, no details were given of the mechanisms that would enable the engine to perform this kind of conditional execution of operations, nor of the details of the cards that would be required to specify them. However, Menabrea did draw the following general conclusion about the scope and power of this procedure:

> This example illustrates how the cards are able to reproduce all the operations which intellect performs in order to attain a determinate result, if these operations are themselves capable of being precisely defined.

[58] In her translation, Lovelace slightly altered the heading of this new column and the notation used to express the preservation of the variables.

[59] Menabrea (1842), p. 685.

[60] Menabrea (1842), p. 686.

Translating Analysis As discussed at the beginning of this section, it appears that Babbage originally saw the Analytical Engine as a device to evaluate algebraic formulae. However, over the years a more expansive interpretation seems to have evolved, articulated around the notion that the engine was, in a very literal sense, a translation of the language of analysis into machinery. This idea was developed on two levels. On the material level, as pointed out above, the physical structure of the engine reflected Babbage's view of the structure of the mathematical universe, with the mill and the store corresponding to the basic categories of operations and variables, respectively.

The various operations that the engine could perform were carried out by largely independent pieces of machinery in the mill. To execute a particular operation, the perforations in an operation card would cause the machinery for that operation to be engaged. So in a very immediate way, mathematical operations were represented by machinery, a situation described by Lovelace as one in which "matter has been enabled to become the working agent of abstract mental operations".

However, as the engine was only planned to implement four basic arithmetical operations, it could be asked, as Lovelace did, whether the "executive faculties of this engine" were "*really* even able to *follow* analysis in its whole extent?" She answered this question in the affirmative, by listing the analytical procedures that the engine could carry out.

On a more abstract level, however, an equivalence was proposed between the engine and the notion of a function itself. Lovelace wrote that

> the engine may be described as being the material expression of any indefinite function of any degree of generality and complexity,

while receiving by means of the cards "the impress of whatever *special* function we may desire to develop or to tabulate".[61]

The cards themselves were described as literal translations of the corresponding algebraic formulae. Menabrea expressed this point of view very clearly, writing that "the cards are merely a translation of algebraical formulae, or, to express it better, another form of analytical notation".[62] Thus, to take a very simple example, the translation of the formula $x \times y$ would require one operation card corresponding to the multiplication symbol, and two variable cards denoting the variables in the store holding the values of x and y.

Another notion of generality was brought out in Babbage's comment that "the operation cards partake of that generality which belongs to the algebraic signs they represent", and Menabrea stated similarly that "the cards will themselves possess all the generality of analysis, of which they are merely a translation". One aspect of this generality had to do with the distinction between variables and numbers. Menabrea pointed out that just as "a formula simply indicates the number and order of the operations requisite for arriving at a certain definite result", so the corresponding

[61] Lovelace (1843), p. 691.

[62] Menabrea (1842), p. 688.

set of operation and variable cards "will serve for all questions whose sameness of nature is such as to require nothing altered excepting the numerical data".[63]

The combination of the claim that the cards translated formulae and the fact that the engine only implemented a limited range of operations raised certain problems, however. For example, consider the case of formulae of the form b^n, where one number is raised to an integral power. If the idea of translation is to be preserved, this formula cannot be interpreted as containing an exponential operation, as the engine contained no such operation. Rather, it should be interpreted as containing a reference to a multiplication operation, with the exponent n indicating how often the multiplication should be carried out. As Menabrea put it, as the cards were to be a translation of the analytical formula, the number of operation cards required to evaluate such terms should be the same whatever the value of n, even though differing values of n indicated that the multiplication operation must be performed a different number of times.

While this interpretation preserves the analogy between operation symbols and operation cards, however, it is difficult to square with the idea that variables simply store quantities that are used in calculations. In the formula b^n, n is not a straightforward symbol of quantity: the value of n is not used directly in calculation, but is rather used to *control* the progress of the calculation, specifying how often the quantity represented by b should be multiplied by itself. Lovelace blamed this on the deficiencies of current analytical notation, writing that "figures, the symbols of *numerical magnitude*, are frequently also the *symbols of operations*, as when they are the indices of powers",[64] and she went on to describe how numbers representing operations and those representing quantities are kept separate in the store, although it appears that this distinction was a matter of conventional usage, rather than being enforced mechanically.

2.6 The Science of Operations

The distinction between operations and the objects operated on was fundamental to the design of the Analytical Engine. This reflected current philosophical thinking about mathematics, as Lovelace pointed out in the first of the notes she added to her translation of Menabrea's account. She went on to give a general account of this new outlook, broadening its scope away from the purely mathematical.

She first offered an abstract definition of what an operation was, as "*any process which alters the mutual relation of two or more things*". Although it was inspired by mathematics, this definition was meant to be applicable to "all subjects in the universe", and Lovelace saw the study of operations as having a completely general scope:

[63] Menabrea (1842), pp. 685, 688.

[64] Lovelace (1843), p. 693.

> the science of operations, as derived from mathematics more especially, is a science of itself, and has its own abstract truth and value; just as logic has its own peculiar truth and value, independently of the subjects to which we may apply its reasonings and processes.[65]

Lovelace did not give details of other applications of the science of operations, although she did speculate that if "the fundamental relations of pitched sounds in the science of harmony" could be expressed as operations of the requisite sort and adapted to the mechanism of the engine, then it might be able to "compose elaborate and scientific pieces of music of any complexity or extent".[66] Later, she referred to "*symbolical results*" being generated by the machine, but without giving a clear example of what was meant by this.[67]

For Lovelace, therefore, the Analytical Engine was not simply a machine which evaluated formulae. In her view, it had a more general significance as "an *embodying of the science of operations*, constructed with particular reference to abstract number as the subject of those operations", in contrast with the Difference Engine, which was "the embodying of *one particular and very limited set of operations*".[68]

By "the science of operations", Lovelace appears to have meant some kind of general study of the way in which the operations required in calculations could be specified. The engine was designed primarily to execute a sequence of operations, and given the assumption that at each step in the calculation "the *same operation* would be performed on different *subjects of operation*" she found it natural to use a notation which represented the sequence of operations, but not the variables required at each step. For a simple example, she notes that she required

> In all, seven multiplications to complete the whole process. We may thus represent them:

$$(\times, \times, \times, \times, \times, \times, \times), \quad \text{or} \quad 7(\times).$$

A more complex formula, which involved finding the quotient of two terms, was represented as follows:

$$\{7(\times), 2(\times), \div\}, \quad \text{or} \quad \{9(\times), \div\},$$

suggesting that Lovelace was not concerned whether the sequence of operations manifested the structure of the original formula or not.

In a later note, Lovelace gave a more complex example, in which she expanded and commented on this notation. The example was to multiply two trigonometrical series with coefficients A_n and B_n by calculating the coefficients C_n of the terms

[65] Lovelace (1843), p. 693; emphasis in original.

[66] Lovelace (1843), p. 694.

[67] Lovelace (1843), p. 695; emphasis in original.

[68] Lovelace (1843), p. 694; emphasis in original.

of the resulting series. The required coefficients C_n could be calculated using the following formulae:[69]

$$\begin{aligned} C &= BA + \frac{1}{2}B_1A_1, \\ C_1 &= BA_1 + B_1A + \frac{1}{2}B_1A_2, \\ C_n &= BA_n + \frac{1}{2}B_1 \cdot (A_{n-1} + A_{n+2}). \end{aligned}$$

Lovelace then gave the following expression, representing "the successive sets of operations for computing the coefficients of $n + 2$ terms":

$$(\times, \times, \div, +), (\times, \times, \times, \div, +, +), n(\times, +, \times, \div, +),$$

observing that "the brackets ... point out the relation in which the operations may be *grouped*, while the comma marks *succession*". The operations in the first set of brackets compute the term C, those in the second the term C_1 and those in the third any subsequent term; it is not altogether straightforward to check the equivalence between the sequences of symbols and the operations that would be performed to evaluate the formulae.

This expression contains what Lovelace termed a *recurring group* of operations, or a *cycle*, and she observed that "[w]herever a general term exists, there will be a *recurring group* of operations". Further, "[i]n many cases of analysis there is a *recurring group* of one or more *cycles*; that is a *cycle of a cycle*, or a *cycle of cycles*". For example, if it was required to form the quotient of two polynomials of order p, the operations required to form the coefficient of one term of the quotient are

$$\big\{(\div), p(\times, -)\big\},$$

and so the operations required to calculate n terms of the quotient could be represented as

$$n\big\{(\div), p(\times, -)\big\}.$$

Lovelace then took a further step, adapting "some of the notation of the integral calculus" to her evolving notation. She rewrote the previous expression in what was intended to be an equivalent form as:

$$\Sigma(+1)^n\big\{(\div), \Sigma(+1)^p(\times, -)\big\}.$$

Lovelace explains this notation by writing that "p stands for the variable; $(+1)^p$ for the function of the variable, that is for ϕp; and the limits are from 1 to p, or from 0 to $p - 1$, each increment being equal to unity".[70]

This notation is further applied to the case of *varying cycles*, where "each cycle contains the same group of operations, but in which the number of repetitions of the group varies according to a fixed rate, with every cycle". What Lovelace appears to have in mind here is the case of a cycle within a cycle, where the inner cycle is

[69] Lovelace (1843), pp. 715–716; A, B and C can here be read as A_0, B_0 and C_0.

[70] Lovelace (1843), p. 719.

repeated a different number of times at each repetition of the outer cycle. Without the new notation, the outer cycle must be written out in full, as in the following example:[71]

$$p(1, 2, \ldots, m), (p-1)(1, 2, \ldots, m), (p-2)(1, 2, \ldots, m), \ldots, (p-n)(1, 2, \ldots, m).$$

Lovelace claimed that this could be equivalently written as

$$\Sigma p(1, 2, \ldots m), \text{ the limits of } p \text{ being from} p - n \text{ to } p,$$

but this idea was not explored or explained further, and it was left unclear how the variable n of the outer cycle could be related to the inner cycle, and how the informal description of the limits might have been symbolized.

Lovelace's notes, then, contain some suggestions for a potentially sophisticated notation for expressing complex sequences of operations, and in particular those containing recurring groups of operations. These ideas were not integrated into the more comprehensive notation that she used for planning calculations, however. Her final example, a "diagram for the computation by the Engine of the Numbers of Bernoulli", used a tabular format similar to that used by Menabrea, but with a greater number of annotations describing the mathematical progress of the calculation. On this diagram, cycles and cycles of cycles were indicated by means of brackets in the margins of the table indicating the recurring groups of operations, but the notation gave no indication of how often a particular cycle would be repeated.

Also, the relationship between the notation and those aspects of the machine that would control the execution of cycles was left rather vague. Lovelace stated that the backing mechanism was to be used to execute the operations in a cycle, but her explanation of how the extent of the cycle or the number of required recurrences were communicated to the engine was restricted to the following suggestion:

> Σ, in reality, here indicates that when a certain number of cards have acted in succession, the prism over which they revolve must *rotate backwards*, so as to bring those cards into their former position; and the limits 1 to n, 1 to p, etc., regulate how often this backward rotation is to be repeated.[72]

2.7 The Meanings of the Analytical Engine

Mechanizing the Mind As with the Difference Engine, an important aspect of the Analytical Engine was its demonstration of how an apparently mental process, that of evaluating algebraic formulae, was in fact mechanizable. Menabrea had argued that mathematics could be divided into a mechanical part, "subjected to precise and invariable laws, that are capable of being expressed by the operations of matter",

[71] Lovelace (1843), p. 719; the sequence $(1, 2, \ldots, m)$ here denotes a group of unspecified operations.

[72] Lovelace (1843), p. 720.

and a part "demanding the intervention of reason" which "belongs more specially to the domain of the understanding". Because it implemented not simply the four basic arithmetical operations, but also the operations that were involved in applying the sequence of operations required to evaluate a formula, the Analytical Engine was of a much wider scope than its predecessor. As we have seen, Menabrea considered it to be a "system of mechanism whose operations should themselves possess all the generality of algebraical notation".[73]

The greater capabilities of the Analytical Engine brought with them a temptation to use increasingly anthropomorphic language in talking about it, as Babbage noted apologetically in 1837:

> In substituting mechanism for the performance of operations hitherto executed by intellectual labour it is continually necessary to speak of contrivances by which certain alterations in parts of the machine enable it to execute or refrain from executing particular functions. The analogy between these acts and the operations of mind almost forced upon me the figurative employment of the same terms. They were found at once convenient and expressive and I prefer continuing their use rather than substituting lengthened circumlocutions.
> For example, the expression 'the engine *knows*, etc.' means that one out of many possible results of its calculations has happened and that a certain change in its arrangement has taken place by which it is compelled to carry on the next computation in a certain appointed way.[74]

In particular, it seems to have been the apparent ability of the Engine to choose between alternative courses of action which raised questions about what the most appropriate way to describe it was. Menabrea tended to emphasize its mechanical nature, and to stress that it could only proceed in a determinate manner. He pointed out that Babbage had had to devise a method for carrying out division which did not employ the usual "method of guessing indicated by the usual rules of arithmetic", and wrote that the machine "must exclude all methods of trial and guesswork, and can only admit the direct processes of calculation. It is necessarily thus; for the machine is not a thinking being, but simply an automaton which acts according to the laws imposed upon it".[75]

At times, Lovelace expressed herself with a greater freedom, however, and her comment on Babbage's comments above was that "[t]his must not be understood in too unqualified a manner. The engine is capable, under certain circumstances, of feeling about to discover which of two or more possible contingencies has occurred, and of then shaping its future course accordingly".[76] This is reminiscent of the passage to which the above quote from Babbage is a footnote, in which he describes his so-called "anticipatory carriage" mechanism by saying that he has designed the machine to "foresee" the effect of a carry operation in addition.

At other times, however, Lovelace too emphasized the machinic nature of the Engine, as in the following remark, later to be discussed in detail by Turing:

---

[73] Menabrea (1842), pp. 669, 674.

[74] Babbage (1837b), p. 31, footnote.

[75] Menabrea (1842), p. 675.

[76] Lovelace (1843), p. 675, footnote.

> The Analytical Engine has no pretensions whatever to *originate* anything. It can do whatever we *know how to order it* to perform.[77]

However, this did not mean that every step and result had to be known in advance: it is "by no means necessary that a formula proposed for solution should ever have been actually worked out, as a condition for enabling the engine to solve it. Provided we know the *series of operations* to be gone through, that is sufficient".[78]

Time and Economy Babbage paid great attention in the design of the Analytical Engine to making its operations as fast as possible, going so far as to say that "the whole history of the invention has been a struggle against time"; he gave a detailed account of the way in which the timing of the machine had been estimated and optimized.[79] This applied at the most detailed levels of the mechanism, as well as in more general concerns, such as the choice of the algorithm to be implemented for the basic operations of multiplication and division.

Menabrea referred to the "economy of time" that the machine enabled, quoting Babbage's estimate that the multiplication of two 20-digit numbers could be completed in three minutes. He further identified "economy of intelligence" as one of the machine's advantages, writing that "the engine may be considered as a real manufactory of figures".[80]

Lovelace reinforced this perception of a connection between the machine and contemporary views about political economy when describing its productive nature:

> In the case of the Analytical Engine we have undoubtedly to lay out a certain capital of analytical labour in one particular line; but this is in order that the engine may bring us in a much larger return in another line.[81]

The means by which this profit was to be realized was made clear by drawing an analogy between the internal structure of the engine and current practices of labour organization:

> The columns which receive [intermediate and temporary combinations of the primitive data] are rightly named *working Variables*, for their office is in its nature purely *subsidiary* to other purposes.[82]

In both its use and its internal structure, then, the Analytical Engine was viewed as having a close relationship with the capitalist economy of the factory system, and embodying its labour relations, just as its predecessor had been.

Avoidance of Error A significant advantage of the engine was what Menabrea called its "rigid accuracy", a property which stemmed not only from the supposed

[77] Lovelace (1843), p. 722.

[78] Lovelace (1843), p. 721.

[79] Babbage (1837b), pp. 39; 55–61.

[80] Menabrea (1842), p. 690.

[81] Lovelace (1843), p. 698.

[82] Lovelace (1843), p. 707.

infallibility of mechanical processes but also the fact that it required "no human intervention during the course of its operations".[83] However, there were still many points in the process of using the engine for a complete calculation at which the intervention of human operators would have been required. Babbage pointed out that users would have to insert potentially large amounts of numerical material into the machine, and also to compose the sequences of operation and variable cards required to translate a particular formula and make sure that they were correctly presented to the engine. While noting this, Lovelace claimed that there was nevertheless less chance of error in these tasks than in purely numerical work.[84]

Babbage suggested a number of techniques for verifying the correctness of the engine's results. For example, he suggested that all numbers entered into it should be immediately printed out, so that a subsequent check could be made that the correct numbers had been entered. For what he considered to be the more difficult task of checking the correctness of the cards used, a number of approaches to verification were suggested, including the use of coloured cards which would provide a visual check that the structure of the cards matched that of the formula.

He also suggested running test cases to verify a formula, using "such numerical values to the constant quantities as shall render its value easily computed by the pen", arguing rather optimistically that

> If trials of three or four simple cases have been made, and are found to agree with the results given by the engine, it is scarcely possible that there can be any error among the cards.[85]

Further ideas put forward included the suggestion that a computation could be carried out using two different, but mathematically equivalent, formulae and the results compared, and the use of a machine which would translate a set of cards back into the analytical formula represented before printing it out for checking. Finally, Babbage suggested that the use of combinatorial cards to reduce the number of cards required would simplify the task of preparing the cards for a computation, and even speculated that "the formula printing-machine might by some improvements itself ultimately work out many of such algebraic developments".[86]

Natural Theology In the 1830s, the Earl of Bridgewater sponsored the publication of eight treatises *On the Power, Wisdom, and Goodness of God, as manifested in the Creation*. Written by a number of well-known scientists and divines, the treatises were intended to demonstrate how the results and theories of recent natural science were compatible with an over-arching view of God as a designer. Babbage detected a 'prejudice' in the published treatises, particularly one written by William Whewell on physics and astronomy, to the effect "*that the pursuits of science are unfavourable to religion*" Babbage (1837a). He published his own views on natural theology in what he called an unofficial, 'ninth' Bridgewater treatise.

[83] Menabrea (1842), p. 689.

[84] Babbage (1837b), pp. 51–54, Lovelace (1843), p. 698.

[85] Babbage (1837b), p. 53.

[86] Babbage (1837b), p. 54.

A central part of his argument focused on the question of miracles. This was an important question for natural theology, as the scientific world view, of a clockwork universe of matter which had been set in motion at the creation and was thereafter governed by unchangeable natural laws, seemed to be at odds with the assumed fact of miraculous events as revealed in the Bible.

Babbage advanced a view which purported to explain how miracles could in fact be produced by the operation of purely mechanical laws of nature. This argument was inspired by the properties of the Analytical Engine. Unlike the mechanism of a clock, in which the same processes would take place repeatedly so long as energy was available to drive them, the Analytical Engine could be set up in such as way that after any prespecified length of time its behaviour would change. Furthermore, any number of such changes could be specified. From the human point of view, these changes, coming perhaps after eons of regular behaviour, would appear miraculous, though they were in fact inevitable, being prefigured in the design of the machinery of the universe.

2.8 Conclusions

Babbage was ultimately unsuccessful in his attempt to build automatic computing engines. A small working prototype of the Difference Engine was completed, and portions of the full-sized machine were built, but work on the Analytical Engine did not proceed beyond the production of design drawings.

In the light of the subsequent history of computers, it is interesting to consider the reasons for this failure. Babbage wrote generally of technological innovation that "[i]t is partly due to *the imperfection of the original trials*, and partly to the gradual improvements in the art of making machinery, that many inventions which have been tried, and given up in one state of art, have at another period been eminently successful".[87] Most historians have followed Babbage's suggestion here, and concluded that Babbage's designs made too great a demand on the mechanical and tool-making expertise available at the time. However, as Babbage elsewhere made clear, the early nineteenth century was a time of great technical innovation, and the work carried out on the Difference Engine was itself responsible for significant advances.

An additional factor in Babbage's failure was certainly financial, in that the costs of the development even of the Difference Engine would in no way be met by the savings that its use would lead to. Babbage was aware of this, believing that the costs of the project should be underwritten by the Government rather than private capital, a position he supported by pointing out the national importance of the application of the Engine to fields such as navigation. The Government ultimately withdrew its support, however, being unconvinced that significant savings would follow from the use of the Engine. Some retrospective support for this position can be drawn from

[87] Babbage (1835b), Sect. 324.

the experience of the American Nautical Almanac Office in trying to make use of Scheutz's completed difference engine in 1859.[88]

Babbage as Pioneer Babbage is commonly taken to be a 'pioneer' of the computer, someone whose designs anticipated modern developments. In contrast, this chapter has attempted to describe Babbage's work as it was presented to and understood by his contemporaries.

It is a misleading anachronism to describe the Analytical Engine as a 'computer'. This characterization prevents us from seeing it in its own terms, as Babbage and his collaborators understood it. As argued above, Babbage thought of it primarily as a machine for the numerical evaluation of algebraic formulae, and much of its structure, organization and use can be best understood by thinking of it in these terms. If we think of it as a computer, it is impossible not to notice the ways in which it differs from current designs, and hence to describe it as a limited first attempt, or to wonder at the fact that Babbage did not include certain current design features. This prevents us from understanding it in its own terms, and from forming a just evaluation of Babbage's achievement.

Nevertheless, there are striking similarities between the design of the Analytical Engine and certain features of mid-twentieth century computer architecture. If we conclude, as seems likely, that the later developments were largely independent of the details of Babbage's work, then what is left with is a striking case of design convergence. Bromley points out that because of the inaccessibility of Babbage's design for the Analytical Engine, it is unlikely that direct influence could have been anything other than superficial, and concludes: "I am bothered that the Analytical Engine is *too much* like a modern computer. Do we infer that a computer can be built in fundamentally only one way?".[89]

This seems too strong. It is notable that Babbage and computer developers of the 1930s and 1940s were facing, in many ways, the same material situation, and were trying to develop computers that were essentially calculators. The material practice of computation had been remarkably constant in the intervening years: de Prony would have recognized and understood the industrial practices in a typical computing laboratory of the 1930s. The similarities between Babbage's and the later work can perhaps be understood as similar responses to the desire to automate the same material practice.

Babbage and Programming The Difference Engine was a physical embodiment of a single algorithmic process, and therefore did not need to be programmed. As suggested in Chap. 1, it can been seen to be more closely analogous to a single program than to a computer. Setting the machine up for a calculation would have been a matter of providing certain initial data, by setting the appropriate number wheels with the required differences.

[88] Grier (2005), p. 69.

[89] Bromley (1982), p. 217.

The Analytical Engine, on the other hand, was to be capable of evaluating any analytical formula, and so some method was needed to control the operations that the machine would carry out on a particular occasion of use. Its design was based around the idea that the sequence of operations required for a computation was specified explicitly by a deck of operation cards. However, Babbage was also aware that patterns could frequently be found in the sequence of operations to be carried out that meant, for example, that the number of operation cards could be reduced. An example of such a capability was the repeating apparatus, which in conjunction with the combinatorial and index cards would allow the engine to return to an earlier point in the sequence and repeat certain operations.

Babbage occasionally gave descriptions of what he saw as the significant patterns of computation. For example, in the *Ninth Bridgewater Treatise*, he gave a general description of the capabilities of the machine, stating that

> it will calculate the numerical value of any algebraical function—that, at any period previously fixed upon, or contingent or certain events, it will cease to tabulate that algebraic function, and commence the calculation of a different one, and that these changes may be repeated to any extent.[90]

However, he does not appear to have considered in much detail the ways in which decks of cards could conveniently be prepared by the user to specify such structured computations. In part, this was no doubt due to the fact that he put much less effort into the preparation of computational plans that to the mechanical design of the engine. According to Allen Bromley, "[a]side from the Bernoulli numbers program prepared for Ada Lovelace's notes, there is no evidence that Babbage prepared any user programs for the Analytical Engine after his 1840 trip to Turin".[91]

One later development is relevant to this issue. Throughout the 1840s, Babbage continued work on the design of the engine, and at various times produced lists of the basic operations that it would provide. Originally, these had been limited to the basic arithmetical operations, but by 1844 they had been supplemented with operations for checking whether the number at a given location in the store was equal to zero.[92] The proposed implementation of this operation involved the use of index numbers on the operation and variable cards. If the specified number was found to be equal to zero, the decks of cards would be turned back by the number of cards specified by the indexes. A similar operation was specified for detecting whether a number was greater or less than zero. There appear to be no examples of the use of these operations in detailed computation plans, however.

This proposal bears an interesting relationship to the idea that the engine was translating the language of analysis, a key part of which was the correspondence between the operations of analysis and the operations carried out by the mill. The operation of ascertaining if a variable is equal to zero does not act on numbers; rather, its purpose is to act on the sequence of cards being presented to the machine.

---

[90] Babbage (1837a), p. 187.

[91] Bromley (2000), p. 11.

[92] Bromley (2000), pp. 8–9.

It therefore represents a new type of operation, and its inclusion means that it is no longer possible to read the sequence of operation cards as a simple translation of an analytical formula.

There is an echo here of Lovelace's suggestions that the Σ in her notation of operation sequences represented a particular non-arithmetical operation of the mill, namely turning back the string of operation cards by a specified amount, and her view that the science of operations was a general theory which could be applied to many different areas. Neither Babbage nor Lovelace was able, however, to cleanly separate the operations required by this new science from the familiar operations of analysis.

Chapter 3
Semi-Automatic Computing

The previous chapter described how Babbage's repeated attempts to design and build calculating engines were ultimately unsuccessful, despite, or perhaps because of, the visionary nature of many of his proposals. The development of automatic computing proceeded in a more gradual and ad hoc manner during the late nineteenth and early twentieth centuries, and this chapter describes some of the major features of this development.

This development was not primarily driven by the demands of scientific or mathematical computation, but rather by the increasing volumes of data processing work required by government bureaucracies and large commercial organizations.[1] Large technological systems, such as railway and telegraph networks, and commercial concerns such as insurance companies required increasingly complex systems to manage their businesses, and towards the end of the nineteenth century they began to be automated, using the punched card technology. The first major application of this approach was in the 1890 census in the United States, which made use of the tabulating machines developed by Herman Hollerith.

3.1 The Census Problem

Like the Difference Engine, Hollerith's system was developed to address a particular practical problem, namely the processing and tabulation of the statistical returns needed by the US Census. The design of the system was conditioned by the needs of this particular application, and so before looking at the details of Hollerith's system, it is worth understanding how census data were processed manually.[2]

The raw data that were gathered about individual citizens in the census of 1880 included information about personal characteristics, civil status, occupation, education and place of origin. These were recorded on sheets allocating one line to each

[1] These developments have been well described by Jon Agar (2003), and in unpublished lectures by Martin Campbell-Kelly.

[2] The following account draws on the details presented by Truesdell (1965).

M. Priestley, *A Science of Operations*, History of Computing,
DOI 10.1007/978-1-84882-555-0_3, © Springer-Verlag London Limited 2011

person within a household. These data were then transformed into summary tables, which gave a statistical overview of the population. For example, a basic table in the 1880 census classified the population by race, sex and place of birth, showing for example the number of white males born in each state of the US.

A two stage process was required to generate the published tables from these data. First, the data were *tallied*: a tally sheet was ruled into boxes corresponding to the classifications required in the table. Clerks then examined each schedule in turn, entering for each individual a tally mark in the appropriate space. When the tally was complete, the marks were counted, and the totals transferred to consolidation sheets which recorded the totals, first for individual enumeration districts and then for progressively larger geographical areas as required.

One of the most significant shortcomings of this process was the need to go back and re-examine the original schedules for each different tally. Some characteristics, such as a person's sex for example, were used in many different tables. This meant in effect that the work of classifying individuals by sex was being carried out over and over again, with an obvious loss of efficiency.

To address this problem, methods were developed to transcribe all the data about individuals onto slips of paper or card. These cards could be marked in some way and so easily grouped together and counted in the various combinations required. Although this process involved an initial overhead in transcribing data onto cards, the hope was that the cost this labour would be offset by the subsequent elimination of tallying in favour of the simpler processes of grouping and counting the cards.

For example, Charles Pidgin developed a 'chip' system for the Massachusetts census of 1885; this system used cards of different colours to represent various characteristics of individuals. These cards could be sorted visually, to generate the necessary counts. In this system, the information about each individual was simply written on the card, however, and later read by clerks.

The suggestion to use machinery to automate part of the processing of the census returns appears to have been first made by John Billings, head of the department of Vital Statistics in the 1880 Census.[3] The suggestion was made to the young Herman Hollerith, who had recently graduated from the Columbia School of Mines, and was employed as a special agent by the Census Office, with the responsibility of collecting statistics concerning the steel industry.

Hollerith worked on this idea throughout the 1880s. He initially represented the information by holes punched in a long strip of paper, similar to the paper tapes used by telegraph systems. However, this proposal had the shortcoming that it was hard to process the information about a subset of the population, as the whole tape had to be read to retrieve any information from it. To get round this problem, Hollerith moved from using a continuous strip of paper to a system based on punched cards. All the information about an individual would be represent by a pattern of perforations on a single card, and by selecting and grouping together the required cards, statistics could be easily obtained on any required subset of the population.

[3] Austrian (1982), pp. 5–9.

Hollerith later claimed to have got the idea of using punched cards from the example of a system used for checking railway tickets, although his brother claimed that the idea had come from the use of punched cards in the Jacquard loom, with which Hollerith would have been familiar. Wherever the idea came from, while he was working in the Patent Office throughout the 1880s Hollerith developed a number of machines for processing census data on punched cards. In 1886, a system using punched cards was used to record and process data for Baltimore's Department of Health. This and further successful trials led to the adoption for the US Census of 1890 of a complete tabulating system developed by Hollerith.

3.2 The Hollerith Tabulating System of 1890

Representation of Data In the 1890 census, the handwritten census returns were encoded onto punched cards, with one card holding all the information about a single person. The cards contained 288 punch positions, divided into a number of fields. A field represented a single property of an individual, such as their age, sex or race, and contained a number of punch positions, each position corresponding to one possible value of that property. So, for example, the field representing a person's sex contained two punch positions, labelled 'M' and 'F': for males, the 'M' position would be punched, and the 'F' position for females.

Numbers were not represented directly on the cards. For example, two fields were used to record a person's age. The first contained 23 positions, including 21 positions representing five-year periods from 0 to 100, and the second contained five positions, labelled 0 to 4. By punching one hole from each field, any age from 0 to 104 could be represented by adding together the labels of the two holes punched. For example, an age of 37 would be represented by punching the hole labelled '35' in the first field and that labelled '2' in the second field.

Hollerith's Machines The equipment that Hollerith designed for use in the 1890 Census did not attempt to fully automate the process of tallying and tabulating the data on the schedules. Rather, it consisted of a number of separate devices which were deployed in a system which was still controlled and driven by human labour.

Two machines, the so-called pantograph and gang punches, were employed in the transcription of data from the census returns to the punched cards. The pantograph punch enabled clerks to punch holes accurately anywhere on the card, whereas the gang punch enabled reference information about the card, such as the enumeration district, to be recorded simultaneously on a group of cards prior to the punching of an individual's details.

Once punched, the cards were counted by *tabulators*. An operator would insert a card into a press, and lower an array of metal pins onto the card. The pins would pass through any holes punched in the card, making contact with a small cup of mercury underneath the cards. This completed a circuit, generating an electrical impulse which could be communicated to one or more counters mounted on the

front of the machine. Depending on the set-up of the machine, counts could be made of any of the properties or groupings of properties represented by the cards.

The demands of the census made it necessary to extract subsets of the cards, for further tabulation of properties within certain subgroups of the population. This process of extraction was assisted by a machine called a *sorter*. The sorter consisted of a number of boxes with lids that were electrically operated. A sorter could be connected to a tabulator in such a way that when a certain combination of properties was detected on a card, the lid of a particular box would be opened automatically. The operator would then insert the card into the open box, and close the lid by hand.

Functionality A large amount of clerical work was still required by Hollerith's system. Firstly, of course, the cards had to be punched from the raw census returns. They were then processed by being repeatedly run through the process of tabulation and sorting. The details of the operations performed were controlled by the physical set-up of the tabulators used, and the connections made between tabulators and sorters.

The tabulators were not limited to counting the number of cards with holes punched at particular positions. A system of relays lay between the card press and the counters, and depending on the connections made between these relays, and number of more sophisticated counts could be made. So, for example, it might be required to count the number of white males in the population. This could be done by connecting the holes for 'white' and 'male' to a single counter, using a relay, in such a way that the count would only be incremented if a current could flow through both holes. Another possibility was to join together the wires coming from a number of punch positions and connect them to a single counter so that the card would be counted if any one of the holes had been punched.

A number of checks and controls could also be built in to the tabulators. For example, additional counters could be set to generate subtotals that would be used to verify the counts. Checks could also be set so that if, for example, a batch of cards representing males was being tabulated, the machine would not accept a card representing a female.

After a tabulation, the numbers displayed on the counters of the tabulators had to be transcribed by hand onto result slips: Hollerith's machines provided no support for the tasks of storing and adding together the partial totals. The groups of sorted cards also had to be dealt with manually, stored and combined to make the batches of cards required for subsequent tabulations. Nevertheless, the gain in speed obtained by automating the tallying process, and in particular the ability to use the cards repeatedly and in various combinations, meant that the 1890 census was able to produce a wider range of statistics more quickly that had been possible previously.

The use of Hollerith's machines for the census attracted wide attention, and many comparisons were drawn between the census office and an industrial factory. One significant innovation of the system was the mechanization of data, the representation of information in a permanent, machine-readable form. Calculating machines of various sorts existed, but like the difference engine, they required that the data to be used in a calculation be entered by a human operator. For each calculation, the

data would have to be reentered. The use of punched cards changed all this: a card encoded the same information as a census return, but unlike the written return, could be read automatically and reused as often as necessary.

3.3 Further Developments in Punched Card Machines

After the 1890 census, Hollerith's business evolved in a number of ways, leading to a number of changes and adaptations to his machines. He began to explore a number of other applications areas such as, in the mid-1890s, the need for railroad companies to keep track of freight. He was awarded the contract to process the data for the 1900 US Census, but by 1905 his relationship with the Census Office had broken down, leaving his business dependent on commercial applications.

Some of the changes were in the direction of increased automation, leading to faster processing times, but did not change the basic design and functionality of the equipment. In particular, a mechanical feed was added to the sorting device, leading to a much greater throughput of cards. This capability was also provided on the tabulator, which removed the need for the operator to manually handle each card.

Some specific projects presented requirements which did involve changes to the functionality of the machines, however. The railroad application required that his tabulators had the capability to add numbers, rather than just counting perforations, so this capability was added to the tabulators, and subsequently extensively used, for example in the Census of 1900.

For bespoke applications, such as the Census and the early railroad companies, Hollerith had designed individual card formats for each user, and supplied machines wired to perform precisely the tabulation required for that task. This approach was not viable as the number of customers grew: instead, a standardized card format was adopted and, by means of a plugboard, it became easier to make the electrical connections between the tabulator and the sorter.

Standardization of Cards Prior to 1890, a variety of ad hoc card formats were used for the early applications of Hollerith's tabulators. They were characterized by the fact that hole were punched using a manual punch similar to that used by guards on trains for punching holes in tickets. They had punch positions arranged in a small number of rows round the edge of the card, as determined by the limited range of the manual punches.

For the 1890 census, however, the pantograph punch enabled holes to be punched anywhere on a card, and the population card contained 288 positions, arranged in 12 rows of 24 holes each. These punch positions were grouped into irregularly shaped 'fields', each field catering for all the possible answers to one of the questions on the census schedule.

For the railroad applications of the mid-1890s, this approach was modified to enable numerical data to be coded directly onto the cards. For example, the freight

account card that Hollerith designed in 1895 for the New York Central and Hudson River Railroad contained a mixture of fields:[4] some were irregularly shaped 'qualitative' fields, as used in 1890, but others were arranged as regular columns containing positions labelled with the digits 0 to 9. A number was represented by grouping together a number of these columns, and punching a hole in each.

Three different types of card, of differing dimensions, were used for the 1900 Census. The population card was very similar to the card used in 1890, but the farm and crop cards, used in the census of agriculture, were far more regular in layout. These cards contained some fields for identification of the farm or crop represented by the card, but the bulk of the card was made up of digit columns grouped into rectangular fields representing numbers.

The vertical fields on these cards were punched by a new device, the key punch. This punch had keys representing the 10 digits: pressing a key would result in a hole being punched in the appropriate place in the current column, and then the card would automatically move to the next field. A so-called X key was provided which would move the card onto the next position without punching a hole, in case there were no data to record in a particular field. Finally, in 1907 Hollerith introduced a standard card format, consisting of 45 digit columns. This remained a standard until 1928, when an 80 column standard was introduced.

Adding During the 1890s the tabulators were deployed in railroad companies, for calculating freight loading statistics. As well as requiring the introduction of cards that could hold numeric data, this required that the tabulators included the capability to add numbers as well as simply count cards.

Hollerith designed an 'integrating tabulator' for trials at the New York Central Railroad. An initial design proved inadequate, but a second version, known as the 'New Integrator', proved more successful and was in use at the Central by 189. This same technology was extensively used in the 1900 agriculture census.

The new integrating calculator was significantly more complex than the original tabulator. It had to interpret a hole punched in a field as a number, and then increment the appropriate counter by the correct amount. In terms of control, however, there were no significant innovations required: the fields on the cards were still physically wired to the appropriate counters.

Increasing Programmability Hollerith's early tabulators were each constructed to perform a specific task. Each machine was equipped with the appropriate number of counters, and the connections between the brushes that detected the presence of holes at particular card positions and the counters were hardwired. This meant that it was in general not possible to use a tabulator for more than one application. In 1900, for example, three types of tabulators were provided, each with connections and counters specifically set up to handle the differing requirements of the population, farm and crop censuses.

[4] Kistermann (1991).

Companies that adopted Hollerith's technology increasingly found, however, that they wanted to run different applications, using different card layouts, and that it was impractical to do this on a tabulator with a fixed configuration. A solution to this problem had been found very early on, by Otto Schäffler, who built the machines used for the Austrian census of 1890. Prior to this, Schäffler's business had been to build telephone equipment, in which plugboards were in common use. He adopted this idea, equipping the tabulators with a plugboard which enabled the connections between pins and counters to be made by plugging wires into a plugboard rather than by soldering a direct connection.

Hollerith did not immediately adopt this idea, however. The large amounts of data produced by the censuses of 1890 and 1900 meant that the time taken to rewire the tabulators between counts was relatively insignificant. There was therefore not much incentive to add to the complexity and expense of the machines by adding a plugboard. As the range of Hollerith's customers grew in the 1900s, however, the advantages of a more flexible machine became clearer, and from about 1912 Hollerith marketed tabulators which included a plugboard. A similar feature had been added to the machines supplied by Hollerith's German subsidiary company Dehomag in about 1910.[5]

By 1907, the various developments and experiments that had been undertaken in response to different customers' demands were stabilized as the decision was taken to begin marketing the 'tabulating machine' as a standard product. This machine used the standard 45-column card, and contained a key punch for the preparation of cards, and comprised an integrating tabulator which made use of a new digital adder invented by Hollerith in 1905, together with an automatic card fed mechanism.

Another common customer requirement was the ability to print subtotals during a tabulation. The deck of cards being tabulated would be sorted into groups, and in addition to the totals displayed at the end of the deck, subtotals for each group were required. Hollerith's first solution to this problem involved the use of special cards, known as *stop cards*, each with a distinctive cutout. When it encountered a stop card, the tabulator would halt, allowing the operator to read off the appropriate subtotals and reset the subtotal counters to zero before restarting the machine.

One disadvantage of this approach was that the stop cards had to be inserted manually into a deck of cards. Later developments enabled the tabulators to examine a particular field, and to stop automatically when a change in the value of that field was detected. This meant that subtotals could be taken off the machine without making any special changes or additions to the cards being processed.

Multiplication Bookkeeping and general financial applications require the ability to perform multiplications as well as addition and subtraction. The calculation of totals on invoices, for example, requires the multiplication of the item price by the number of items ordered. One approach to this problem was to add functionality to existing machines and to design a special 'multiplying tabulator', by analogy with

[5] Kistermann (2005).

Hollerith's original integrating tabulator which incorporated addition. This approach was followed by Dehomag in Germany.[6]

In the United States, by contrast, multiplication was mechanized as an operation separate from the tabulating process. In 1928 IBM, which by this time had taken over Hollerith's company, had built a prototype multiplier, and from 1931 this went into production as the IBM Type 600, a distinct machine which could multiply two numbers on a card and punch the product in a separate, blank field on the same card. In 1933, an improved multiplying punch, the IBM Type 601, was produced, which could evaluate a range of simple formulae involving the numbers on a card as well as recording the answer.

The invention of the multiplying punch changed the role of the punched card within a data processing system. Up until this point, all punched card systems had treated the cards as a read-only source of data. Cards were punched at the beginning of the process and through repeated processes of tabulation and sorting, the information they contained could be handled in various ways. The multiplying punch, however, for the first time allowed a tabulating system to add to the data recorded on the cards during processing.

Summary Punched card systems did not completely automate the process of data processing. Rather, manufacturers provided a range of machines, each performing a specific task and for a given application, a selection of machines could be used together. A certain amount of manual assistance, in punching and collating decks of cards and transporting them from one machine to another, was still required.

The manual processes required had been reduced to very simple ones, however. The planning of an application required the design of a process that was, if not fully automatic, at least in principle mechanizable, in which the human contribution was principally in routine tasks requiring little application of skill or interpretation.

Furthermore, there were many aspects of these systems that were configurable, including the set-up of the plugboard for a particular computation, the detection of changes in control fields, and subsequent printing of subtotals, and the configuration of the sorter and the calculations performed by multiplying punches. A punched card installation had the form of a large, distributed machine from the components of which the particular machine required for the task at hand could be constructed.

3.4 Comrie and the Mechanization of Scientific Calculation

Despite the uptake of punched card equipment in the commercial world, as late as the mid-1920s scientific calculation was still largely carried out using traditional manual methods. In 1925, Leslie J. Comrie, a mathematician who was shortly to do much to change this situation, suggested a number of reasons for this. Among these he considered particularly important the high initial cost and unreliability of

[6] Heide (2009), p. 122.

the calculating machines then available, and the fact that the use of machines would also require a substantial initial overhead in the development of new computational methods and non-logarithmic trigonometrical tables.[7]

In 1925, Comrie was appointed to the position of Deputy Superintendent of the Nautical Almanac Office (NAO) in Greenwich, and he subsequently served there as Superintendent from 1930 until 1936. In 1936, he left the NAO and set up what has been described as the world's first independent computing centre, the Scientific Computing Service.[8] From 1925 onwards he had attempted to increase the use of computing machinery at the NAO by adapting existing commercial machines to scientific use rather than developing new and specialized machines. He hoped by this strategy to obtain machines that were reliable and affordable; this approach proved very successful, and by 1932 could write that:

> During the past six years, the calculations done in H.M. Nautical Almanac Office have been completely mechanized. Not a single logarithm is now used. The older generation has been succeeded by one which knows only how to produce figures mechanically.[9]

In carrying out this process of mechanization, Comrie investigated all the types of computing equipment then available. Two particular classes of machine, namely register machines and Hollerith punched card machinery, turned out to offer the most substantial benefits, including the possibility of automating significant portions of large-scale computations.

Register Machines From 1929 onwards, a variety of register machines were used at the NAO for a wide range of applications based on the method of differences.[10] Comrie explicitly drew attention to the striking continuity between this work and that of Babbage a century earlier. He described the register machines in use at the NAO as "difference engines" and "modern Babbage machines", and even reminded his readers that Babbage was the first person to be awarded the Gold Medal of the Royal Astronomical Society, the very society whose journal Comrie was writing for in 1932.

These register machines consisted of a keyboard and a number of mechanisms called *registers* which were capable of storing and adding numbers. The machines could carry out two types of operation. Firstly, a number could be manually entered on the keyboard and then printed, while at the same time being added into selected registers. Typically, some registers would provide the additional capability for a number to be subtracted from their current contents. The second type of operation involved printing the total value in a register; at this point the accumulated total could be cleared, or transferred to another register. It was this last feature, the ability to transfer numbers between registers, which in Comrie's view made it possible to use these machines for scientific computing.

---

[7] Comrie (1925).

[8] For more details on Comrie's career, see Croarken (1990).

[9] Comrie (1932b), p. 523.

[10] Comrie (1932b, 1936).

Table 3.1 Calculating a function from its second differences

Position	1	2	3
Crossfooter	Non-add	Add	Add
Register	Add	Non-add	Add
Print	Cols. 1–17	Cols. 1–13	Cols. 1–13
Stop	14	6	14
Operation	Total register	Set second difference	Total crossfooter

Computed results were printed on paper which was wide enough to permit the printing of multiple columns. A movable carriage carried the paper past the printing mechanism, and a number of *stops* could be set, at each of which an operation could be carried out and a number printed. This resulted in output that was neatly tabulated in columns. At the end of a line, the machine would automatically return to the first stop position. The position of the stops and the operations performed at each were configurable in different ways on different machines.

For example, the Burroughs Class 11 machine contained two registers, known as the *crossfooter* and the *register*. It was controlled by a combination of a *stop-rod* and a *control rod*. The stop rod controlled the positions on the paper at which the machine would stop and print a number. The control rod controlled the details of the operation that would be performed at each position; for example, whether a number entered on the keyboard would be added into the crossfooter and register or not. Different computations could be performed by different settings of the rods; it was easy to change the rods in the machine, and old rods were kept so that it was easy to set up the machine quickly to perform a variety of different computations.

Comrie used a tabular notation to describe the set-up of a register machine for a particular problem. For example, we can consider the task of using the Burroughs machine to compute the values of a function from its known second differences. The overall plan was to enter the second differences on the keyboard, to add these to the first difference stored in the crossfooter, and then to add this in turn to the function value stored in the register. This procedure is summarized in Table 3.1.[11]

'Position' refers to the three columns printed, containing the function value, the second difference, and the first difference. The computation proceeded by moving from one position to another, and at each position performing the operations that were specified in the table. So in position 2, for example, the next second difference would be entered on the keyboard by the operator and printed. This difference would be added to the crossfooter, which stored the first differences. At position 3, the first difference in the crossfooter would be printed out, and at the same time added to the register, thus forming the next function value. The 'Add' in the crossfooter row at this position indicates that the crossfooter was not cleared at this point. A car-

[11]This table is from Comrie (1932b), p. 534. The operation row is not in the original, but appears in later tables given by Comrie. It has been added here to clarify the working of the procedure.

Table 3.2 Comrie's revised format for register machine calculations

Position	Stop	Operation	Print	Contents of registers
–	Margin	Initial contents	–	$\Delta^v_{-\frac{1}{2}}$

riage return would then be carried out, and in position 1 the new function value was printed from the register.

The computation started in position 1, where the initial argument and value of the function to be tabulated were entered and printed. The carriage was then moved manually to position 3, where the initial first difference was entered and added to the register. The machine then returned to position 1, printed out the second value of the function, and then moved to position 2. From this point on, the operations described in the previous paragraph were repeatedly carried out until the required tabulation was complete.

Comrie was struck by the cyclical nature of such computations, and the degree to which they could be automated on the available machinery, the human operator being viewed simply as another part in a larger mechanism:

> In the application of a machine of this nature the operations to be performed occur in cycles, and it is advisable that the operator should be given as little responsibility as possible, beyond that of entering the correct figures on the keyboard, and of performing a few simple mechanical movements. The degree of automatic control that is effected is truly remarkable.[12]

Later machines offered greater scope for calculations of this type. For example the National Accounting Machine contained six registers, allowing computations that went as far as the sixth difference, thus giving the same degree of accuracy as Babbage had planned for his Difference Engine. Because of the greater complexity of this machine, the tabular notation was adapted slightly, being supplemented with columns that listed in mathematical notation the value held by each register as the computation proceeded, as shown in Table 3.2.

Hollerith Machinery In contrast to the register machines, Hollerith equipment tended to be used in calculations where there was a large amount of data that needed to be processed repeatedly, and where the structure of the computation mapped well onto the repeated runs of punched card decks through a tabulator.

One such application was the tabulation of the positions of the moon, carried out at the NAO in 1929. Lunar positions were calculated using the data in Brown's *Tables of the Moon*. This contained 180 tables giving values of harmonic functions of the form $a \sin(b + ct)$: in order to calculate a given property of the position of the moon, values from a number of different tables were selected and added together. This was straight-forward, though laborious, work: as Comrie said, "although the

[12] Comrie (1932b), p. 529.

entries from each table are used over and over again, combinations of entries from any one group of tables do not recur".[13]

Comrie partially mechanized this procedure, replacing the work of two full-time employees of the NAO with "mechanical methods that have … eliminated much fatigue, increased tenfold the speed with which results can be obtained, and reduced the cost to one-quarter of its former amount".[14] This mechanization required that the data from Brown's tables were transferred onto punched cards, a job that took six months before the calculations could begin.

The process used was as follows. Firstly, the tables that were needed for a given calculation were identified, and the cards for those tables were arranged in stacks. The next value in the result was then found by adding the values on the top card of each stack and then moving these cards to the bottom of their stacks. In general, the stacks would be of different heights, as the different harmonic functions had different periods. As the calculation proceeded, therefore, different combinations of cards from the stacks would be selected and summed.

The cards were taken off the stacks by hand, and a new deck of cards prepared in which the cards were ordered by date rather than by table. This new deck of cards was fed through a printing tabulator, which added and printed the total for each date, and then into a sorter which rearranged the cards back into the groups which corresponded to each table. These sorted groups were then added by hand to the end of the corresponding stacks so that the cards would again be available for selection and tabulation as required.

Crucial to the success of this procedure was the ability of the tabulator to detect when all the cards for a particular date had been read, so that the total for that date could be printed and the counter reset. Comrie referred to this as 'automatic control'.

A later application, dating from 1936, was more statistical, dealing with data about the experimental planting of sugar beet.[15] Eight data values were punched onto cards for each of 7200 field locations, and these cards were then processed to find out the optimum sample size for sugar beet.

Comrie's scientific work with Hollerith machines was taken further by Wallace Eckert at Columbia University during the 1930s.[16] Eckert pointed out that punched card machines were "designed for computation where each operation is done on many cards before the next operation is begun", whereas for applications such as numerical integration it was required to perform different arithmetical operations in quick succession. To facilitate this change of use, he developed a device known as the *calculation control switch* which connected and controlled the operation of multipliers, tabulators and punches so that a sequence of operations could be carried out by the different devices in succession.

[13] Comrie (1932a), pp. 694–695.

[14] Comrie (1932a), p. 694.

[15] Comrie (1937).

[16] Eckert (1940).

3.5 Semi-Automatic Programming

The computing systems described in this chapter were not completely automatic. Commercial punched card installations consisted of a number of separate machines, each performing a single well-defined task. The sequencing of these tasks was left to human operators, however, as was much of the physical manipulation of the data, in the form of carrying packs of cards from one machine to another.

A similar situation obtains in the work of Comrie and Eckert. Comrie did much to reduce the complexity of the work carried out by humans, but did not remove it altogether. Operators were still responsible for entering data into the machines' registers and for physically ensuring that operations were carried out. The nearest approach to automation was the calculation control switch developed by Eckert.

One might feel a reluctance to describe the activities described in this chapter as programming. The semi-automated computations carried out on calculators and Hollerith machinery sit somewhere in the middle of a spectrum which has at one end a numerical analyst devising a new pen-and-paper algorithm, and at the other the programming, in the modern sense, of a fully automated machine such as the Analytical Engine.

Examples can be found to justify this usage, however. For example, Truesdell's book on the development of the US Census contains examples of "outlines of actual tabulation programs"; the term 'program' was being used here to refer to the organization of the required data processing, without there being any commitment as to the degree of automation involved in the process.[17]

The overall situation is that between 1885 and 1935 the amount of mechanical computation being carried out in the world dramatically increased. This increase was at first most noticeable in the commercial world, where both calculators and Hollerith machinery were increasingly used, but this usage later spread to scientific calculation as well, thanks to the efforts of Comrie and Eckert in particular.

As well as the use of actual physical machinery, this trend led to the instructions for calculation, even where they were to be executed by humans, becoming more routine, and requiring less 'intelligence', or interpretation, in their execution, For example, Comrie's development of a certain new interpolation method was in part motivated by a desire to reduce the amount of judgement and expertise involved in existing methods, and so to enable the process to be carried out by more junior staff.[18] As the next chapter describes, Turing and Post would take this line of thought further in their theoretical work, simplifying the execution of computations to the extent that it became plausible to imagine that the human element could in fact be replaced by machines.

[17] Truesdell (1965).

[18] Croarken (1990), p. 25.

Chapter 4
Logic, Computability and Formal Systems

The work of Comrie and Eckert demonstrated the benefits that could be obtained from even a partial automation of the processes of computation. Human input was still required in order to control operations and ensure that steps in the calculation were performed in the right order and with the right data, but increasingly all that was required was the ability to perform routine labour which involved little skill or initiative. During the 1930s, the extent to which even this residual human agency could be replaced by machines began to be investigated more systematically, both in theory and in practice.

On the theoretical side, mathematical logicians constructed various accounts of the notion of mechanical computation, or 'effective computability'. Different, but provably equivalent, accounts of this notion were published in 1936 by Stephen Kleene, Alonzo Church, Emil Post and Alan Turing.[1] At the same time, logicians also developed a new theory of formal languages, one which made concrete the idea of a notation or language that could be processed 'mechanically', and so by extension read and interpreted by actual machines.

Also in the mid-1930s Konrad Zuse in Germany and Howard Aiken in the US started to build machines that would more completely automate the processes of calculation, including the control functions still performed by humans in Comrie's laboratory. Zuse and Aiken worked independently of each other, and in ignorance of current research in mathematical logic, but there are striking similarities between the machines they designed and the theoretical machines described by Post and Turing. Their work is described in the following chapter, after a discussion of the evolution of the logical notions of effective computability and formal languages.

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. However, since the seventeenth century philosophers and logicians had explicitly linked the two

[1] This 'confluence of ideas' was analyzed by Robin Gandy (1988), who did not, however, go on to consider contemporary technological innovations.

M. Priestley, *A Science of Operations*, History of Computing,
DOI 10.1007/978-1-84882-555-0_4, © Springer-Verlag London Limited 2011

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.[2]

The new view of algebra formulated at the start of the nineteenth century and summarized in Peacock's treatise of 1830 viewed it as the study of the operations that were defined on numbers. The relevant properties of the operations of interest were captured by rules for the symbolic manipulation of formulae, and these rules could then be applied independently of the meaning of the symbols. This suggested the possibility of applying this approach to domains other than numbers, as Ada Lovelace hinted at in her brief proposal of a machine to compose music.

The first application of these ideas to a domain outside mathematics was made by George Boole, who studied the properties of the operations applicable to the truth values of logic.[3] Boole's goal was to mathematize existing logic, particularly the syllogism, but the formulae of his algebra of logic were not expressive enough 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 was sufficiently expressive, but one in which the processes of deduction did not possess the clarity and ease of use of algebra.[4] Deduction was represented by a formal system defined by a number of axioms and rules of inference, and Frege's system and those based on it, notably Russell and Whitehead's *Principia Mathematica*,[5] shared with the algebra of logic the property that logical relationships and proofs could be checked by manipulating symbols, putting aside any thought of their meaning. Among others, Gödel, drew attention to 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.[6]

In arithmetic, algorithmic processes such as long division will lead in time to an answer to any problem, if correctly applied. By contrast, Frege's system did not seem to provide a guaranteed way of establishing whether a particular conclusion followed from a set of premisses. The decision problem or *Entscheidungsproblem*, given prominence by Hilbert and Ackermann,[7] was the question of whether there was a mechanical process 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 introduced many

[2] See Pratt (1987) or Davis (2000) for accounts relevant to the development of computers.

[3] Boole (1854).

[4] Frege (1879).

[5] Whitehead and Russell (1910).

[6] Gödel (1931).

[7] Hilbert and Ackermann (1928).

ideas and techniques that were widely used later, and it is a convenient place to begin a more detailed consideration of the logical background.

4.1 Gödel's Construction

Gödel's incompleteness proof was based on the idea of constructing, in the formal system of *Principia Mathematica*, a self-referential sentence similar to those that occur in paradoxes such as the Liar.[8] The various formulations of the Liar involve a sentence of natural language which asserts its own falsity; examples are "I am now lying" or "this assertion is not true". Call this latter statement S: the problem is that it does not seem possible to say definitively whether S is true or false. Assume, for example, that S is true; what it says is therefore correct, namely that the assertion S itself is not true. If S is not true, however, then the assertion it makes seems to coincide with the facts, implying that it is, after all, true.

A similar situation seems to arise with a sentence such as "this statement is not provable"; for brevity, call this statement U. Let us assume that all statements which can be proved are in fact true. Then if U is provable, what it asserts is true; however, what it asserts is its own unprovability, so from this contradiction we conclude that U is not, in fact, provable. This, however, is exactly what U asserts of itself, so we conclude that U is in fact true. This suggests that there are sentences which are true, but cannot be proved to be so, a claim which undermined the hopes of logicians to construct a formal system within which all the truths of mathematics could be proved.

Informal presentations of the paradoxes, such as those given above, rely on the fact that in natural languages like English it is possible for sentences to refer to other sentences, and even to themselves, and also to make semantic assertions, such as "S is true" about those sentences. Given these resources, the paradoxical sentences can easily be constructed. It is not obvious that a system designed for formalizing mathematics will be equally expressive, however, and a large part of Gödel's paper was devoted to showing that this was in fact possible, and presenting the technical details of how to define the self-referential sentences in any formal language, like that of *Principia Mathematica*, which had the ability to formalize number theory.

Arithmetization Gödel based his proof on the details of a specific formal system, which he called P. The description of P was given in natural language; and started with the definition of a set of primitive signs. The primitive signs of P were '$\sim$', '$\vee$' and 'Π', representing the logical notions 'not', 'or' and 'for all', '0' and 'f' for the number zero and the successor function, and the punctuation symbols '(' and ')'. There were also variables such as x_1 and y_1 of type 1, standing for numbers, x_2 and y_2 of type 2, standing for classes of numbers, and so on.

[8] See Barwise and Etchemendy (1987).

The formulae of the system were defined to be finite sequences of primitive signs, and Gödel wrote that "it is easy to state with complete precision *which* sequences of primitive signs are meaningful formulae and which are not".[9] This was carried out by first defining a general notion of a sign, as a sequence of the form a, *fa*, *ffa*, *fffa* and so on, in which a could either be 0 or a variable of type 1; these signs therefore denoted numbers. Elementary formulae were defined to be combinations of signs of the form $a(b)$ where b was a sign of type n and a a sign of type $n+1$. Finally, formulae were defined to be members of the class containing the elementary formulae and sequences of signs of the forms $\sim(a)$, $(a) \vee (b)$ and $x\Pi(a)$, where a and b were themselves formulae and x a variable of any type.

A number of further signs, such as '=', were then introduced by definitions, and Gödel went on to give a number of axioms; some of these, such as

$$\sim(f x_1 = 0),$$

which asserted that zero was not the successor of any number, defined the properties of the natural numbers, and others defined a system of propositional and predicate logic. In essence, P combined Peano's axioms for the natural numbers with the logic of *Principia Mathematica*, and provided a concrete example of a formal system which was powerful enough to represent elementary number theory.

Gödel then pointed out that for metamathematical purposes the exact choice of primitive signs was irrelevant, and proposed to use natural numbers as primitive signs in place of the conventional typographic symbols. The primitive symbols were assigned natural numbers as follows:

$$\begin{array}{llll} \text{`0'} \to 1, & \text{`}f\text{'} \to 3, & \text{`}\sim\text{'} \to 5, & \text{`}\vee\text{'} \to 7, \\ \text{`}\Pi\text{'} \to 9, & \text{`('} \to 11, & \text{`)'} \to 13, & \end{array}$$

and variables of the form 'x_n' were assigned the number p^n, where p was a prime number greater than 13. Given this encoding, every formula could be represented as a sequence of natural numbers instead of a sequence of primitive signs. Gödel then described how the sequence $n_1, n_2, \ldots, n_k$ of natural numbers could be encoded as the single natural number $2^{n_1}.3^{n_2} \ldots p_k{}^{n_k}$, where p_k is the kth prime number. Proofs in P were simply sequences of formulae, and so by repeating this process they too could be coded as natural numbers.

The details of Gödel's construction meant that every sign, formula and proof in P could be represented by a distinct natural number, its *Gödel number*. Gödel denoted the function which mapped linguistic elements to their Gödel numbers and picked out an "isomorphic image" of P in the natural numbers by Φ.[10] This encoding of the formulae of P as numbers, or *arithmetization*, is at the heart of Gödel's approach.

Formalizing Metamathematics Arithmetization is an essential preliminary to the attempt to construct a formal equivalent of the sentence U in the system P. P is a

[9] Gödel (1931), p. 147, emphasis in original.

[10] Gödel (1931), p. 147.

language for talking about numbers, so by encoding the linguistic structures of P as integers, a sentence of P can be interpreted as referring to the signs, formulae or proofs of P itself. The other thing that needs to be done is to show how notions such as "sentence" or "provable" can be expressed in the language of P: only when this has been done will it be possible to exhibit a translation into P of the sentence U.

The terms requiring translation deal with the syntax and semantics of P; they are normally be expressed in natural language and referred to as *metamathematical* properties of P. By using arithmetization, however, they can be defined by predicates in the system P which apply to the isomorphic image $\Phi(P)$, rather than as informal properties of the formulae of P themselves. In other words, an informally stated syntactic or semantic property R_P of formulae of P, such as the property of being provable, can be expressed formally as a property R_N of natural numbers provided that R_P is true of certain formulae if and only if R_N is true of the corresponding Gödel numbers, or in symbols, if:

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

Gödel did not define the metamathematical properties directly in the language of P, however, but proceeded in a slightly roundabout manner which allowed him to use a more expressive notation. He defined a class of number-theoretic functions and predicates which could be specified by so-called 'recursive' definitions; this class included bounded quantifiers and a predicate which picked out the smallest number with a given property. He further proved that in principle every recursive function and predicate could be directly defined by a formula of P itself, and that the use of recursive functions was a convenience adopted for the sake of readability, but not a necessity.

Like a formula of P, a recursive predicate defines a property of natural numbers; however, by interpreting these numbers as elements of $\Phi(P)$, the class of Gödel numbers, it can also be interpreted as defining a property of sentences of P. Gödel accordingly defined the relevant metamathematical properties of P by means of a series of recursive definitions, culminating in the definition of a predicate *Bew* which was true of the Gödel number of a formula in P if and only if that formula was provable. With this machinery in place, Gödel could construct the formal equivalent of U, and so demonstrate the incompleteness of formal systems like P.

It is not necessary to examine the details of Gödel's construction here; it is worth noting, however, that it made it explicit that many linguistic properties, particularly those having to do with the syntax of language, could be defined mathematically or formally, rather than 'precisely' in a natural language. This gave support to the efforts of Carnap and others later in the decade to come up with a definition of formal languages, and thanks to the association that already existed between the notions of formality and mechanizability, also suggested ideas about languages that could be processed by machines.

Gödel's Strategy In summary, then, Gödel encoded the formulae 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 formulae of P. These

formulae could then be interpreted in two ways: firstly as statements about natural numbers, and secondly, thanks to the mapping Φ, as statements about formulae of P. By means of this second interpretation, P was capable of acting as its own metalanguage. This enabled Gödel to construct a formula of P which, in this second interpretation, made reference to itself and asserted its own unprovability.

The techniques and arguments employed by Gödel were highly influential. In particular, Turing adapted Gödel's strategy in his definition of a universal machine, as discussed in detail in Sect. 4.6. 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.

4.2 Recursive Functions

The class of functions that Gödel called 'recursive' had been investigated before 1931. Functions over the natural numbers were commonly defined using 'simple recursion'; a familiar example is the definition of the operation of adding two natural numbers a and b by the following equations:

$$a + 0 = a,$$
$$a + (b + 1) = (a + b) + 1.$$

A definition by simple recursion suggests a simple step-by-step approach to the evaluation of the function. For example, given the definition above we can reduce an expression like $2 + 3$ so that it only uses the primitive successor operation by rewriting it as follows:

$$2 + 3 = (2 + 2) + 1 = \big((2 + 1) + 1\big) + 1.$$

Procedures like this 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.

With this in mind, in 1919 the Norwegian mathematician Thoralf 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 and based 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".[11] This approach highlighted 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

[11] Skolem (1923), p. 333, emphasis in original.

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".[12]

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

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

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

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

This is a generalized form of the definition of addition given above. 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.[13]

It turned out, however, that there were functions which appeared intuitively to be effectively computable, but which could not be defined by using only substitution and recursion. Wilhelm Ackermann discovered two techniques for constructing such functions: one involved higher-level recursions, making use of "functionals" which could take functions as arguments, and the other involved simultaneous recursion on more than one variable.[14] A simple example of the second category is the function defined by the following equations, which involves a 'double' recursion:

$$\begin{aligned}\psi(0, y) &= y + 1,\\ \psi(x+1, 0) &= \psi(x, 1),\\ \psi(x+1, y+1) &= \psi\big(x, \psi(x+1, y)\big).\end{aligned}$$

It can be proved that this function grows faster than any function that can be defined using only substitution and recursion, and yet it appears that for any numbers m and n the equations show how the value of $\psi(m, n)$ could be calculated in a mechanical, step-by-step manner in which the arguments of ψ are continually decreasing.[15]

To address this problem, Gödel later gave a more general definition of recursive functions, based on a suggestion made by Jacques Herbrand.[16] The idea was that any given system of equations, such as those defining the Ackermann function above, which included an unknown function ψ and a number of functions already known to be recursive, defined a "general recursive function" if from the equations exactly one equation of the form

$$\psi(k_1, \ldots, k_n) = m$$

[12] Hilbert (1926), p. 388.

[13] Gödel (1931), p. 159.

[14] Ackermann (1928).

[15] See Cutland (1980), for example, for further discussion of the function ψ.

[16] Gödel (1934).

could be derived, for natural numbers k_i and m. In other words, a general recursive function was one defined by a system of equations involving a function symbol ϕ from which it followed that ϕ was in fact a uniquely defined function. Definitions involving only substitution and recursive definition were special cases for which this property could easily be proved.

In his 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.[17] 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.

4.3 λ-definability

The λ-calculus, a notation for the definition of functions, 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 formulae.[18]

Schönfinkel took the notion of a function as primitive, 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 $x_1, \ldots, x_n$.[19]

Church first made use of this work in a paper on the foundation of logic.[20] In his original notation, the function of **x** defined by an expression **M** was represented by the notation λ**x**[**M**], and the application of a function **F** to an argument **X** by the notation {**F**}(**X**). These notations were related by the rule that function applications of the form {λ**x**[**M**]}(**N**) could be evaluated by substituting the argument **N** for the variable **x** in the expression **M**, giving a result which Church symbolized as $S^{\mathbf{x}}_{\mathbf{N}}\mathbf{M}|$. This procedure could also be reversed, allowing a term to be rewritten as the application of a function to an argument.

[17] Kleene (1936a).

[18] Schönfinkel (1924).

[19] Curry (1929).

[20] Church (1932).

It turned out that the attempt to found logic on the basis of the λ-calculus 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.[21]

In 1936 Church and Kleene both published proofs that the set of functions that could be defined in the λ-calculus was exactly the same as the set of recursive functions, and therefore that both approaches could be identified with the notion of effective computability.[22] Church's paper gave a more detailed account of the formal properties of the notation of the λ-calculus, including an arithmetization which Church described as "the Gödel representation of a formula".[23] This was used to demonstrate that syntactical operations on formulae 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 1934 definition of general recursive functions has been described as an attempt, carried out at Church's suggestion, to put forward an alternative account of effective computability.[24] For Church and his colleagues, however, both approaches seemed intuitively acceptable, and their unexpected equivalence gave support to Church's thesis.[25]

4.4 Direct Approaches to Defining Effective Computability

Both Gödel and Church characterized the informal notion of effective computability by specifying 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.[26] In 1936, Emil Post and Alan Turing independently gave analyses of effective computability which aimed at greater "psychological fidelity", in Post's words.[27] 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.

Post and Turing described models which were based on taking seriously the metaphor that human beings perform certain intellectual tasks in a mechanical way.

[21] Rosser (1984), p. 345.

[22] Church (1936), Kleene (1936b).

[23] Church (1936), p. 349.

[24] Rosser (1984).

[25] Church (1936), footnote to p. 346.

[26] Gandy (1988), Soare (1996).

[27] Post (1936), Turing (1936).

Turing, for example, was inspired by the behaviour of 'computers', the term then current to describe people who carried out complex calculations by following predefined plans.[28] 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.

Post's Formulation In Post's model, computational work was carried out by a "problem solver or worker" in a "symbol space", which Post envisaged as "a two way infinite sequence of spaces or boxes". Each box could be in one of two states: either "empty or unmarked", or "having a single mark in it, say a vertical stroke". The worker operated within this symbol space, and could only occupy one box at a time.

A problem was presented to the worker by singling out a specific box as the starting point and marking a finite number of boxes with a stroke. After the work was complete, the answer would be given by the final configuration of marked boxes. During the solution of the problem, the worker could perform only the following "primitive acts":

(a) Marking the box he is in (assumed empty)
(b) Erasing the mark in the box he is in (assumed marked)
(c) Moving to the box on his right
(d) Moving to the box on his left
(e) Determining whether the box he is in, is or is not marked.[29]

The solution for a problem was given by a finite sequence of *directions*; the ith direction had one of the following forms:

(A) Perform operation O_i [O_i = (a), (b), (c), or (d)] and then following direction j_i.
(B) Perform operation (e) and according as the answer is yes or no correspondingly follow direction j_i' or j_i''.
(C) Stop.[30]

The set of directions was to be preceded by a standard header reading "*Start at the starting point and follow direction* 1".

Unlike Gödel, Kleene and Church, Post did not define an arithmetization of his notation. Without giving any proof, however, he anticipated that his account would "turn out to be logically equivalent to recursiveness in the sense of the Gödel-Church development",[31] as indeed proved to be the case.

The proposals made by Post and Turing were similar in many respects, though Turing gave much more detail and proved a number of significant theoretical results.

[28] Jon Agar (2003) has described how similar 'mechanical' processes had been introduced in non-numerical areas, particularly in the British Civil Service, and speculates that awareness of this was an additional factor leading to Turing's mechanical definition of computability.

[29] Post (1936), p. 103, emphasis in original.

[30] Post (1936), p. 104, emphasis in original.

[31] Post (1936), p. 105.

In place of Post's symbol space, Turing described a "tape" infinite in one direction only and divided into squares each capable of storing one of a range of distinct symbols. Instead of referring to a vaguely anthropomorphic notion of "worker", however, Turing talked in terms of "machines" which embodied just the agency that was necessary to execute the basic operations and respond to the contents of the tape. This made explicit the sense in which the model represented the nature of *effective* computation, and Turing went a step further in showing 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". The next two sections describe Turing's proposals in more detail.

4.5 Turing's Machine Table Notation

Turing's proposal involved the definition of a class of abstract machines embodying the essential processes carried out by a human clerk or computer. The behaviour of specific machines was described using a notation that Turing called *machine tables*, and in the first half of his 1936 paper this notation was developed into a powerful and sophisticated formalism for describing computations.

Despite the importance of Turing's work, the machine table notation itself has not received much attention from logicians, and has gained a reputation for being obscure and confusing.[32] When it has been discussed in detail, it has often been with a view to identifying anticipations of features found in later programming languages rather than understanding its structure and the underlying reasons for its design.[33] Some of the resemblances between Turing's notation and subsequent languages are indeed striking, but focusing on these leads to a rather unhistorical interpretation of Turing's work. This section describes the machine table notation in the context of the other notations for computability defined in the 1930s, and highlights the continuities between Turing's work and that of his contemporaries.

Turing Machines When carrying out complex calculations, humans follow well-defined procedures in a way that is often described as being 'mechanical'. There is a clear intuitive difference between calculations that follow a rule-based procedure like long multiplication, and the use of methods that involve guesswork or intuition. In trying to understand this distinction and come up with a definition of the informal notion of effective computability, Turing took the mechanical metaphor literally, defining the 'computable numbers' to be those which could be written down by a machine.

Turing went on to flesh out this proposal by defining in detail a class of machines, now known as 'Turing machines', that would be able to calculate and write down numbers. The principal features of these machines were based on aspects of the

[32] See Chaitin (2001), p. 16, for example.

[33] See Knuth and Trabb Pardo (1980) and Copeland (2004a) for representative examples.

behaviour of a human being carrying out a computation. Turing identified three important characteristics of such procedures.

1. Human computation typically involves writing symbols on paper. Abstracting from the example of school exercise books, where digits are written within preprinted squares, Turing suggested that it would be sufficient to provide each machine with a *tape*, a one-dimensional sequence of squares in each of which a single symbol could be placed. He further assumed that only a finite number of different symbols could be used.
2. Turing then discussed the role that memory plays in human calculation: at any stage in a computation, the next action taken will depend on the calculator's memory of what stage in the procedure has been reached, and perhaps also some details of the results computed so far. Turing described this situation by saying that the next step in any computation was determined by the 'state of mind' of the calculator and the symbols being observed. States of mind were modelled by what Turing called *m-configurations*, and he stipulated that a machine could be in one of a finite set of m-configurations at any given time. Turing further assumed that at any moment a machine would only have access to some of the squares on its tape, known as the *scanned* or *observed* squares. This was intended to reflect the fact that computations can be carried out which are so large that a human cannot immediately recognize all the details of the work being done, and at different times will focus on particular aspects only.
3. Human calculators can be observed to direct their attention to different parts of the paper being used and to write symbols at various places. Accordingly, the basic operations available to a Turing machine are to place a symbol in one of the scanned squares, to erase a symbol found in a scanned square, and to change the distribution of the scanned squares on the tape. Turing claimed that "these operations include all those which are used in the computation of a number".[34]

A Turing machine computes in 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 new m-configuration and scanned symbols then define a new configuration which determines the machine's behaviour in the next step of the computation.

A particular class of machines can be defined by specifying the set of symbols used, the details of which squares on the tape can be scanned at any one time, and the precise repertoire of basic operations. The machines that Turing described in detail are only capable of scanning one square at a time, and at any step in the calculation can only change the scanned square to one of the adjacent squares. The basic operations, then, allow the machine to move by one square left or right on the tape, as well as writing or erasing a symbol in the currently scanned square. The symbols used may vary from machine to machine, and are specified as required.

[34]Turing (1936), p. 232.

The different roles played by memory in computation were clearly distinguished by Turing. It is possible to carry out small computations mentally, in which case the intermediate results are not written down, but just remembered for a short period of time. This suggests the possibility of representing the intermediate results as m-configurations rather than writing them on 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 typical 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.

Machine Tables An individual Turing machine carries out a single computation. In 1936, Turing was interested in the definition of computable numbers and so was concerned primarily with machines which produced unending sequences of 0s and 1s. These sequences were interpreted as the binary representations of real numbers between 0 and 1. Each machine computed a single well-defined sequence. Turing described these machines using a notation known as *machine tables*: a table does not describe the physical structure of machine, however, but its behaviour, or the sequence of basic operations that it will carry out.

The simplest form of machine table consists of a set of rows, each defining what the machine does in a single step of a computation. The behaviour of a machine is determined by its configuration, so each row in a table corresponds to a single machine configuration. and consists of the following elements:

1. The m-configuration and the scanned symbol that define the configuration in question. Gothic characters were used to denote m-configurations, and symbols were shown literally; the word 'none' was used to denote a blank square. The word 'any' was used to indicate that the identity of the symbol in the scanned square had no effect on the machine's behaviour.
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 in the scanned square, E for the operation of erasing the symbol in the scanned square, and L and R for the operations of moving the scanned square one position to the left or right, respectively.
3. The m-configuration that the machine moves to when the operations have been carried out, known as the *final* m-configuration.

Using these conventions, Turing's first table described a machine which printed the sequence '010101 . . .' on alternate squares of the machine's tape:[35]

[35]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 conforms to Turing's initial description of how the machines behaved, which stated that they could perform at most one write or erase operation and one move at each step in the computation. Turing immediately extended the notation in two ways, however. 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 allowed all the rows in a table that shared the same m-configuration to be grouped together, using a notation similar to the mathematical notation for 'definition by cases'. The required behaviour is then associated with the currently scanned symbol. These conventions can be used to give a simpler table for the machine defined above which uses only one m-configuration:[36]

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

Turing gave the following informal explanation of how these simple tables were to be interpreted:

> 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.[37]

This basic notation is enough to permit the fundamental structures of computation to be expressed. The rows in a table are not ordered, but the sequencing of operations is determined by the explicit specification of a final m-configuration in each row; this mechanism also allows recurring cycles of operations to be specified. Finally, the ability to select between alternatives is provided by allowing the machine, when it is in a particular m-configuration, to carry out different courses of action depending on the nature of the currently scanned symbol.

A Standard Tape Format In the examples given so far, the computation performed is so simple that each machine can simply write the required output directly on to its tape. There is no need to examine or alter any symbols once they have been written, and no requirement to calculate or recall any intermediate results. More complex problems cannot be tackled in this way. In general, machines need to be able to examine their earlier output and to erase and rewrite the symbols that are used temporarily to record the progress of a computation.

[36] Turing (1936), p. 234.

[37] Turing (1936), p. 233.

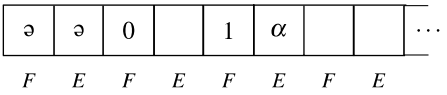
This presents a difficulty, because a Turing machine has no way of knowing where on the tape the current scanned square is. The squares on the tape have no absolute coordinates, and the view that the machine has of the tape is strictly local. At any moment it can examine one square only and move to an adjacent square, but it has no built-in way of recognizing the scanned square as one which has been visited previously, distinguishing one occurrence of a symbol from another, or locating a specific square on the tape.

Turing addressed these problems by defining a standard tape format to be used in computations. This format is not part of the definition of the Turing machines, and many alternative formats could be chosen. It is simply a set of conventions which make it easier to write and understand programs.

There are two aspects to the tape format used by Turing. Firstly, a special symbol is used to mark the beginning of the tape, or rather, since the tape is assumed to be unbounded to left and right, the position of the machine when computation started. Turing's examples only made of the portion of the tape to the right of this position, though other conventions and uses would be perfectly feasible.

Secondly, a means is required of enabling squares to be marked in some way so that a particular square can be recognized at different stages in a computation. Turing achieved this by dividing the squares on the tape into two categories, known as *F-squares* and *E-squares*. These squares alternate, each F-square being followed by an E-square which can store an arbitrary symbol used to mark, or label, the F-square to its left. In Turing's application, the F-squares hold the *figures*, the binary digits 0 and 1 which made up the output of the computation, whereas the E-squares hold temporary and erasable marks. More general conventions could be used in other applications: the general role of the E-squares is to allow the F-squares to be named and subsequently identified.

The tape format described by these two conventions is illustrated in the diagram below:



The first two squares on the tape contain the 'start of tape' characters; as the tape to the left of these characters is not used, it is not shown on the diagram. The tape contains two figures, a 0 and a 1. Turing allows no spaces in the sequence of figures stored in the F-squares, so the blank F-square shown at the right of the figure above marks the end of the tape, or rather the end of the portion of the tape that has been used in the current computation. The figure 0 is unmarked; the figure 1, and the square it is in, are said to be *marked* by the symbol α which appears in the following E-square.

Turing used these conventions in his second example, a machine which prints the sequence 0010110111011110…[38] To print the next block of 1s at the end of

[38]Turing (1936), p. 234.

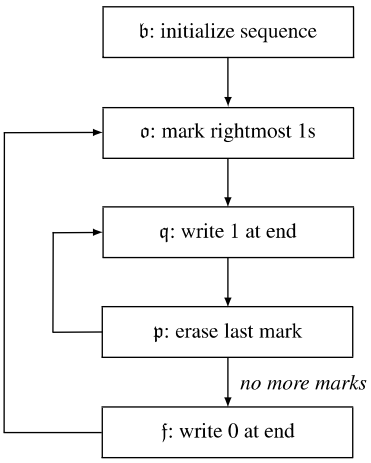


Fig. 4.1 The design of the machine to print 0010110111011110...

the tape, it is necessary in some way to count the 1s in the preceding block and to print a block containing one more. Turing achieves this by marking all the 1s in the rightmost block of 1s, writing a 1 at the end of the tape, and then removing the marks. Each time a mark is removed, an additional 1 is written at the end. Once all the marks have been removed, therefore, the new block contains one more 1 than the block before, and by repeating this process indefinitely, the required sequence is generated.

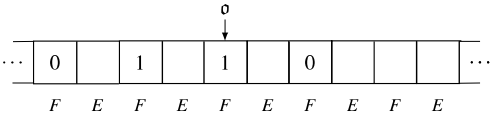
The machine tables that Turing presented typically had a very clear structure, with each *m*-configuration having a well-defined role to play in the computation. These roles, together with an indication of the way that the computation moves from state to state, can conveniently be depicted in an informal flowchart, as shown in Fig. 4.1. This makes clear the nesting of the two loops in the computation, a feature which is not so apparent in the corresponding machine table.

The detailed table for this computation, comprising the same *m*-configurations as the flowchart, is shown below.

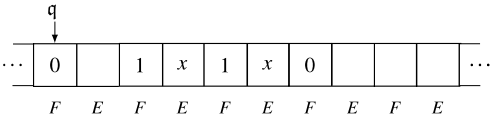
m-config.	*symbol*	*operations*	*final m-config.*
b		Pə, R, Pə, R, $P0$, R, R, $P0$, L, L	o
o	1	R, Px, L, L, L	o
	0		q
q	Any (0 or 1)	R, R	q
	None	$P1$, L	p
p	x	E, R	q
	ə	R	f
	None	L, L	p
f	Any	R, R	f
	None	$P0$, L, L	o

In m-configuration $\mathfrak{b}$, the symbol column is left blank, so the operations specified will be carried out whatever the contents of the scanned square. At this point, the two start of tape symbols and the first two 0s in the output sequence are printed. Note that figures are only printed on F-squares, so that *two* R operations are required between the print operations, in order to skip over the intervening E-square. The machine then moves one F-square to the left, so that it is positioned over the first 0 written on the tape, and moves to m-configuration $\mathfrak{o}$.

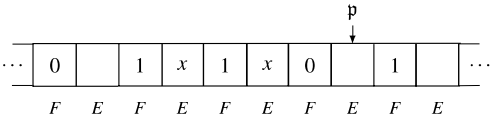
Whenever m-configuration $\mathfrak{o}$ is entered, the last figure written on the tape is a 0 and the machine is scanning the F-square to the left of this figure. The first time this m-configuration is entered, the symbol in this square will be a 0, but in general it will be the final 1 in the last sequence of 1s on the tape, as the diagram below illustrates.



In this diagram, the scanned square is identified by an arrow with the current m-configuration written above it. In m-configuration $\mathfrak{o}$ the machine moves left, marking each 1 with the symbol x; when it encounters a 0 it stops and moves to m-configuration $\mathfrak{q}$. At this point the rightmost block of 1s on the tape has been marked, and the scanned square is the F-square to the left of this block, which will contain a 0, as shown below.



In m-configuration $\mathfrak{q}$, the reference to the symbols '0 and 1' is not a restriction, but a clarification: when this state is entered, an F-square after the beginning of the tape is being scanned and the only symbols written in such squares by this machine are 0 and 1. The machine now moves right over the F-squares until it finds an empty one. This signifies that the end of the used portion of tape has been reached, so the first 1 of the new block is written, and the machine moves left to an E-square and enters state $\mathfrak{p}$, as shown below.



In m-configuration $\mathfrak{p}$, the machine moves left on the E-squares until it finds either a x or the beginning of the tape. If an x is found, it is deleted and the machine

moves back to m-configuration $\mathfrak{q}$. As described above, this results in an additional 1, corresponding to the mark that has just been deleted, being written at the end of the tape, and the machine then moves back to m-configuration $\mathfrak{p}$.

This alternation between m-configurations $\mathfrak{p}$ and $\mathfrak{q}$ continues until the machine, moving left on E-squares in m-configuration $\mathfrak{p}$ reaches the start of tape. This means that there are no more marks on the tape, and hence no more 1s to be written. The machine then moves to m-configuration $\mathfrak{f}$, in which it moves right on the F-squares to the end of the tape, writes a 0, moves to the F-square preceding the last 0 on the tape, and then moves back to m-configuration $\mathfrak{o}$ ready to repeat the whole process.

This example illustrates how Turing used a standard tape format to monitor and control the progress of a more complex computation. It also illustrates one way in which this format makes the machine designer's job harder: it is necessary to keep track carefully of whether the machine is expected to be scanning an E-square or an F-square at each point in the computation.

Variables and Functions Turing next considered 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 which will form part of most machines and which may be carried out many times in a given computation; these included, for example, locating, copying, comparing and erasing symbols on the tape. It would obviously save effort, and make tables shorter and more readable, if these processes could be defined in one place and reused, rather than being written out in full whenever required.

In the context of recursive functions, this problem had been addressed by the technique of substitution, which allowed an already existing function to be used in the definition of a new function. For example, Gödel's definition of the predicate *Bew* came at the end of a long sequence of intertwined definitions in which simple functions were repeatedly reused to define more complex ones. Turing applied this technique for a similar purpose 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. Conventional notation uses variables to represent those elements of the expression that can vary in different contexts of application. For a computational process, like long multiplication, it is natural to say that the same procedure can be used to operate on different numbers, or that it can be used in different overall contexts. However, if different operations were carried out, one would be tempted to say that the definition of the procedure had changed.

Turing reflected these intuitions in the machine table notation by introducing variables for symbols and m-configurations, and by allowing m-configurations to be denoted not simply by names, but by expressions known as *m-functions*. Tables containing these extensions were called *skeleton tables*.

The skeleton table 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.[39] If α does occur on the tape, the

[39] Turing (1936), p. 236.

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 $\mathfrak{B}$.

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 $\mathfrak{C}$, $\mathfrak{B}$ and α. The fact that the three m-functions involved have the same name, distinguished by a subscripted index, was an informal naming strategy used to emphasize that they are parts of a self-contained table with a single unified purpose.

When the machine defined by this table starts, it is in m-configuration $\mathfrak{f}(\mathfrak{C}, \mathfrak{B}, \alpha)$, in which it moves left until it reaches the beginning of the tape. The machine then moves to m-configuration $\mathfrak{f}_1(\mathfrak{C}, \mathfrak{B}, \alpha)$ and begins to search for an α. If an α is found, the machine moves to the 'success' m-configuration, $\mathfrak{C}$, and the computation ends. If a blank square is found the machine moves to m-configuration $\mathfrak{f}_2(\mathfrak{C}, \mathfrak{B}, \alpha)$, and otherwise it moves one 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 $\mathfrak{f}_1(\mathfrak{C}, \mathfrak{B}, \alpha)$, with one exception: the discovery of a blank square means that two blanks in a row have been found, and hence 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-functions have a different significance depending on whether they appear in the first or last column of a table. In the first column they behave rather like λ-expressions, binding the variables that appear in that row: this can be seen by noting that the bound variables in a row could be consistently renamed without changing the behaviour specified by the table. In the final column, however, the m-functions are applied in order to determine the machine's next m-configuration.

Application of m-functions involves the replacement of 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 move to the m-configuration specified by the expression $\mathfrak{f}(\mathfrak{d}, \mathfrak{e}, x)$. This is the first row of the skeleton table above, with the obvious replacement of $\mathfrak{d}$ for $\mathfrak{C}$, $\mathfrak{e}$ for $\mathfrak{B}$, and x for α. The machine will then search for the first occurrence of the symbol x on the tape, as specified in the skeleton table. If this search is successful, the machine will move to m-configuration $\mathfrak{d}$, in which case 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".[40] Applying this procedure to the table fragment above, 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 basic notation without variables and m-functions.

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{c}$			$\mathfrak{f}$
$\mathfrak{f}$	$\mathfrak{d}$	L	$\mathfrak{f}_1$
	not $\mathfrak{d}$	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	$\mathfrak{e}$
$\mathfrak{e}$			...

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 the invocation of 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.

[40] Turing (1936), p. 236.

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 and more recently by Copeland.[41] However, a richer historical understanding of Turing's motivation for the introduction and use of skeleton tables is gained by viewing it in the light of contemporary practice. As noted above, the strategy of defining complex functions in terms of simpler ones was commonly used by writers on recursive functions and effective computability.[42] In this context, Turing's use of skeleton tables appears as a natural extension of the same strategy to the new domain of machine tables.

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.

Substitution, in this context, was defined by Gödel as the "substitution of some of the preceding functions at the argument places of one of the preceding functions".[43] In the machine table notation, this means allowing m-function expressions to appear in the argument positions of the application of an m-function. As in the case of recursive functions, this provided a powerful mechanism whereby new operations could 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:[44]

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-function $\mathfrak{e}(\mathfrak{C}, \mathfrak{B}, \alpha)$ 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 erase 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 important to note that it implies a different order of evaluation. 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 the m-function $\mathfrak{f}(\mathfrak{e}_1(\mathfrak{C}, \mathfrak{B}, \alpha), \mathfrak{B}, \alpha)$, however, the effect is that the computation denoted by the m-function $\mathfrak{e}_1$ takes place *after* that denoted by $\mathfrak{f}$.

[41] Knuth and Trabb Pardo (1980), p. 201, Copeland (2004a), p. 12.

[42] See Skolem (1923) and Kleene (1935a, 1935b) for further examples.

[43] Gödel (1931), p. 159, footnote.

[44] Turing (1936), p. 237.

The m-function $\mathfrak{e}(\mathfrak{C}, \mathfrak{B}, \alpha)$ 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-function to achieve this:[45]

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. A machine in m-configuration $\mathfrak{e}(\mathfrak{B}, \alpha)$ moves straight to an application of 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 machine will move to the success m-configuration, which in this case is simply $\mathfrak{e}(\mathfrak{B}, \alpha)$, and this process will repeat as often as necessary to delete the second and any subsequent occurrences of α. Eventually, there will be no more αs on the tape and the deletion will be unsuccessful, and the machine will move to the m-configuration $\mathfrak{B}$.

Free Variables The final syntactic feature of the notation provides a way to pass the value of the currently scanned symbol to the final m-configuration without having to name it, by allowing 'free' symbol variables, which are not among the arguments to the initial m-function, 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.[46]

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{pe}(\mathfrak{C}, \beta)$			$\mathfrak{f}(\mathfrak{pe}_1(\mathfrak{C}, \beta), \mathfrak{C}, \mathfrak{d})$
$\mathfrak{pe}_1(\mathfrak{C}, \beta)$	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.[47]

[45] Turing (1936), p. 237.

[46] Turing (1936), p. 237.

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

mconfig.	*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, and wrote 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".[48]

4.6 Universal Machines

Turing initially stressed the role of memory in human computation: an individual machine simulates the behaviour of a human who is performing a calculation by following a memorized procedure. However, humans can also compute by following written instructions, as in the case where a new procedure is being carried out for the first time. Turing described another scenario in which written material is required to enable a computation to proceed:

> It is always possible for the computer to break off from his work, to go away and forget all about it, and later to come back and go on with it. If he does this he must leave a note of instructions (written in some standard form) explaining how the work is to be continued. This note is the counterpart of the "state of mind".[49]

In the extreme case we can imagine that the computer only performs a single step of the computation in each period of work.

If we imagine that the computer also forgets the details of the procedure being carried out, or perhaps that each step is performed by a different computer, it is clear that the note of instructions must comprise a description of the procedure as well as a record of the stage the computation has already reached. In terms of machines, this means that the note must include the machine table as well as details of the current m-configuration and the symbols on the tape.

Now, consider the case of a person who is able to interpret a note of instructions and carry out the basic operations involved in computation. It appears plausible that they would be able to perform any computation whatsoever, regardless of whether

[47] Turing (1936), p. 238.

[48] Turing (1936), pp. 236, 239.

[49] Turing (1936), p. 253.

or not they knew the details of the procedure involved. A person with this ability would have a kind of 'universal' skill, and would not be limited to carrying out a specific set of memorized procedures.

The ability to interpret and carry out instructions itself appears to be an effective, or 'mechanical' skill, at least if it is assumed that the instructions are sufficiently explicit and detailed. This raises the question of whether it is in fact mechanizable, of whether it would be possible to construct a single machine which would be able to perform any computation. Turing answered this question in the affirmative, by explicitly defining such a *universal* machine, which he called $\mathcal{U}$.

$\mathcal{U}$ is a Turing machine which can be supplied with the machine table for any other Turing machine $\mathcal{T}$. It works through the steps that $\mathcal{T}$ would have performed, recording on its tape all the details of the configurations that $\mathcal{T}$ passes through in the course of a computation. Matters can be arranged so that $\mathcal{U}$ writes down exactly the output symbols that $\mathcal{T}$ would write down, even though the internal details of the computation carried out are different. $\mathcal{U}$ is therefore "a single machine which can be used to compute any computable sequence",[50] and the existence of $\mathcal{U}$ 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 [universal] machine as doing something quite simple, namely carrying out orders given to it in a standard form which it is able to understand".[51]

Turing began the construction of the universal machine by formalizing the simple form of machine table to which all tables can be reduced. This format is referred to as the *standard form* of a table. It is assumed that an enumeration $q_1, \ldots, q_m$ of the m-configurations used in the table is given, and also an enumeration $S_0, \ldots, S_n$ of the symbols which can appear on the tape, including an explicit representation of a 'blank' symbol to denote an empty square. 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

[50] Turing (1936), p. 241.

[51] Turing (1946), p. 21.

moves one square to the left or right, or stays put, and the final m-configuration is q_m.

For a concrete example of standard form, consider the first example table on p. 80. 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 ... 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 each line in the standard form table is prefixed 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 read by other machines. Like Kleene and Church, Turing went one step further and produced an arithmetization of his notation, thereby 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.

Turing's Universal Machine Turing gave a completely explicit definition of the universal machine $\mathcal{U}$ using the machine table notation. He first defined a number of general-purpose skeleton tables, and the table for $\mathcal{U}$ itself is broken down into nine subtables, each with a single well-defined role to play in the overall computation.

For present purposes, the most important thing is to understand how the execution of $\mathcal{T}$ is represented on the tape of $\mathcal{U}$. The progress of $\mathcal{T}$ is defined by its table, by the symbols written on its tape, and by the position of the currently scanned square. In general, these last two factors will change at every step of the computation.

$\mathcal{U}$ therefore keeps a record on its tape of the complete contents of the tape of $\mathcal{T}$ at each stage of the computation, along with the standard form of $\mathcal{T}$'s table. During the course of a computation, the structure of $\mathcal{U}$'s tape is as follows:

$$;\underbrace{DADDCRDAA}_{\textit{instruction}};DAADDRDAAA::DCDAADCC:0:\underbrace{DCDDAADDCC}_{\textit{complete configuration}}.$$

The instructions making up the table of $\mathcal{T}$ appear in standard form at the start of the tape, terminated by the special symbol ::. The remainder of the tape is occupied by *complete configurations*, also in a standard form. These record the contents of $\mathcal{T}$'s tape at a particular point in the computation, along with a note of the position of the currently scanned square. These complete configurations are separated by colons; at each stage of the simulation $\mathcal{U}$ will write the figure, if any, produced by $\mathcal{T}$ at that stage, followed by the next complete configuration.

A complete annotated account of the tables comprising Turing's definition of $\mathcal{U}$ is given in the Appendix; this gives a very good idea of how the machine table notation could be used in the definition of larger computational processes.

4.7 The Concept of a Formal Language

The second area of research to be considered in this chapter is the development of a mathematical theory of a formal languages themselves. Logicians were familiar with the similarity between the syntactic operations involved in the definition of a formal notation and the recursive, or 'inductive', definitions used in mathematics. For example, in an early paper Church commented on an informal explanation of the structure of the well-formed formulae of his system, saying that "[t]his is a definition by induction".[52]

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", 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".[53]

This insight made the development of a mathematical theory of formal languages possible, a development associated particularly with the work of Alfred Tarski and Rudolf Carnap.[54] This section briefly describes the major features of their account of formal languages, which later served as the theoretical framework within which programming languages were studied.

[52] Church (1932), p. 352.

[53] Kleene (1936a), p. 727, Church (1936), p. 94, footnote 8.

[54] Tarski (1933), Carnap (1937, 1939, 1942).

Object Language and Metalanguage The starting point of this account was the distinction drawn between the language under investigation, or *object language*, and the *metalanguage*, the language in which the investigation was carried out. This distinction originated in Hilbert's notion of metamathematics, or the study of the formal properties of mathematical systems. Gödel was one of the first people to make explicit use of this distinction: in the technical parts of his 1931 paper, he used distinct logical symbols to distinguish the object language P from the metamathematical notation used to define recursive functions over the expressions of P by for the two cases.

For Tarski, the importance of the distinction between object and metalanguage lay in the fact that not every language possessed "terms belonging to the theory of language",[55] and so in general it would not be possible to discuss the syntax, say, of a language in that language itself. Carnap made the link to syntax explicit, writing 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*".[56]

This distinction raised the possibility of the need for an unending hierarchy of metalanguages. Carnap argued against this view, emphasizing 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.[57]

Different metalinguistic resources were needed for different purposes. For the purposes of logical syntax, Carnap only needed a metalanguage in which he could describe 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 not only the syntactic forms of object-language sentences, but also their meanings. Tarski therefore demanded that the metalanguage was 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".[58]

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", and Carnap

[55] Tarski (1933), p. 167.

[56] Carnap (1937), p. 4, emphases in original.

[57] Carnap (1937), p. 53.

[58] Tarski (1933), p. 172.

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".[59]

Formal 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.[60]

The rules of formation described the structure of the expressions of a language, and defined which expressions constituted meaningful sentences or formulae. 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.[61]

Property (α) specified that the alphabet of the language should 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) said that it should be possible to distinguish the subset of well-formed expressions, or sentences, within the complete class of expressions purely by looking at 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 open minded, and commented 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.

---

[59] Tarski (1936b), p. 403, Carnap (1937), p. 1, italics in original.

[60] Carnap (1937), p. 4.

[61] Tarski (1933), p. 166, italics in original.

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".[62]

Carnap believed that Tarski's definition of truth was a good example of the type of semantic definition needed for formal languages. Tarski required that "the sense of every expression is uniquely defined by its form".[63] 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 its syntactic construction. 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.[64]

The Structure of the Metatheory Tarski's definition of truth drew attention to 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, which Morris based on the process of *semiosis*, a 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".[65] 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". 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".[66]

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".[67] Carnap suggested that all three aspects, which he called "pragmatics", "semantics" and "logical syntax", should be studied as part of a theory of language.

[62] Carnap (1939), p. 6.

[63] Tarski (1933), pp. 165–166.

[64] Tarski (1933), p. 193.

[65] Morris (1938), p. 3.

[66] Morris (1938), pp. 6, 13.

[67] Carnap (1939), p. 4.

4.8 The Relationship Between Turing's Work and Logic

Turing's 1936 paper has often been described as foundational to the development of computing and the computer science that soon followed. This chapter has taken a different perspective, however, emphasizing instead the close relationship between Turing's ideas and contemporary work in mathematical logic. This is not an attempt to deny Turing's originality, however, but rather to relocate it in the details of his mathematical practice.

Turing's use of Gödel's work fits well with the account of theoretical innovation as conceptual modelling put forward by Andrew Pickering.[68] Pickering viewed this as a three-stage process, and called the first stage *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. For example, in his development of the machine table notation, Turing appears to have systematically imported some key notational features from more established logical theories into his new domain, as described in Sect. 4.5. Similarly, Gödel's technique of arithmetization was transcribed and used in the definition of the universal machine, as discussed in Sect. 4.6.

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. One significant difference originates in the basic semantic distinction between the languages. The atomic formulae in conventional languages are formed by applying a predicate to one or more terms, which are in their turn made up from variables and constants combined with function applications. Semantically, terms denote objects in some domain of interpretation, and the role of atomic formulae is to make assertions which can be true or false. More complex formulae can be built up using truth-functional connectives and quantifiers, and these formulae 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 appear on the tape, the m-configurations, and the primitive actions that can be taken by a machine. More complex terms can be built up using m-functions. However, there are no predicates, and hence no atomic formulae 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

[68] Pickering (1995).

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",[69] 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 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".[70] 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.

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 understand their semantics: 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, then, 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. Both began with the definition of 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. Gödel's P is a formal language

[69] Turing (1936), p. 233.

[70] Turing (1936), p. 243.

for number theory, so its formulae 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 formulae. Finally, the chosen 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 ...[71]

However, 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 been argued that Turing applied and generalized existing work in logic, particularly that of Gödel, to give a formalization of the semantic notion of obeying a command, namely the universal machine.

[71] Morris (1938), p. 16.

Chapter 5
Automating Control

In October 1945, shortly after the end of the Second World War, a conference on Advanced Computation Techniques was held at the Massachusetts Institute of Technology.[1] The conference was held to coincide with the first public demonstration of MIT's new differential analyzer, but in the event many of the presentations at the conference were about machines of a very different type. This new type of machine worked on digital rather than analogue principles, provided a much greater degree of automation than existing machines, and in some cases held the promise of extremely fast computational speeds.

The best known of these new machines were American: machines using electrical relays had been developed in a collaboration between Harvard University and IBM, and also slightly later at Bell Laboratories, and an electronic machine, the ENIAC, had been constructed at the University of Pennsylvania. The electronic machines in particular demonstrated the possibility of attaining computational speeds far in excess of what could be achieved by electro-mechanical means.

New developments had also been taking place elsewhere, however: in Germany, Konrad Zuse had constructed a number of mechanical and relay-based machines, and in the UK efforts to build machines to assist in the task of breaking German codes had culminated in the development of Colossus, a special-purpose machine that nevertheless had much in common with the more general-purpose calculators. In the confused situation at the end of the war, however, Zuse's machines did not achieve wide publicity, and even the very existence of Colossus remained classified information in the UK for many years.

The detailed stories behind the construction of these machines are well known. There were many differences of detail between them, and the fact that the ENIAC was the first large-scale electronic machine has often led historians to consider it separately from the relay machines, and to treat it rather as a precursor of later developments. However, when viewed from the perspective of the development of programming, it is evident that a common approach informs the design of all these

[1] A brief summary of the conference was given by Archibald (1946).

M. Priestley, *A Science of Operations*, History of Computing,
DOI 10.1007/978-1-84882-555-0_5, © Springer-Verlag London Limited 2011

machines, one which represents a distinctive stage in the development of automatic computation. This chapter gives a brief description of some of these machines and the way they were programmed, before drawing some general conclusions on their common features.

5.1 Konrad Zuse's Early Machines

In 1935, Konrad Zuse had recently graduated as an engineer from the Technical University of Berlin and, apparently motivated by a desire to automate the long and complex calculations he had to perform, left his recently acquired job at the Henschel Aircraft Company and decided to dedicate himself to building computers.[2] 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".[3]

The Z1, which was completed in 1938, was a purely mechanical implementation of Zuse's ideas. However, it turned out to be rather unreliable, and so Zuse started work on its successor, the Z2. This machine had the same basic design as the Z1 but was built with a mechanical memory and a calculating unit built using relays. Finally, in 1938 he started construction of the Z3, which was again built to the same design, but this time using relays exclusively. The Z3 became operational in 1941, but despite being demonstrated to a variety of government departments, was never actually put to work. It was destroyed in an air raid in 1944.

The Design of Zuse's Machines According to Zuse's later reconstruction of the design process, he began by considering the paper forms that were typically used to plan manual computations. Numbers were entered in cells at various places on these forms, and the layout of the form was utilized to indicate the order in which the operations were to be performed. Zuse gave an example where numbers in the horizontally adjacent cells of the form were to be multiplied together and numbers in vertically adjacent cells were to be added.

His first thought was simply to try to mechanize this arrangement, using a two-dimensional layout in which numbers would no longer be written on paper, but instead represented by holes punched in the forms. The numbers would be read and transferred to a calculating unit by a sensing device attached to a mobile arm. Zuses's subsequent refinements of this idea included the use of reusable registers instead of punched holes to store numbers, and then the abandonment of the form layout altogether. Zuse came to view this as only being of importance for human computers. A machine, on the other hand, would not benefit from the intuitive hint

[2]The biographical details in this section are largely taken from Zuse's autobiography, first published in German in 1984. See Zuse (1993) for a translation.

[3]Ceruzzi (1983), p. 29.

that adjacent numbers were to be operated on together, and a simple and arbitrary sequence of registers could be used instead.

A computation for the Z3 was specified by a *computing plan*, which broke a formula down into a sequence of the elementary operations required to compute it. As an example, Zuse gave the evaluation of the formula $\sqrt{a^2 + b^2} = c$. If a, b and c were identified with the registers V_1, V_2 and V_3, respectively, the following plan would computer the value of c:

$$\begin{aligned} V_1 \cdot V_1 &= V_4, \\ V_2 \cdot V_2 &= V_5, \\ V_4 + V_5 &= V_6, \\ \sqrt{V_6} &= V_3. \end{aligned}$$

Zuse's design, therefore, drew a fundamental distinction between a *storage unit*, comprising a set of registers which only stored numbers, and a *calculating unit*. A computing plan was to be read from a punched tape; a *program unit* would transmit the numbers of the registers required for a computational step to a *selection unit* which was responsible for accessing the required data. The program unit would also communicate the details of the operation required to the calculating unit.

The basic outlines of this design were modified slightly in the case of the Z3. A tape reader read commands, passing register addresses to the memory unit, and a control unit synchronized the machine for performing the requested operation. Numbers could be entered from a keyboard, and results were output on a numerical display.[4]

Programming the Z3 Programming the Z3 was very simple: numbers could be loaded from the registers into the calculating unit, an operation performed, and the result transferred back to a register. The calculating unit contained two registers, $R1$ and $R2$. To perform an operation, the first number was loaded into $R1$, and the second into $R2$. The required operation was then specified, and the result placed in $R1$. Subsequent load operations placed numbers in $R2$, so continued calculations could be easily performed. Finally, the result in $R1$ could be stored or displayed, at which point $R1$ was cleared; the next load operation would then refer again to $R1$.

As an example of the Z3's code, Table 5.1 shows the sequence of instructions required to carry out the calculation shown above. This example makes no attempt to minimize the number of commands involved in the computation. As the Z3 was only used for demonstration purposes, it appears that at this stage Zuse did not consider ways of iterating subsequences of commands or of conditionally executing certain sections of a program.

[4]This description, together with the following details on programming the Z3, are taken from Rojas (2000).

Table 5.1 A program for Zuse's Z3 computer

Pr 1	(load a into $R1$)
Pr 1	(load a into $R2$)
Lm	(form a^2 in $R1$)
Ps 4	(store a^2 in V_4)
Pr 2	(load b into $R1$)
Pr 2	(load b into $R2$)
Lm	(form b^2 in $R1$)
Ps 5	(store b^2 in V_5)
Pr 4	(load a^2 into $R1$)
Pr 5	(load b^2 into $R2$)
Ls1	(form $a^2 + b^2$ in $R1$)
Ps 6	(store $a^2 + b^2$ in V_6)
Pr 6	(load $a^2 + b^2$ into $R1$)
Lw	(form $\sqrt{a^2 + b^2}$ in $R1$)
Ps 3	(store $\sqrt{a^2 + b^2}$ in V_3)

5.2 Mark I: The Automatic Sequence Controlled Calculator

In the mid-1930s, Howard Aiken, then a graduate student at Harvard, conceived the idea of building an automatic calculator to perform scientific calculations. Unlike Zuse, Aiken did not intend to build the machine himself, but hoped to interest a company with experience in building calculating machines in his ideas. He initially contacted the Monroe Calculator Company, but after little progress was made in negotiations, Aiken succeeded in interesting IBM in his proposals.

Between 1939 and 1943, the calculator was built by a group of IBM engineers, following the design specifications laid down by Aiken. In early 1944, it was moved from IBM's research laboratory to Harvard where it became operational on March 15. A formal dedication ceremony was held on 7 August, an event which gave rise to considerable bad feeling between Harvard and IBM, who felt they had not been given sufficient recognition for their role in the development of the machine and their generosity in donating it to Harvard.

The new calculator was immediately put into service to help solve a wide range of scientific and mathematical problems. These including ballistic computations, a question about implosion properties for John von Neumann, and the production of a new set of tables of Bessel functions. Many of these problems required significant machine time to complete. The implosion calculation took most of September 1944 to run, and the production of Bessel function tables ran throughout 1945 and much of 1946, unless interrupted by higher priority problems.

Originally known as the Automatic Sequence Controlled Calculator (ASCC), once Aiken's machine was installed at Harvard it acquired the more familiar name of Mark I. It was the first in a series of machines, the Marks II, III and IV, developed at Harvard during the 1940s and 1950s. In many ways, these later machines followed the design of Mark I, and so by the time they were developed were seen as being

rather conservative, and although performing much useful computation, did not have the impact on computer developments that Mark I had.

Aiken's Proposal The clearest description of Aiken's motivation in designing the ASCC is contained in a proposal he wrote in 1937 for an "automatic calculating machine".[5] This proposal formed the basis for Aiken's initial contacts with IBM. The proposal makes clear that Aiken had in mind a machine that would provide help with the growing computational needs of "the mathematical and physical sciences". He listed a number of examples, including the tabulation of many new functions.

Aiken situated his proposed machine at the end of long a tradition of mechanical aids to computation. Babbage's work is described in some detail, though it is not clear whether Aiken fully understood Babbage's plans for the Analytical Engine. The influence of Babbage's used of punched cards on Hollerith's work is discussed, and Aiken highlighted the fact that commercial uses of Hollerith machines were in fact carrying out many of the things that Babbage wanted to accomplish.

However, in Aiken's opinion, the Hollerith machines were not suitable vehicles for scientific computation. He gave a number of reasons for this, including the facts that they could not handle positive and negative numbers and that mathematical functions were not available. He also pointed out that scientific calculations have a different structure from data processing tasks: whereas a commercial punched card machine would carry out a single operation on a large data set, represented by a pack of cards, the reverse situation, that of performing an extended sequence of operations on a small data set, was what was typically required in scientific computation. He pointed out the repetitive nature of many computations, and stated that

> For this reason calculating machinery designed for applications to the mathematical sciences should be fully automatic in its operation once a process is established.

Aiken described the current computational capabilities of IBM's punched card equipment, and appeared to envisage that his new machine would be constructed out of a suitable set of machines of existing types, linked together. In the light of this, he wrote that

> The whole problem of design of an automatic calculating machine suitable for mathematical operations is thus reduced to a problem of suitable control design ... The main features of the specialized controls are machine switching and replacement of the punched cards by continuous perforated tapes.

Aiken's view of the typical calculation that his machine would perform involved an independent variable which would be incremented in equal steps. The machine would then calculate the value of the required function for each value of this variable. To carry this out automatically, Aiken defined the Master Control as the part of the machine responsible for moving numbers from one position in the machine to another, and starting a particular operation.

[5] Aiken (1937).

The Design of the ASCC Aiken planned to build the machine by plugging together existing IBM products and supplying suitable control facilities. In the end, it turned out that the requirements of the machine were such that many new components were required. However, the overall design of the machine remained that of a number of connected but functionally distinct components.

The various components of the ASCC were connected by means of the *number transfer bus*. The bus was used to convey numbers, which were coded as a series of timed electrical impulses, between components; numbers consisted of 23 digits together with a plus or minus sign. The functional components of the machine were connected to the bus by relays: input relays controlled the transfer of numbers from the bus to a component, and output relays controlled transfer from a component to the bus.

The most important components were 72 storage registers. Each register could hold a single number, and was connected to the bus by input and output relays. The registers were not simply storage devices: when a number was transferred to a register it was added to, or subtracted from, the existing contents of the register, rather than replacing the number already stored there. This essentially replicated the functionality provided on existing manual register machines.

Whereas addition and subtraction were performed by the registers, a separate unit was provided to carry out multiplication and division. To perform a multiplication, say, the numbers to be multiplied together had to be transferred to the multiplication unit and the multiplication operation started. Once the operation was complete, the product could be transferred to a register for storage and later use.

The ASCC also contained three interpolation units, which could calculate the values of functions or return them from data punched on paper tape, and a functional unit which could evaluate the special functions $\sin x$, $\log_{10} x$ and 10^x. Numeric data could be provided to the machine on 60 constant registers and two card readers. Numbers had to be entered manually onto the constant registers at the start of a computation; by contrast, packs of cards could store numbers that could be used repeatedly. The results produced by the ASCC could be punched onto cards, and subsequently used as input to a later calculation, or printed on electric typewriters connected to the machine.

The overall behaviour of the ASCC was under the control of the sequence control unit. At each step of the computation, this unit was responsible for setting relays so that a number could be transferred from one location in the machine to another, and for invoking the operation to be performed, such as the execution of a mathematical process such as a multiplication or function interpolation.

The sequence unit read commands from a paper tape. A command had three components, specifying the input source and output destination for the transfer of a number, and the operation that should be initiated. As well as the mathematical functions, the operation of reading the next command on the tape had to be explicitly specified as part of a command; this made it possible to arrange for a computation to be automatically halted in certain circumstances. Commands were read from the paper tape in strict sequence: in particular, it was not possible to wind back the tape or to skip over a command.

Despite the use of punched cards, and the connection with the Hollerith machines manufactured by IBM, the overall design of the ASCC more closely resembled a giant version of the existing manual register machines than a typical punched card installation. As Comrie had pointed out, the ability to transfer numbers between registers was the key factor that made register machines suitable for carrying out scientific calculation, and this was precisely the basic operation carried out by the ASCC.

Programming the ASCC Commands read by the sequence unit consisted of a pattern of perforations in the 24 punch positions available on the input tape. These were divided into three fields of eight positions each. Fields *A* and *B* each contained a coded representation of a location within the machine, while field *C* defined the settings of the relays that controlled the operation of the machine. The general form of a command was "Take the number out of unit *A*; deliver it to unit *B*; start operation *C*", and according to Aiken and Hopper, a sequence of commands of this form was sufficient to carry out any computation that used the five fundamental arithmetical operations.[6]

Because of the nature of the registers, the operation of addition did not need to be explicitly specified. A command of the form (21, 7321, 7) would have the effect of adding the contents of register 3 (code 21) to register 71 (code 7321) and then advancing to the next command on the tape (operation code 7).[7] Subtraction was performed by forming the complement of the number stored in the *A* register and adding that to the contents of the *B* register. Forming the complement had operation code 32, so the command to subtract the contents of register 3 from the contents of register 71 and then execute the next command would be coded as (21, 7321, 732).

The operation of resetting a register to zero was accomplished by subtracting the contents of the register from itself. This was coded by a command of the form (21, 21, 7), in which the *A* and *B* fields have the same location code. A special 'reset relay' detected a command with duplicate location codes,and ensured that the necessary complement was taken.

Multiplication could not be coded by a single command: two commands were required to move the numbers to be multiplied into the multiplication unit, and a third to move the product back to a register. To multiply the numbers in registers 56 (code 654) and 18 (code 52) and store the result in register 13 (code 431) the following commands would be given:

(654, 761, *blank*)

(52, *blank*, *blank*)

(*blank*, 431, 7).

[6] Aiken and Hopper (1946).

[7] The code numbers represent the columns to be punched in each field of the tape. The location code 21, therefore, indicates that columns 1 and 2 would be punched, giving a representation of the binary numeral 00000011, corresponding to register 3.

In the first command, the 761 code in the *B* field represents the first register in the multiplication unit; the command has the effect of transferring the contents of register 56 to this register. The multiplication unit was designed to compute the first nine multiples of this number before reading in the multiplier. To allow time for this to happen, the first command does not contain the operation code 7 specifying that the sequence unit should immediately read the following command. Instead, control was passed to the multiplication unit which would instruct the sequence unit to read the next command when construction of the table of multiples was complete. The second command simply contains the code for the register containing the multiplier; its destination was left implicit. The actual multiplication would then take place, and when complete, the multiplication unit would instruct the sequence unit to read the third command, specifying the location in which to store the computed product. This command does contain the 'continue' code, 7, in field *C*, so at this point control was returned to the sequence unit and the computation would proceed normally.

While the internal operations of the multiplication unit were taking place the number transfer bus was not in use. It was soon realized that in many cases the overall time taken by a computation could be reduced by using these intervals to carry out other operations that did not involve the multiplication unit. This was accomplished by interposing the necessary commands between the three commands that defined the multiplication. For example, the code sequence

(654, 761, 7)
(21, 7321, *blank*)
(52, *blank*, *blank*)
(*blank*, 431, 7)

would perform the same multiplication as in the previous example, but would also add the contents of register 3 to those of register 71, using the main bus, while the multiplication unit was computing its table of multiples. The first command ends with the continuation code instructing the sequence unit to proceed immediately to the second; when this is complete, the sequence unit pauses and waits for the multiplication unit to request the next command. This technique clearly required the programmer to pay great attention to the timing of the computation, to ensure that the interposed commands are completed by the time the multiplication unit is ready for its next command.

Division and the computation of function values by means of the interpolators and the 'electro-mechanical tables' for sines, logarithms and exponentials were coded in a similar way to multiplication, involving the use of multiple commands and subsidiary sequence control by the specialized units. They also allowed the use of interposed commands, therefore.

Certain registers were equipped with additional circuits to provide specialized functionality. These including the provision for storing numbers with 12 or 46 digits instead of the standard 23, thus increasing either the storage capacity or the precision of the machine, depending on the needs of particular applications. A particularly significant extension involved register 72, the so-called *check counter*, to provide

a means for checking the results produced during a computation, and halting the machine if an error was detected. It was argued that all numeric checks could be reduced to checking whether a given value c was equal to zero, to within a specified degree of tolerance t. In other words, we need to check if $t - |c| > 0$. If T and C were the codes for the registers holding the values of t and c, respectively, such a check could be coded as follows:

$$(T,\ 74,\ 7)$$
$$(C,\ 74,\ 71)$$
$$(\mathit{blank},\ \mathit{blank},\ 64).$$

The first line loads t into register 72 (code 74) and the second subtracts the value of $|c|$ from the same register, using the special operation code 1 which returned the negative absolute value of the contents of a register. The operation 64 in the final line generated a continuation code only if it was detected that the result of the subtraction was not less than zero. Otherwise, the machine would stop, and the operator would be required to investigate what had caused the check to fail.

The ASCC could execute automatically any sequence of commands presented to the sequence unit. However, it was soon recognized that many problems could not be efficiently represented as a single sequence of commands. In particular, many problems had an iterative structure, requiring a particular sequence of commands to be given to the machine many times; for example, problems involving the tabulation of functions fell into this category.

On some occasions, the iterated commands were simply punched repeatedly on the sequence tape; however, this involved additional punching, and also preparatory work to calculate the required number of iterations.[8] An alternative strategy was to physically join the two ends of the tape containing the commands to be iterated, thus forming a loop, or 'endless tape'. This enabled the ASCC to carry out the iteration automatically, but typically meant that a computation would require a 'starting tape' to be run before the iteration could begin, thus increasing the amount of operator involvement required.

These problems were addressed in 1947 by equipping the ASCC with a second control unit, known as the 'subsidiary sequence unit', and defining commands that would enable control to be switched from one unit to another. In practice, however, a significant degree of human intervention was still required to run a computation on the ASCC; the complete documentation for a program included detailed instructions for the operators of the machine as well as the commands making up the program itself.

5.3 The ENIAC

The ENIAC was constructed in a collaborative project between the Moore School of Electrical Engineering at the University of Pennsylvania and the Ballistics Research

[8] Bloch (1947).

Laboratory (BRL) of the US Army Ordnance Department at the nearby Aberdeen Proving Ground. The BRL was committed to the use of mathematics to improve the state of ballistics, and had been liaising with the Moore School since before the war. Of particular concern was the production of firing tables for new weapons, and the ENIAC was intended to carry out the calculations involved in the production of such tables.

The initial impetus for the Moore School work was provided by John Mauchly who before the war was professor of physics at a small college near Philadelphia.[9] 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, and in 1941 visited the University of Iowa and examined an electronic device developed by John Atanasoff which was intended to solve simultaneous equations. 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 Presper Eckert, an electronic engineer, and succeeded in interesting him in the possibility of using electronic technology to construct very high speed calculating devices.

In 1942, Mauchly wrote a report entitled "The Use of High-Speed Vacuum Tube Devices for Calculating", in which he stressed 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.[10]

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, a mathematician with a background in ballistics who in 1942 had been posted to the US Army Ordnance Department at the BRL.

A significant computational problem faced by the BRL was the timely production of firing tables for new artillery. A firing table listed properties of the trajectory of a shell, depending on various external factors, and a large amount of computation was involved in the production of each table. 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 designed and built over the subsequent two years; even before its official completion, it was used to perform calculations for the Manhattan Project

[9] See Stern (1981) and McCartney (1999) for general accounts of the history of the ENIAC.

[10] Mauchly (1942).

in the autumn of 1945, and it was first publicly demonstrated on February 14, 1946, when it was famously reported as being able to "compute the trajectory of a shell faster than the shell itself flies".[11] It was then transferred to the Ballistics Research Laboratory, where it was extensively used until finally being decommissioned in 1955.[12]

The ENIAC was not the first actual or proposed machine to use electronics for automatic calculation, but the project was on a far larger scale than its predecessors, and demonstrated once and for all the feasibility of constructing reliable, large-scale electronic devices. Although its design was inspired by the need for the production of firing tables, it was not a special-purpose machine, and in the course of its active life was used for many other mathematical problems.

The Structure of the ENIAC The ENIAC was a large and innovative machine: it consisted of 30 functional units, which together contained approximately 18,000 vacuum tubes and 1,500 relays, making it by a long way the most complex electronic device in existence at the time it was built.[13] Its design, however, drew extensively on existing technology: in the original proposal of April 1943, Mauchly and Eckert stated that

> [i]t is then, in every sense, the electric analogue of the mechanical adding, multiplying and dividing machines which are manufactured for ordinary purposes,[14]

and this was particularly clear in the units which performed arithmetic operations. The most significant of these were the *accumulators*: the ENIAC possessed 20 accumulators, each of which could store a single 10 digit number and perform the operations of addition and subtraction. These units behaved in a similar way to the ASCC's registers: when an accumulator received a number, it would be added to or subtracted from the number already held there. In a further echo of the design of the ASCC, multiplication was performed by a separate, dedicated unit, and a further unit was provided to carry out the operations of division and the extraction of square roots.

The accumulators therefore provided storage for 20 numbers as well as the ability to operate on these numbers during a computation. Constant numeric data could be provided on three *function table units*. Although these were intended to store the values of tabular function, in fact any numbers known before the start of a calculation could be stored on them. Numbers could also be stored on a unit known as the *constant transmitter*: this was intended to be used for the parameters specific to a particular computation, and numbers were read into it from standard IBM punched card reader. If the numeric data produced in the course of a calculation exceeded the

[11] Burks (1947), p. 756.

[12] Fritz (1994).

[13] Goldstine and Goldstine (1946) give a contemporary account of the ENIAC, from which many of the following details are taken.

[14] *Proposal for ENIAC, 4/8/43*, quoted in Marcus and Akera (1996).

electronic storage available, numbers could be sent to a *printer*, from which they would be punched onto cards and later read in again via the constant transmitter.

The remaining three units were the *master programmer*, which controlled certain aspects of the overall computations, and *initiating* and *cycling* units which provided the electronic signals which controlled the operation of all the other units.

The Operation of the ENIAC The ENIAC did not read numbers or instructions directly from punched card or paper tape readers, as the Z3 and ASCC had done. The mismatch between the slow speed of these mechanical input devices and the very high speed of the electronic computing circuits made this approach infeasible: the time taken to read a card would be so long relative to the time taken to process the number on the card that all the advantages of speed provided by electronics would be lost. With numeric input, this problem was solved by the constant transmitter, which transformed the data given on punched cards into a stored electronic form. This approach was not followed in the case of program commands, however; instead, the designers of the ENIAC adopted an distributed approach to control.

The individual units of the ENIAC contained *program switches*, which defined the operations that unit would perform in a given computation. There was no central control unit controlling the execution of operations; instead, control was established by means of electronic signals, known as *program pulses*, which were passed between units. When a unit received a program pulse, it would carry out the operations specified by its program switches, and then emit a program pulse to trigger the next operation required, typically on a different unit.

The units of the ENIAC were therefore connected by a number of circuits: *digit trunks* transmitted numbers, and *program trunks* transmitted program pulses. Each trunk was made up of a number of lines along which individual signals could pass. The operations of the machine were synchronized by means of an electronic signal generated by the cycling unit. The fundamental unit was the *addition time*, or the time taken by an accumulator to perform an addition, 1/5000th of a second.

The master programmer unit controlled certain aspects of the overall flow of control in a computation. It consisted of ten *steppers*, each of which received and counted program pulses and depending on the number of pulses received so far, sent the subsequent outgoing pulse to one of six different connections. A number was associated with each of the six outgoing connections: these numbers acted as thresholds, and when the threshold number of pulses was received, the destination of the outgoing pulse would change to the next connection. The master programmer therefore allowed cycles of repeated operations to be defined in a computation.

The program switches on an accumulator allowed it to be set to receive a number, adding it to or subtracting it from the current contents of the accumulator, or to transmit a number. A number could also be transmitted as a complement, to enable it to be subtracted from the contents of the accumulator receiving it. In a single addition time, an accumulator could receive a number, and transmit its contents through either or both of its outputs. An accumulator could be set to perform these operations repeatedly, up to nine times.

Programming the ENIAC Programming the ENIAC was fundamentally different from the Z3, the ASCC, or even Babbage's Analytical Engine. Rather than preparing a list of commands specifying the operations to be carried out, programmers had to set the program switches on the individual units manually according to requirements of a given problem.

For example, consider the basic task of adding the number in one accumulator to another. The analogous operation on the ASCC would have been coded by a single command that specified the addresses of the two registers. When executing this command, the central control unit would ensure that the in and out relays on the relevant registers were correctly set, the addition would be carried out, and the control would move on to read the next command from the input tape.

On the ENIAC, by contrast, the programming involved in this operation would be distributed across the two accumulators involved. The program switches on the first accumulator would be set to transmit the contents of the accumulator, and on the second to receive a number and so add it to its existing contents. These program switches would be set manually before the computation started. At the appropriate point, a program pulse would be sent to the appropriate switches on both of the accumulators and the number would be transferred. Either accumulator could then be set up to transmit a program pulse to the unit which had been set up to perform the next operation in the program.

In the absence of a central control unit, then, the ENIAC had to be physically configured to execute the operations required for a given computation in the correct sequence. Operations were performed when a unit received a program pulse, so a key factor in programming the ENIAC was to connect the various units together in such a way that program pulses would be transmitted at the correct time to the correct units. This was done by physically wiring the relevant units to the program lines, which connected all the units of the machine; this wiring was done manually before a computation started.

Given this, it would not have been practical or useful to write down a program for the ENIAC as a list of commands. Instead, what were essentially wiring diagrams were used, showing the connections that had to be made between functional units and program lines to ensure the correct sequencing of operations. In addition, it was necessary to give the setting of the program controls on the units involved in the computation.

Figure 5.1 shows how a calculation involving the constant transmitter and three accumulators could be defined.[15] At the beginning of the calculation, accumulators 18, 19 and 20 hold the values n, n^2 and n^3, respectively; the aim is to replace these values with $n+1$, $(n+1)^2$ and $(n+1)^3$. The approach adopted is to make use of the relationships $(n+1)^2 = n^2 + 2n + 1$ and $(n+1)^3 = n^3 + 3n^2 + 3n + 1$ and to form the new values by simple addition. The constant value 1 which will be required is set up on the constant transmitter.

[15]This diagram is adapted from an example given by Goldstine and Goldstine (1946), p. 108. The major difference is that Fig. 5.1 shows the program pulses being passed directly between units whereas the original shows the program lines explicitly, making it harder to following the routing of the pulses.

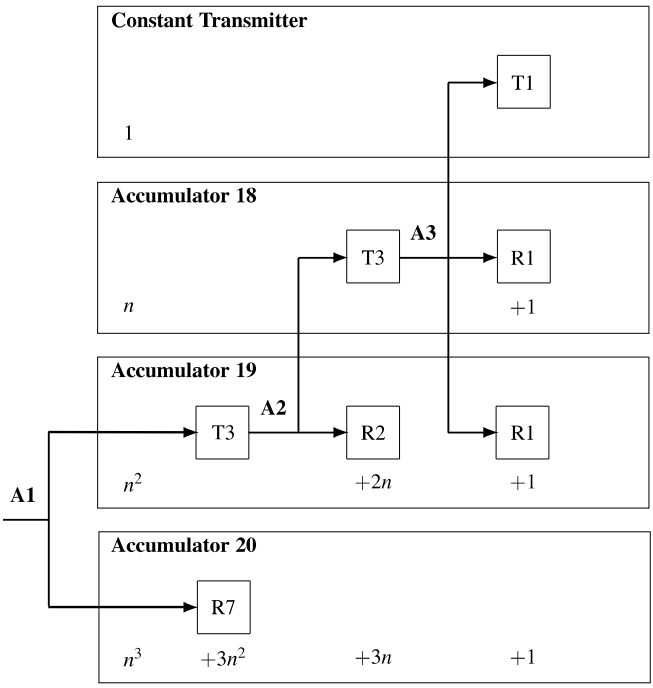


Fig. 5.1 A simple computation on the ENIAC

Each unit involved in the calculation is represented by a box, within which are shown the program controls that will be used. In this example, the program controls are simply set to transmit (T) or receive (R) a number; this number of times this operation will be repeated is also shown. The digit trunk is not shown: it should be remembered that any number that is transmitted will be sent to all units, but only those whose program controls are set to receive a number at that moment will do anything with it. All other units will ignore it.

The computation proceeds in three stages, and three program lines are used to transmit the program pulses which move the machine from one stage to another. Figure 5.1 is not a realistic wiring diagram: the lines labelled A1, A2 and A3 do not represent the program lines explicitly, but simply show the effect of the connections made in routing the program pulses from one unit to another. As a result, the diagram makes it clear that the output of one program control could be connected via a program line to the inputs of more than control, and hence distribute a program pulse to more than one unit.

The number stored on each unit is shown at the bottom of each box, along with the changes made to this value as the computation proceeds. To start the calculation, a program pulse is sent on line A1 to program controls in accumulators 19 and 20. As a result, accumulator 19 is set to transmit its contents three times, and accumulator 20 is set to receive seven numbers. The effect is that n^2 is added three times to accumulator 20, following which it is still set to receive four numbers.

Once this transmission is over, accumulator 19 emits a program pulse on line A2. This is sent to accumulator 18, which prepares to transmit its contents three times, and also to accumulator 19 itself, which prepares to receive two numbers. Accumulator 18 then transmits its contents, n, three times: this is added twice to the contents of accumulator 19 and three times to accumulator 20, which after this is still set to receive one more number.

Once this transmission is over, accumulator 18 emits a program pulse on line A3, which is sent to the constant transmitter and accumulators 18 and 19. As a result, the constant transmitter transmits the number 1 once, and this is added to all three accumulators. At this point there are no outstanding transmit or receive operations set on any unit, and the computation finishes.

Cycles and the Master Programmer It is easy to see from this example how a simple cycle of operations could be set up. If the output of the program control on the constant transmitter in Figure 5.1 was connected to program line A1, the program pulse it emitted would be detected by accumulators 19 and 20 and the computation would start again from the beginning. This would set up an endless repetition of the operations involved, however. It was the job of the master programmer to control cycles which would only repeat a fixed number of times.

Suppose that it was required to repeat the calculations shown in Fig. 5.1 and to print out the values in the accumulators after every seven repetitions. In addition, the whole computation should stop once 200 sets of results have been printed. A diagram illustrating the use of the master programmer to enable this level of control is shown in Fig. 5.2.[16]

The master programmer contained ten units known as *steppers*, which received, counted and emitted program pulses. Each stepper had a number of counters on which the number of program pulses received was recorded. In addition, a stepper could be in one of six *stages*, and a number could be associated with each stage, by entering it manually on switches provided on the master programmer. Two steppers are shown in Fig. 5.2: stepper 1 has the number 200 set on its first stage, and 1 on its second, and stepper 2 has 7 set on its first stage and 1 on its second. Unused stages are all set to zero.

Each stage of a stepper had its own output connection, and the output connection that a program pulse was sent to depended on the current stage of the stepper and the number of pulses received. At the beginning of a calculation, all steppers were in stage 1 with the count set to zero. So long as the count was less than the number set for the current stage, the program pulse would be emitted at the output connection for that stage. When the count reached the stage total, however, the stepper advanced to the next stage, and the count was reset to zero. Stepper in Fig. 5.2, for example, will first receive 200 pulses in stage 1, emitting the pulses to the input of stepper 2. After 200 pulses, however, it will move to stage 2.

[16]This diagram is adapted from an example given by Goldstine and Goldstine (1946), p. 109. As in Fig. 5.1, the routing of program pulses between units is shown directly, Note that the labelling of the program lines is consistent with the original diagram, and that the program pulses are emitted in the order A1, A2, A3, A5, A6, A4.

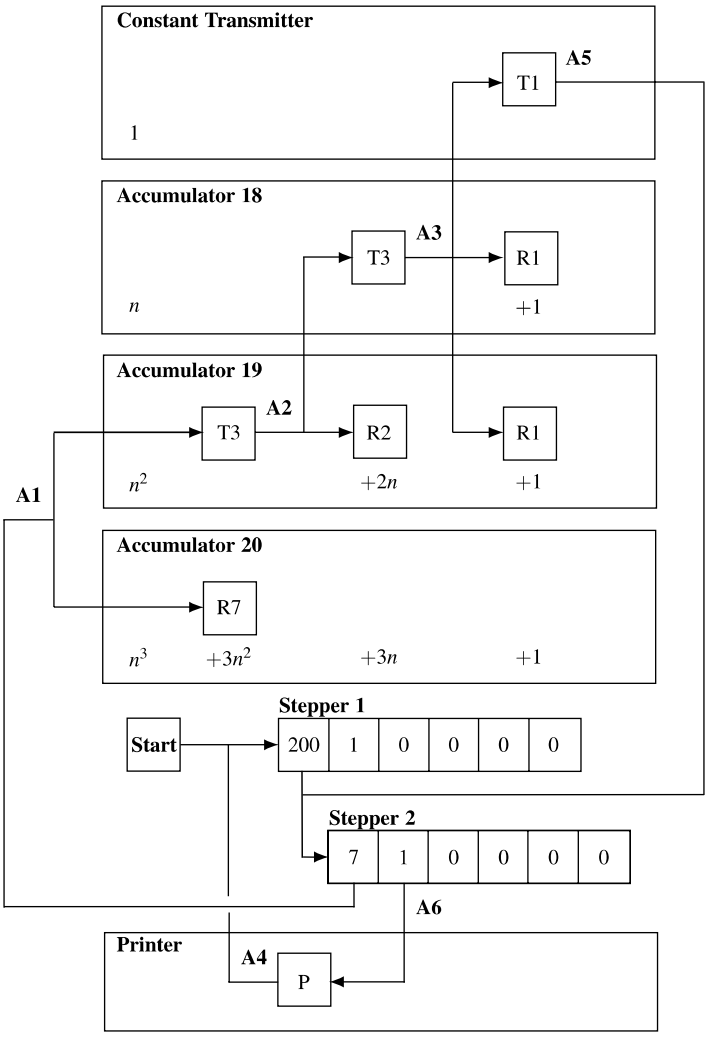


Fig. 5.2 A looping computation on the ENIAC

The overall flow of the computation shown in Fig. 5.2, then, is as follows. The first program pulse is sent from the initiating unit to stepper 1, which is in stage 1. The pulse is then sent to stepper 2, which is also in stage 1 and so emits it on program line A1, causing the operation in the cycle to be performed for the first time. Once the operations are completed, a program pulse is sent from the constant transmitter on program line A5 to the input of stepper 2. The count on this stepper will be increased to 2, and as it is still in stage 1, a pulse will be emitted on line A1 and the operations performed again. This cycle will be repeated seven times; stepper 2 will then move to stage 2 and the program pulse will instead be emitted on line A6 and sent to the printing unit. Once printing is complete, a pulse will be sent on line

A4 to stepper 1 which will increment its count and send a pulse back to stepper 2. Stepper 2 will have been reset to stage 1 at this point, and so another cycle of seven calculations followed by a print operation will be performed. This whole process will be repeated 200 times, and stepper 1 will then move to stage 2. As this marks the end of the computation, the output from this stage is not shown on the diagram.

The basic capability of the master programmer, therefore, was to enable the ENIAC to be set up to perform cycles of operations a fixed number of times. Each stepper could program a sequence of six such cycles, and cycles could be nested to a maximum depth of 10 by connecting one stepper to another as shown in Fig. 5.2.

Conditional Execution Like the Z3 and ASCC, the original design of the ENIAC made no provision for affecting the flow of control based on the values calculated so far. However, experience soon made clear the necessity for such a capability in most programming tasks, and modifications were made to the machine to provide basic discrimination facilities.

The basic modification was to provide each stepper with a *direct input* which moved the stepper from one stage to the next regardless of the number of pulses counted so far. Furthermore, it was arranged that digit lines as well as program lines could be attached to the direct input. So, for example, a digit line representing the sign of a number could be connected to a stepper's direct input in such a way that if the number was greater than zero, the stepper moved on to the next stage. Depending on the output connections made, this meant that the ENIAC could perform different operations depending on the sign of a particular number in an accumulator.

5.4 The Bell Labs Relay Machines

In 1938 George Stibitz, who worked at the Bell telephone company's laboratories, suggested the possibility of building calculating machines using standard telephone relays in conjunction with teletype apparatus. A prototype 'complex computer' was constructed which was capable of adding, subtracting, multiplying and dividing complex numbers; this machine was demonstrated in 1940. During the war, further experience was gained by building two special-purpose machines, a so-called 'relay interpolator' and a 'ballistic computer'.

In 1944 the decision was taken to develop a general-purpose computing system using relays, and by the beginning of 1948 two copies of this machine had been built, one of which was installed at the National Advisory Committee for Aeronautics and the other at the Ordnance Department at the Aberdeen Proving Ground. These machines were on a much larger scale than their special-purpose predecessors, and provided a greater degree of automation.

In many ways, the overall purpose and design of the Bell Laboratories Relay Computing System, as it became known, resembled those of the ASCC, though

there were many differences of detail.[17] It was intended to support the increasing need for numerical computation in research and development, and was thought of as an 'adding machine' that could be controlled to perform the four basic arithmetical operations as required.

The Design of the Bell Labs Machine Each of the two installations of the Bell Labs machine was viewed as a computing 'system' comprising two computers. The computers in a system were linked together, and results could be passed between them, depending on the needs or the scale of the computation being undertaken. Program tapes could be set up on independent tape readers which were connected to a computer when ready; in this way, the operation of the machines was interrupted as little as possible by the slow manual process of mounting tapes.

Each machine contained 15 storage registers and eight 'sign registers', which just recorded the sign of a number. A single calculating unit provided support for the five mathematical operations of addition, subtraction, multiplication, division and the extraction of square roots. Unlike the ASCC and the ENIAC, the registers were purely storage devices without even the capability to perform addition, and this single calculator was used to carry out all numerical operations.

A problem for the machine was coded on a number of input tapes, consisting of a single problem tape, and up to five subsidiary routine tapes. A problem control unit read the problem tape, directing the overall behaviour of the machine and also reading numeric data from the tape. Program orders were read and executed by the routine control unit, which had the ability to carry out certain conditional orders. Precomputed function data could be read from a number of table tapes, and results could be printed on teletypes or punched onto paper tape.

A more complicated form of control was provided by a special unit called the discriminator, which could receive and record the signs of numbers. Control of the computation could be made to depend on the number of signs received, or on certain aspects of the pattern of signs, such as a sequence of minus signs uninterrupted by a positive sign. This rather baroque device could be used to control the situations under which iterations could be terminated or restarted. However, descriptions of the machine do not make it clear whether the discriminator was controlled by orders on the routine tapes, or whether it was set up for a calculation and directly influenced control when the specified circumstances obtained.

Programming the Bell Labs Machine The overall control of computations was defined on open-ended problem tapes which could be read in one direction only. They were divided into a number of 'problem control sections' (PCS), each of which was further divided into subsections of three types. 'Cycle control sections' (CCS) contained the numeric data that were required for a specific computation; orders that needed to be carried out only once during a computation could be coded in 'switch

[17] The Bell Labs machines were described in a presentation by Samuel Williams at the 1947 Harvard conference, and in an extended paper by Franz Alt published in the following year. See Williams (1947) and Alt (1948a, 1948b) for details.

to routine' sections (SWR), and instructions to the printer, for example to control the production of headings for the printed output, were coded in 'switch to printer' sections (SWP). Routine tapes contained instructions only, and table tapes contained precomputed function values, arranged in a hierarchical structure of 'pages' and 'blocks' designed to make the necessary values easy to retrieve.[18]

A computation was started by reading the problem tape. Any instructions found in SWR and SWP sections were immediately executed, but when a cycle control section was encountered, control was passed to the first routine tape. Routine tapes were usually endless loops, and orders would be read in until control was eventually passed back to the problem tape. In the course of this process, numbers could be read from the current CCS on the problem tape, and control could be passed to other routine tapes.

The commonest type of order was one which instructed the machine to perform an arithmetic operation on the numbers stored in two registers, storing the result in a third register. These orders were coded in single instructions of the following form:

$$AC - CH = B\overline{EO}.$$

In this code, the 15 registers in the machine were denoted by the letters A to O, and the example above instructs the machine to subtract the number in register C from that in register A, storing the result in register C. The meaning of each symbol in an order was dependent on its position. The letters C and H appearing after a register name instructed the machine to clear or hold (preserve) the number in that register, respectively. The special symbol $\overline{EO}$ denoted the end of the order.

Some instructions only required reference to two rather than three registers. To calculate the square root of the number stored in register A and store it in register B, for example, the following instruction might be written:

$$AC\surd = B\overline{EO}.$$

Another type of instruction that only required the specification of two registers caused a value to be moved from one register to another:

$$AC \equiv B\overline{EO}.$$

In all cases, the names of the source register or registers were followed by a C or H to specify what should happen to the original contents of the register after the operation was complete. If an attempt was made to store a number in a register that was not cleared, an error condition was raised and the machine halted.

Some orders enabled the future course of a computation to be determined on the basis of values calculated so far. A decision could be made based on the sign (positive or negative) of the number stored in a register, or on whether such a number was equal to or different from zero. Instead of simply continuing to the next order, the machine had the capability of switching to a different section of a routine tape, switching to a different problem, or halting the computation.

[18]The terminology of pages and blocks reflects the extent to which table tapes were seen simply as an automated version of existing manual tables.

These choices were coded as an extended version of the transfer order illustrated above. For example, the following order examines the sign of the number stored in register A:

$$AC \equiv PRB\overline{EO},$$

continuing to the next order if it was negative, but moving to beginning of section 2 of the current routine if it was positive.

5.5 The Significance of the Automatic Calculators

The primary purpose of the machines described in this chapter was to automate scientific calculation, and to a larger extent than in any previous devices, this aim was indeed achieved. It is striking, however, how much manual intervention was still required in order to carry out a significant computation. On the ASCC, for example, operators were required to plug the machine in specified ways, to change tapes at certain points, and to watch out for specified events such as the machine stopping when a programmed check failed.

Functional Scope All the machines carried out numerical computation of the kind needed in scientific and engineering applications. These included the solution of systems of differential equations, numerical solution of differential equations, and perhaps most notably the production of tables using the well-known methods of interpolation and differencing. A major consumer of this new computing power was the Ordnance Department of the US Army, which required the production of firing tables for the many new weapons developed at its Proving Grounds at Aberdeen, Maryland, during the war.

The development of novel techniques in numerical analysis meant that the only operations required to perform such computation were the fundamental arithmetic operations of addition, subtraction, multiplication and division. The basic pattern of a computation was to carry out an extended sequence of these basic operations, along with the marshaling and distribution of the input data provided and the results that were produced.

These machines were therefore required to automate the process of carrying out a defined sequence of these basic operations, a goal which also required some way of storing and retrieving the numbers to be operated on. It is notable that this was essentially the same goal that Babbage had in the design of the Analytical Engine. Aiken in particular was very conscious of this continuity, to the extent of including a paper on Babbage's work in the session entitled 'Existing Calculation Machines' at the conference he organized in 1947.[19] However, it does not appear that prior knowledge of Babbage's work was in fact influential in the design of the ASCC and other machines.

[19]The Navy Department Bureau of Ordnance and Harvard University (1947).

Within the scope of this overall project, different choices were made about which operations counted as 'basic'. For example, after designing a dividing unit for the ASCC, Aiken claimed that it would have been more efficient to have provided a unit for finding the reciprocal of a number, and then performing division by multiplying by the reciprocal. The Bell Labs machine, on the other hand, provided the extraction of square roots as a fifth basic operation.

Analogy with Manual Computation In the 1930s, scientific computation was still largely carried out by human computers, with the aid of calculating machines of the register or punched card types, as discussed in Chap. 3. Despite the efforts of Comrie and Eckert, the paradigmatic situation was that of single human computer carrying out a paper-based process, and using mechanical assistance for carrying out individual numerical operations.

This paradigm served, explicitly or implicitly, as the basis for the design of the automatic calculators. The analogy was made explicit by Samuel Williams, writing in 1947 about the Bell Labs machine.[20] Williams presented two adjacent diagrams of virtually identical structure, one depicting the process of manual computation, and the other the design of the Bell Labs machine itself.

In the manual scenario, the computation was specified by an algebraic *formula* together with some numerical *problem data*. The actual computation was carried out by a *calculator* capable of performing at least the basic arithmetical operations; in the course of a computation, numbers might be stored on a *work sheet* and *tables* would be used to look up the precomputed values of the functions involved in the formula. The results of the computation would be recorded on an *answer sheet*.

In the design of the automated machine, the control function performed by the formula was instead carried out by a *routine control* unit, and the numeric data were provided by a *problem control* unit. A calculator provided the same functionality as in the manual case, but *storing registers* took the place of the work sheet and a *table control* unit provided precomputed function values. Finally, a *printer control* unit took care of the production of the final results.

This analogy makes it clear what the scope of the new machines was. Control of the manual computation was the task of the human computer, responsible for interpreting the formulae provided and making appropriate decisions at key points about how to proceed. In the design of the machine, however, control is built in: four of the seven components shown by Williams have the word 'control' in their titles. Furthermore, the human capacity to make decisions is replaced by a new unit, called the *discriminator*, which is fully integrated into the machine structure.

Physical Representation of Computational Structures A striking feature of the machines considered in this chapter is the way in which the overall structure of a computation was represented in the physical structure of the machine, rather than in the orders presented to it. The initial focus of the designers of these machines was on the computation itself, the sequence of mathematical operations carried out by

[20] Williams (1947).

the calculator. Automatic sequence control meant, in practice, giving the machine a list of commands on a paper tape or cards: the machine would then read these commands and perform the associated operations.

Even this was not a universal characteristic, of course. On the ENIAC, even the sequencing of operations was represented physically, being defined by the way in which the accumulators and the other units of the machine were connected to the program lines.

There was initially no clear distinction between the list of commands provided to the machine and the operations performed. However, it soon became clear in practice that something more complex than a simple correspondence between the two was required to make best use of the machines and to ease the task of coding for them. In particular, the fact that many computations were iterative in structure and required the repetition of a relatively small set of commands was soon recognized.

However, the command tapes of the ASCC and the Bell Labs machines could only be read in one direction: it was not possible to reverse the tape to go back to an earlier point and repeat a sequence of commands. One way to address this problem, adopted on the ASCC, was simply to punch the required commands on the input tape as many times as necessary. However, this was clearly a time-consuming and error-prone practice, and also involved additional preparatory work to ensure that the number of repetitions was adequate for the purposes of the computation.

An alternative approach was to make a tape 'endless', for example by gluing its ends together. The commands on such a tape would be executed over and over again until a suitable check stopped the entire computation. This basic approach was also followed on the Bell Labs machine's routine tapes.

The designers of these machines converged on a common view of the overall structure of a typical computation This consisted of a main sequence of instructions, which would be run through once, and a number of subsidiary sequences, which would usually be executed repeatedly. In one way or another, this structure was reflected in the physical design of the machine.

Tables A further feature shared by many of these calculators, and one that clearly links them with the existing practice of manual calculation, is the use that they made of tables of functions. Tables were a key resource for human calculators, one which provided easy reference to thousands of precomputed values for many common and application-specific functions. The design of all the automatic calculators discussed in this chapter included specific hardware components to replicate this ability.

The production of tables for use in manual calculation was a key application of most of these machines. The development in the Second World War of many new types of ordnance meant that there was a great need to produce firing tables for new weapons. Outside of military applications, the ASCC for example was for a number of years primarily used for the production of new tables of Bessel functions.

These examples show how these machines, far from representing a revolutionary break, represented an evolutionary step in computing practice. Current practices were inscribed in their very design, and in application they added to the available computational power, but without changing its nature.

In practice, it soon became apparent that there was a need for commands which could affect the course of a computation. The ASCC, for example, had commands for the use of the check counter, and later for switching control between sequence mechanisms. It is notable, however, that these facilities were for the most part not foreseen or provided as part of the original design of the machines.

This is compelling evidence that the need to think about how to automate control beyond the requirements of a simple sequence of operations was far from obvious to the designers of these machines. In particular, the idea of defining a code which would contain both arithmetical and control operations, which seems so natural in hindsight, was far from obvious to computer designers in the 1940s.

Subsequent Developments Although computers based on relays were destined to be made obsolete by the vastly greater computational speeds available on electronic machines, developments on these projects continued throughout the 1940s and early 1950s. Electronic machines were at this time still highly experimental and untested, and the relay machines represented the most significant source of computational power.

In 1942, for example, Zuse started work on a further machine, the Z4. This had a similar overall design to his earlier machines, having a mechanical memory and electro-mechanical arithmetic and control units. This machine had rather an exciting life. Towards the end of the war, when it was almost completed, it was evacuated from Berlin to Göttingen to protect it from air raids. It was assembled and ran a few calculations, but was then moved again to an underground ordnance factory in the Bavarian Alps; as the war came to an end it ended up in storage in the village of Hopferau, where Zuse and his family were then living.

In 1949, however, Eduard Stiefel from the ETH in Zurich heard that the Z4 had been saved, and decided to acquire it for use at the ETH. In 1950, Zuse made a number of modifications to the machine that had been requested by Stiefel. The most significant change made to the programming of the machine was the inclusion of two conditional instructions.[21]

Zuse's original design had included Start, Jump and Stop instructions. The Z4 would read instructions from the tape, but not execute them until a Start instruction was encountered. A Jump instruction would cause the machine to skip instructions until another Start instruction was encountered, and a Stop instruction would bring the computation to a halt. Stiefel's modifications provided versions of the Jump and Stop instruction which would only take effect when the number in the first register of the calculating unit was negative.

The machine was moved to Zurich in September 1950, and remained in continual use for solving problems in numerical analysis until 1954.

[21] Speiser (2000).

Chapter 6
Logic and the Invention of the Computer

The logical investigation of the concept of effective computability, described in Chap. 4, and the development of the machines described in Chap. 5 were largely independent of each other. In particular, Zuse and Aiken were primarily motivated by the desire to avoid having to perform long calculations by hand and were, at least initially, ignorant of developments in logic. Zuse, for example, came up with what he thought was a novel notation to describe certain features of the design of his machine, only to be told later that he had in effect rediscovered the propositional calculus.[1]

By contrast, at least some logicians were aware of the importance of practical computation. By the 1930s, calculating machines and punched card machinery were widely used in industry and commerce, and techniques for organizing large-scale scientific calculations were being developed. 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.[2]

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, the machines described in Chap. 5 were obsolescent, partly because of their limited computational capacity, but also because the design principles on which they were based had been superseded. Of particular significance was the adoption of the so-called *stored-program* design: unlike the earlier machines, which, with the exception of the ENIAC, read their instructions from an external medium such as punched cards or paper tape, the later machines held instructions internally, in the same medium that was used to store the data being operated on. The stored-program concept was evolved by the ENIAC group in collaboration with the

[1] Zuse (1993), p. 46.

[2] Agar (2003).

M. Priestley, *A Science of Operations*, History of Computing,
DOI 10.1007/978-1-84882-555-0_6, © Springer-Verlag London Limited 2011

mathematician John von Neumann, who joined them on a consultancy basis in 1944, and was first described in a proposal describing the ENIAC's successor, known as the EDVAC.

Much recent historical writing has tended 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,[3] as have the links between the office automation industry and the post-war computer.[4] However, even in this broader historiographical tradition, the EDVAC is seen as a watershed. In their history, entitled simply *Computer*, Campbell-Kelly and Aspray devoted a chapter entitled "Inventing the Computer" to the topic. 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, and Ceruzzi later wrote that "the stored-program principle remains a valid focus for computing's history".[5]

During the 1940s, the logical and practical approaches to computation became increasingly intertwined. From the war years onwards, Turing contributed to the development of various machines in Britain, and in the United States von Neumann was centrally involved in the planning and construction of new machines, starting with the EDVAC in 1944. After 1950, it became common to describe electronic digital 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 way in 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 on the way to the development of the 'computer as we know it', an innovation in which logic played a crucial role in guiding the direction of development. This view was clearly expressed by the historian of computing, Michael Mahoney:

> 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.[6]

The image of the confluence of two separate lines of research has been used by a number of other writers, such as 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".[7] In her account, the two strands had been separated by the increasing formalization of mathematics since the nineteenth century and were then reunited by the practical demands of the war.

---

[3] Aspray (1990a).

[4] See Aspray (1990a) and Agar (2003).

[5] Ceruzzi (2001), p. 51.

[6] Mahoney (1988), p. 116.

[7] Peláez (1999), p. 359.

Other writers have made a stronger assertion, namely that the confluence of these two strands was a *necessary* precondition for the emergence of the computer in its modern form. Stan Ulam, for example, a mathematician who became an computer user at a very early stage thanks to his involvement with the Manhattan project, wrote that "computer development *became possible only* by a confluence of at least two entirely different streams" and, in his influential biography of Turing, Andrew Hodges described 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".[8] Forceful arguments in favour of this theory have also been made by the logician and computer scientist Martin Davis.[9]

These arguments rely to a considerable extent on retrospective interpretation, however, and some contemporary sources suggest that a widespread belief about the importance of logic to the practical development of the computer only emerged some time after the fact. For example, in the mid-1950s 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.[10]

This chapter considers the role that logic played in the development of the stored-program computer, and the interaction between theory and practice in this process. A notable recent trend has been to emphasize the importance of logic. Mahoney, for example, strengthened the 'confluence' account, writing that "[a]s logic machines, the first stored-program computers … emerged as byproducts of theoretical inquiry into the nature and limits of logical thought".[11] This idea has been reinforced by Davis, whose writings give the impression that the first stored-program computers were created by means of a relatively straightforward implementation of Turing's abstract machines.

This line of thought leads to a strong conclusion about the nature or essence of the computer, namely that it can best be characterized by its relationship with logic, rather than by its relationship with numerical analysis or electronic engineering. Martin Davis expressed this view very clearly, writing that "a computing machine is really a logic machine".[12] This view gains some support from 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 can be explained by their origins in Turing's concept of a universal machine.

The best place to begin an examination of these views is by considering the events and the intellectual background that led to the development and introduction of the stored-program concept.

---

[8] Ulam (1980), Hodges (1983), p. 556, emphases added.

[9] Davis (2000).

[10] Wang (1957), p. 63.

[11] Mahoney (1989).

[12] Davis (2000), p. xii.

6.1 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.[13] The major practical effort during this period was the development of the ENIAC, described in Chap. 5, by a group of engineers led by John Mauchly and Presper Eckert.

The group soon recognized that there were several shortcomings in the design of the ENIAC, however, and during 1944 work started on a follow-up project. 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 fruitful collaboration, which led in 1945 to the writing of the *First draft of a report on the EDVAC*,[14] an internal report describing aspects of the design of a proposed successor machine to the ENIAC. Although it was not intended for publication, this report was widely circulated and is generally credited with articulating for the first time the high-level design principles underlying virtually all subsequent computers.

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.[15] 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.

The most significant perceived shortcoming of the ENIAC was not do with its construction or the design of its electronic circuitry, but rather the ease with which it could be programmed. As described in Chap. 5, the ENIAC had to be 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.[16]

The inconvenience of programming the ENIAC was soon recognized to be a significant limitation, however, and one of the goals for the subsequent project was to come up with a practical way of programming an electronic machine whose speed

[13] See Campbell-Kelly and Aspray (1996) and Ceruzzi (2003), for representative accounts.

[14] von Neumann (1945), hereafter simply '*Draft Report*'.

[15] Campbell-Kelly and Williams (1985).

[16] ENIAC (1943).

made the familiar approach of reading instructions from punched cards impractical. In January 1944, Eckert proposed a solution that was partly mechanical and partly electronic, making use of magnetic storage devices.[17]

Eckert's proposal envisaged a rotating shaft with a number of discs or drums of different types mounted on it. Some of these, the type (a) devices, would be able to be magnetized and demagnetized quickly, and therefore would provide "a method of storing, in some usable code, those characters or digits which must be used later or indicated". Type (b) devices, on the other hand, 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 Eckert was envisaging different storage media for data and program code, with numbers being stored on the volatile type (a) discs while the type (b) discs played the role of punched cards or paper tape in other machines, holding the program instructions. The proposal therefore clearly addressed the problem of reprogramming the ENIAC: the machine Eckert 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.

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

In mid-1944, the ENIAC group was joined on a part-time basis by von Neumann. Although originally a pure mathematician, von Neumann was 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. 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

[17] Eckert (1944).

there.[18] Following this visit, von Neumann wrote to Oswald Veblen that "I have also developed an obscene interest in computational techniques".[19]

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 seemed to be in a state enabling them to be of immediate use to Los Alamos, however. Von Neumann also gained first hand experience of the computational equipment then 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".[20]

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.[21] 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.[22]

Whatever the reason, it appears that von Neumann had not heard about the ENIAC project until he met Herman Goldstine, apparently by chance, who told him about his involvement in the development of an electronic calculating device.[23] Well aware by this time of the limitations of electromechanical technology, von Neumann was quick to appreciate the potential of the increased computation speed promised by the electronic ENIAC and its planned successor, and soon became involved with the Moore School 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 existing orientation of the ENIAC group towards the applications of automated calculation in areas such as ballistics and meteorology.

---

[18] Todd (1974).

[19] Aspray (1990b), p. 27.

[20] Aspray (1990b), p. 33.

[21] Stern (1981).

[22] Aspray (1990b), p. 35.

[23] Goldstine (1972), p. 182.

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.[24]

A second report, in September 1945, goes into more detail about the historical background of the project, and claims that the stored-program concept dates from Eckert's 1944 proposal, 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.[25]

The attribution of credit for the invention of the stored-program concept has proved to be very controversial. Turing's universal machine has been retrospectively 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.[26] 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, the other computer developers in the USA do not seem to have considered storing data and instructions in the same store. Like Zuse's machines, the ASCC and the Bell Labs machines were programmed by means of externally supplied programs. In 1940, Norbert Wiener described a computing machine which would use electronic technology and store data on a rewritable tape, in a manner very reminiscent of Turing's machines.[27] Wiener was thinking of a special-purpose machine, however, and despite incorporating many of the features of the post-1945 machines, his account did not include any mention of a stored program.

---

[24] Eckert et al. (1945).

[25] Anonymous (1945).

[26] Zuse (1993), p. 53.

[27] Wiener (1940).

Testimony from the members of the Moore School group themselves is mixed, and is coloured by the fallout from a split between von Neumann and Eckert and Mauchly. Goldstine and Arthur Burks, who both had a background in mathematics and logic, 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 credited primarily for his work on logical control and coding.

The documentary evidence, summarized above, does not give an unequivocal answer to the question of who first came up with the stored-program idea. Drawing on Eckert's 1944 disclosure in particular, some writers have concluded that priority should be assigned to Eckert and Mauchly.[28] 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 taken 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".[29] The relationship between logic and the stored-program concept is discussed further in Sect. 6.4.

6.2 The Early Development of Cybernetics

The previous section described the origins of the idea of the stored program in the context of the field of automated calculation and the search for a feasible method of programming electronic computers. Logic seems not to have played an explicit role in this process, but there was one relevant area where logic and computers did come together at this time, namely in 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 was spending the years between 1936 and 1938 working with Alonzo Church. In his thesis, William Aspray related 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 interest 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 Neu-

[28] Stern (1980), Metropolis and Worlton (1980).

[29] Ceruzzi (1998), p. 22.

> mann and Turing separated when Turing returned to England, leaving both determined to build computers to test the possibility of mathematically modelling the human brain.[30]

There are a number of anecdotal references to von Neumann's interest in Turing's work and the high regard in which he continued to hold 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.[31] 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".[32]

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 beginning of the war started work at MIT's Radiation Laboratory with Julian Bigelow, an engineer, investigating ways of making antiaircraft artillery more effective.[33]

As the speed of aircraft increased, human gunners were unable to track them visually, and the accuracy of fire was decreasing; given the strategic importance of aerial bombardment in modern warfare, this was a significant problem. Wiener and Bigelow developed new mathematical methods enabling an aircraft's future position to be more accurately predicted given the details of its course so far. This process involved a feedback loop: an initial prediction would be used to aim the gun, and information about the accuracy of the resulting fire would be used in the calculations leading to the next prediction.

Wiener and Bigelow described the more philosophical aspects of this work in an essay written with the physiologist Arturo Rosenblueth, a friend of Wiener's. This paper described an approach to the study of behaviour which emphasized the observable inputs to and outputs from an object; this 'behaviouristic' approach was contrasted with a 'functional' approach which described behaviour by giving an account of the internal structure and workings of objects. As a result, Wiener and his collaborators were able to argue that "a uniform behavioristic analysis is applicable to both machines and living organisms".[34] One motivation for this approach was that it offered a uniform approach to studying systems, such as antiaircraft batteries, that had both human and mechanical components.

The paper outlined a taxonomy of behaviour, and gave an account of the notion of acting with a purpose. The authors concluded that "[a]ll purposeful behavior may be considered to require negative feedback", a principle which was later commonly seen as encapsulating the central message of cybernetics.[35] Throughout, the paper

[30] Aspray (1980), pp. 147–148.

[31] Aspray (1990b), p. 178.

[32] Randell (1972), p. 10.

[33] Heims (1980), p. 182ff.

[34] Rosenblueth et al. (1943), p. 22.

[35] Wiener (1948), p. 19, Wisdom (1951).

stressed the similarities between the behaviour of machines and organisms. In the absence of "any one or more qualitatively distinct, unique characteristics present in one group and absent in the other", the behaviourist approach allowed both to be studied in the same way, the authors further observing that "[s]uch qualitative differences have not appeared so far".[36]

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".[37] 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".[38]

Von Neumann read this paper in 1943, apparently at the recommendation of Wiener and Bigelow, and according to Bigelow, was "enormously impressed" with the work of McCulloch and Pitts.[39] 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".[40] 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.[41]

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 computer programming, and so will not be discussed further here.

---

[36] Rosenblueth et al. (1943), p. 22.

[37] McCulloch (1948).

[38] McCulloch and Pitts (1943), p. 129.

[39] Aspray (1990b), pp. 180, 313, note 23.

[40] von Neumann (1948), p. 412.

[41] Wiener (1945).

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. However, he was visited by Wiener and McCulloch during the war, and spent the early months of 1944 in the United States, visiting Claude Shannon at Bell Labs.[42]

6.3 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 the neuronal structures of the brain. 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*". This term was then explained as denoting "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 wrote later that "the device is primarily a computer",[43] reinforcing the idea that the EDVAC was thought of as a machine to automate mathematical calculation; the *Draft Report* nowhere suggested any wider uses for it.

Von Neumann next addressed the overall structure of the machine. The design was based on a small number of relatively high-level components, each with one specific function to perform. This was very different from the design of the ENIAC, which was based upon a set of 20 identical accumulators, each capable of carrying out a number of different functions, such as 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 role.

The first component that was considered was motivated by 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.[44]

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

[42] Hodges (1983).

[43] von Neumann (1945), Sect. 1.1, emphasis in original; Sects. 1.2, 2.2.

[44] von Neumann (1945), Sect. 2.2.

of the machine viewed as a body. Like those of 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*".[45]

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—in existing devices this is done as indicated in 1.2—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*.[46]

In its description of a machine which will be supplied with and will carry out instructions for different computations, this passage is reminiscent of the universal machine described by Turing. Von Neumann does not draw attention to this analogy, however, and as the reference to "existing devices" makes clear, did not view this as a particularly innovative feature of the EDVAC design.

The identification of a CC again distinguished the EDVAC from the ENIAC, and placed it firmly in the mainstream of established practice in automatic computation. In virtually all preceding machines, including Babbage's Analytical Engine, Zuse's machines, the ASCC and the Bell Labs machine, the instructions for a program were read and interpreted by a single device which then configured the machine so that the requested operation was carried out.

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",[47] 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*.[48]

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

[45] von Neumann (1945), Sect. 2.2, emphases in original.

[46] von Neumann (1945), Sect. 2.3, emphases in original.

[47] von Neumann (1945), Sect. 2.4.

[48] von Neumann (1945), Sect. 2.5, emphases in original.

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 referring explicitly to the different roles of 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.[49]

One effect of this was to draw a further analogy between the internal components CA, CC and M of the EDVAC and the brain. This analogy was reinforced in the next section of the report when von Neumann came to consider the 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, 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. He commented that "[i]t is worth mentioning, that the neurons of the higher animals are definitely elements in the above sense",[50] thus reinforcing further the connection between brains and computers.

To clarify this, 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. 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 were 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.[51]

In other words, von Neumann was now presenting the 'machine as brain' metaphor as a substantial structural equivalence, and proposing 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.[52] The problem of understanding the functioning of the brain consisted 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,

[49] von Neumann (1945), Sect. 2.6, emphases in original.

[50] von Neumann (1945), Sect. 4.2.

[51] von Neumann (1945), Sect. 6.2.

[52] von Neumann (1948).

and the behaviour emerging from this organization. The two parts were linked by an 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.[53]

This discussion has demonstrated that the underlying cybernetic assumption of a fundamental analogy between natural organisms and machines was made explicit in the first presentation of the architecture of the modern computer, and von Neumann carefully demonstrated 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 Sect. 6.7.

The community of applied mathematicians who were the primary users of the early computers, and presumably very conscious of their limitations, were much less enthusiastic about seeing computers as brains, however, and in von Neumann's later reports on computer design, coauthored with Burks and Goldstine, little use is made of the analogy.[54] It continued to play an important role in von Neumann's thinking, however, notably in a paper entitled *The General and Logical Theory of Automata* which he presented to an audience of cyberneticians in 1948.[55]

The role of logic in the *Draft Report*, then, is rather indirect. Rather than locating the EDVAC in an explicitly logical tradition, as Turing had done with 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.

6.4 Logic and the Stored-Program Concept

Seen in its historical context, the *Draft Report* appears to be a rather conservative document, in terms both of the design and the intended purpose of the machine it describes. The EDVAC was part of a very practical programme of research into automatic computation, and like earlier machines it was intended to be primarily a calculator: the problems of manual calculation provided the impetus which led several individuals to enter the field, and the importance of this area was greatly amplified by the military demands of the war. The details of its design owed as much to machines such as the ASCC as to its immediate predecessor, the ENIAC, and for all that von Neumann had an interest in logic and Turing's proposals, the EDVAC project had started well before he became involved with it. It would oversimplify

[53] Turing (1946).

[54] Goldstine and von Neumann (1946), Burks et al. (1946).

[55] von Neumann (1948).

a complex historical situation to suggest, with Mahoney and Davis, that the stored-program design emerged simply as a byproduct of theoretical research in logic.

A weaker claim, however, might be that specific features of the design were adopted for reasons of logic, and a number of aspects of the *Draft Report* have been characterized as being 'logical' in one way or another. For example, Aspray has described one 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.[56]

While this is probably a fair description of the contribution von Neumann made to the presentation of the EDVAC design, it should be remembered that the abstract description of a computer's architecture as a set of functionally distinct subsystems was not original to the *Draft Report*. 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. The store and mill of Babbage's Analytical Engine are perhaps the earliest example of such an architecture, 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 were 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",[57] 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,[58] but again, this does not point to a specific influence of logic on the design of the EDVAC.

Perhaps the strongest argument in support of the influence of logic derives from the discussion in the *Draft Report* of the stored-program concept, the idea of storing both the instructions and the data of a program in the same memory. This is widely taken to be the key feature that distinguishes 'modern' computers from their predecessors. The argument goes on to point out that Turing's universal machine seems to incorporate this idea: the tape of the universal machine contains a representation of the machine being simulated, as well as the other data required by the simulation. Given von Neumann's familiarity with Turing's work, the argument goes, we can credit him with bridging the gap between theory and practice, and giving logic a crucial role to play in the invention of the computer.

One objection that can be made to this is the observation that the use of a single tape is not essential to Turing's arguments. A universal machine requires access to

[56] Aspray (1990b), p. 40.

[57] von Neumann (1945), Sect. 2.3.

[58] Shannon (1938).

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.

This point is sometimes obscured by a confusion between the idea of universality and the stored-program principle itself. 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",[59] a comment referring to the programmability of computers rather than the stored program itself.

The historical evidence supporting the idea of a direct influence from logic on the idea of the stored program is rather 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, seemed more concerned to link the new machine with the developing area of cybernetics 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. The accounts given by members of the EDVAC team do not stress the similarity of the two forms of stored information, numeric data and stored instructions, and nor do they give abstract or logical reasons for holding both in a single store; rather, the two forms of information were clearly distinguished and pragmatic reasons, firmly based on engineering arguments, were given for holding the two in a single store.

In the *Draft Report* itself, von Neumann suggested that the most important factor was the need to deal with the very high computational speeds that could be achieved in an electronic machine:

> Indeed, all existing (fully or partially automatic) computing devices use R—as a stack of punched cards or a length of teletype tape—for all these purposes [including the storage of instructions] … Nevertheless it will appear that a really high speed device would be very limited in its usefulness unless it can rely on M, rather than R, for all the purposes enumerated …[60]

A different argument was given by Eckert in one of the Moore School lectures in the summer of 1946.[61] 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 observed that different types of problem can have significantly

[59] Ceruzzi (2001), p. 50.

[60] von Neumann (1945), Sect. 2.9.

[61] Eckert (1946).

different memory requirements, in terms of the relative amount of space required for these numbers and orders. Maximum flexibility and economy in construction could be obtained by combining both in a single store, rather than providing separate memory components for each.

In another report dating from 1946,[62] 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.[63]

None of the writings originating from the Moore School group mention a logical or theoretical rationale for the introduction 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 justified by practical, not theoretical, arguments. A logical pedigree for the idea would not, on its own, have been sufficient to ensure its incorporation in the design without a detailed examination of its engineering implications.

6.5 The EDVAC Code and Address Modification

The real significance of the stored-program concept relates not to hardware design, but rather to software and the ease of programming. The *Draft Report* stipulated that orders were to be stored in the same memory organ as numbers, and recognized that this meant that both types of information were coded in the same way. This made it possible to treat orders as if they were numbers and to carry out operations on them, which in turn led to the possibility of a computation modifying the very instructions that a computer was following as the computation progressed.

This idea of *self-modifying code* turned out to be highly significant. In the *Draft Report* it was introduced rather tentatively, however, and the general idea of treating orders as data was only applied for the specific purpose of modifying the memory address to which a particular order referred.

---

[62] Goldstine and von Neumann (1946).

[63] Burks et al. (1946).

Memory Structure From the user, or programmer's, point of view, the EDVAC had a simple structure. The basic memory elements, known as *minor cycles*, consisted of 32 *units*, or binary digits. It was envisaged that the memory would be implemented on a number of mercury delay lines, each of which would store a number of minor cycles. A particular minor cycle was therefore be identified by two numbers, μ which identified a particular delay line, or *major cycle*, and ρ which identified a minor cycle on that line. The sizing considerations examined in the report suggested that 256 delay lines should be built, each providing storage for 32 minor cycles, or numbers.

A minor cycle could hold either a single number or an instruction. These two possibilities were distinguished by the value of the first unit in the cycle, i_0. A minor cycle where $i_0 = 0$ represented a number, and one where $i_0 = 1$ represented an order.

The Operations of CA The arithmetic unit CA contained storage for three minor cycles. Two of these, denoted by I_{CA} and J_{CA}, were thought of as input organs, and the third, O_{CA}, as an output organ. The basic principle was that when an arithmetic operation had to be carried out, its arguments would be transferred to I_{CA} and J_{CA}, and when the calculation was complete, the result would be available at O_{CA}. The transfer of data into CA was in fact performed in the following manner:

> Every real number coming from M into CA is routed into I_{CA}. At the same time, the real number previously in I_{CA} is moved on to J_{CA}, and the real number previously in J_{CA} is necessarily cleared.[64]

The operations that could be performed by CA were the following:

1. The basic arithmetic operations $+$, $-$, $\times$, $\div$ and $\surd$.
2. Two operations i and j to move data from I_{CA} and J_{CA}, respectively, to O_{CA}. These were intended, among other things, to facilitate the transfer of data from one location in M to another, by routing it through CA.
3. An operation s which "was 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".[65] The operation s would move either the contents of I_{CA} or J_{CA} into O_{CA}, depending on whether the number originally in O_{CA} was greater or less than zero.
4. Two operations bd and db which would convert numbers between binary and decimal formats, for input and output.

The Orders The orders that were defined in the *Draft Report* are summarized in Table 6.1. The report went into some detail about how the orders would actually be coded in the units of a minor cycle, and in addition a *short symbol* for each order was given. The short symbols were intended to be mnemonic and to make the reading and writing of orders easier.

[64] von Neumann (1945), Sect. 11.1.

[65] von Neumann (1945), Sect. 11.2.

Table 6.1 The orders of the EDVAC

Description	Short symbol
Carry out operation w in CA ...	
... and hold result in O_{CA}	wh
... and transfer result to minor cycle $\mu\rho$	$w \to \mu\rho$
... and hold result in O_{CA}	$wh \to \mu\rho$
... and transfer result to next minor cycle	$w \to f$
... and hold result in O_{CA}	$wh \to f$
... and transfer result to I_{CA}	$w \to A$
... and hold result in O_{CA}	$wh \to A$
Transfer number from minor cycle $\mu\rho$ to I_{CA}	$A \leftarrow \mu\rho$
Connect CC with minor cycle $\mu\rho$	$C \leftarrow \mu\rho$

The basic purpose of the code was to enable numbers to be moved between M and CA, and to specify what operation was to be performed on the numbers in CA. The order $A \leftarrow \mu\rho$ moved the contents of a specific minor cycle into I_{CA}. In addition, if the control organ came across a minor cycle which contained a number rather than an order, it would interpret this as a request to move that number to I_{CA}.

A number of different orders defined which operation of CA was to be performed and how the result of the operation should be handled. After an operation was complete, the number in O_{CA} could be transferred to a specific minor cycle in memory, or simply to the minor cycle following the one holding the current instruction. Alternatively, it could be reentered into CA itself, and either held in or cleared from O_{CA} itself.

Address Modification and Conditional Execution This code only defined a very limited form of instruction modification, which arose from the fact that the effect of transferring a number from O_{CA} to a minor cycle in M depended on whether that minor cycle currently held a number or an order; this could be detected by examining the contents of its first unit i_0. If the minor cycle held a number, its entire contents were replaced by the contents of O_{CA}. If it held an order, however, only the last 13 digits, representing the address of a minor cycle, would be copied.

The default operation of the control organ was to proceed through minor cycles sequentially, executing the orders found there one after the other. A control transfer order was defined to alter this default sequencing by switching the control organ to a specified minor cycle. This was an unconditional jump, however; programming a conditional jump, where the behaviour of the program depended on data values computed so far, required the use of the s operation and address modification.

Suppose that it was required to transfer control to one of the instructions c_1 or c_2, depending on whether a certain number was greater than zero. This could be accomplished in the follow manner. First, compute the number to be tested, and

leave it in O_{CA}. Secondly, move the addresses c_1 and c_2 into CA, and then invoke the operation s. Depending on the sign of the number in O_{CA}, this would replace it with either c_1 or c_2. Finally, transfer the new contents of O_{CA} to the minor cycle containing the transfer order; this would change the address in the transfer order which, when executed, would transfer control to c_1 or c_2, as required.

The code specified in the *Draft Report* was never in fact used. In subsequent publications, von Neumann and his colleagues modified it in various ways, and by the time the first stored-program machines came into regular operation in 1949, the codes used were considerably more sophisticated than this first attempt. Chapter 7 contains a general discussion of programming using this style of code.

6.6 Turing and the ACE

The importance of the *Draft Report* lay in the concepts and approach put forward in it rather than in the specific details of the design presented. Later in 1945, von Neumann developed an algorithm for sorting and collating data; in the course of this work he made several changes to the machine design and code presented in the *Draft Report*,[66] and subsequent designs, such as that described in 1946 for a machine to be built at the Institute of Advanced Study,[67] differed from the EDVAC in many details. Although they are important to the history of computer architecture, these alternative proposals did not introduce any significantly different approaches to the issue of coding. The situation is slightly different, however, in the case of a design produced by Turing in 1946.

In 1945, at the end of the war, Turing had joined the UK's National Physical Laboratory (NPL) in Teddington. He was given a copy of von Neumann's report, and by the end of the year had produced a proposal of his own outlining the design of a stored-program computer that he proposed the NPL should build, the Automatic Computing Engine, or ACE.[68] Turing's report is in many ways comparable in scope and ambition to the *Draft Report*, and a comparison of the designs presented in the two reports is a good way of highlighting the characteristic features of each.

The influence of the *Draft Report* is apparent in Turing's work, and in fact he recommended that the two be read together. In many details, the ACE is similar to the planned EDVAC. Both designs use mercury delay lines as the principal 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. Furthermore, Turing used 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

[66] Knuth (1970).

[67] Burks et al. (1946).

[68] Turing (1946).

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.[69]

As discussed above, von Neumann presented the EDVAC as fundamentally a calculator. By contrast, Turing made an explicit 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.[70]

He went 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.[71]

This difference in orientation is reflected in the design of the ACE in several ways, most noticeably in the way the machine is structured. At the beginning of the report, Turing described the ACE as consisting of 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.[72] 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.

The EDVAC's arithmetic unit 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 it 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. It was to contain a number of "quick reference temporary storage units (TS)" in addition to the delay lines,[73] 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

[69] Carpenter and Doran (1977), Peláez (1999).

[70] Turing (1946), p. 39.

[71] Turing (1947), p. 107.

[72] Turing (1946), pp. 21–22.

[73] Turing (1946), p. 22.

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 put forward 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 difference between the two reports concerns the way in which they viewed programs. Although its code supported loops and subroutines, in the *Draft Report* a program is thought of as being primarily a sequence of instructions which invoke the basic arithmetic operations provided by the machine; this is very similar to the conception of programming adopted in earlier machines such as the ASCC.

In contrast to this, Turing suggested that the basic operations required for the task being programmed should be defined as subroutines built from the ACE's primitive instructions. This is reminiscent of the procedure that he had followed in 1936, when 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 Chap. 4 that this technique was derived from existing practice in the definition of recursive functions, and the ACE proposal carries this approach forward into the sphere of practical computation. Turing recognized that this principle could be applied even to basic arithmetical operations: the arithmetic circuits in the ACE were not viewed as fundamental components, as they were in the *Draft Report*, but simply as a means 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. Von Neumann had clearly distinguished orders from numbers, 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 in general terms to the possibilities that would be created by allowing the machine to write its own orders. Although this possibility was not exploited by the universal machine, machine tables and data were distinguished on its tape by convention, rather than by explicit details of coding.

It seems plausible to suggest, then, that whereas von Neumann set out to design an automatic calculator, Turing was more concerned to produce a practical version of the universal machine which could, with the addition of some specialized circuitry, be used as a high-speed calculator. In a sense, the ACE could therefore be described as being more influenced by logic than the design of the *Draft Report*. In practice, however, the EDVAC design was vastly dominant. The ACE was never implemented in precisely the form described in Turing's report. The first machine completed at the NPL was the 'Pilot ACE', built on a smaller scale than Turing's proposed ACE and differing 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 ideas died out.

This raises the interesting question of why von Neumann's ideas about computer architecture, which were less directly derived from logic than Turing's, proved so much more successful in practice. One suggested answer to this question downplays internal factors, such as the increase in complexity inherent in Turing's approach to programming, and suggests that the primary reason for the greater success of the EDVAC design was its "instrumentality".[74] By emphasizing the provision of high-speed calculation, the argument goes, von Neumann addressed an immediate social need, and the EDVAC design was therefore quickly picked up as providing a solution to a pressing 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 contributory factors included 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 the involvement of von Neumann himself. Furthermore, the *Draft Report* presented the EDVAC as a relatively straightforward evolution from well-known machines such as the ASCC, 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. This suggests that a connection between stored-program computers and the universal machine was not widely made in 1946: this point and its implications are considered in more depth in subsequent sections.

6.7 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 the reception and representation of computers can be traced 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, the journal of the American Institute of Electrical Engineers contained an article which discussed the "Impact of the War on Science" which did not, however, make any reference to computing technology.[75] Later that year, though, 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

[74] Peláez (1999).

[75] Briggs (1946).

> mechanical and electrical computing equipment which, in terms of speed and flexibility, will completely outdistance anything thought of before.[76]

The new machines, in particular the ENIAC and the ASCC, were widely reported in the press. One prominent aspect of the coverage was the analogy drawn between high-speed calculators and the brain. During the war, it had been common to refer to various devices 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.[77]

The press coverage of electronic computers in this period has been surveyed by Dianne Martin, who concluded 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".[78] This phenomenon was not restricted to journalistic accounts; 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".[79]

Computer developers themselves often viewed such characterizations as being 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".[80] 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.[81]

Such arguments were often supported by an appeal to a principle first stated by Babbage's collaborator Ada Lovelace: in Hartree's words, this claimed that "[t]hese machines can only do precisely what they are instructed to do by the operators who set them up".[82] Along with the related question of whether machines could think, this principle generated a substantial public discussion in the following years.

In Sect. 6.2, it was shown that the cybernetic conception of the computer, which was explicitly drawn upon by von Neumann and Turing, depended on the belief

[76] Archibald (1946).

[77] Boring (1946).

[78] Martin (1993), p. 130.

[79] Berkeley (1949).

[80] Hartree (1946a).

[81] Martin (1993), p. 129.

[82] Hartree (1946a).

that, considered in the abstract as information processing machines, a strong identification could be made between the brain and the electronic computer. The fact that computers were described 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 the ASCC and ENIAC, and not the proposals contained in the *Draft Report*, which were described as revolutionary. It was several years until the first machines based on the stored-program design became operational, and even longer until they were widely available. At the point at which they entered public discourse, then, the new automatic computers were not represented or understood as logic machines. However, the way in which they were described reflected the dual heritage that von Neumann had made explicit: they were scientific devices for carrying rapid and autonomous calculations, which could also be seen as models or analogues of the brain.

6.8 Universal Machines

Within a few years this situation had changed somewhat, however, and it became common, at least in technical circles, to make an explicit link between electronic computers and Turing's universal machine concept. Before examining how this came about, it is interesting to consider the notion of universality in more detail.

In Turing's 1936 paper, the word 'universal' is applied to a specific machine $\mathcal{U}$ which is able to simulate the behaviour of any other machine, given a suitable representation of the table of the machine to be simulated. $\mathcal{U}$ 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 to perform only the basic operations required by one particular computation, in the same way that the control circuits of the ENIAC were rewired for each different 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 $\mathcal{U}$.

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 equally well to machines such as Zuse's Z3 or the ASCC, and leads to the conclusion that these machines can also be described as universal, despite the fact that they read their programs from external storage devices.

In the examples above, the machine being described as universal belongs to the same class of machines as those being simulated: $\mathcal{U}$ 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.[83] If this is to be possible, 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; this is discussed further in Chap. 8.

This suggests another sense in which a machine can be described as universal: we might ask whether a particular machine is capable of simulating the behaviour of any other machine whatsoever. The repertoire of possible machines is unlimited, of course, so this cannot be established by giving details of particular simulations. Instead, it is shown that a 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. Such a machine can be said to be *Turing-complete*. 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?' ".[84] 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.[85]

Turing evidently imparted this understanding to his close collaborators. In 1946, a semi-popular account of the ACE made an explicit link between the construction of automatic computing machines with *On Computable Numbers*:

> Although this Harvard machine [the ASCC] is an independent and original development, the possibility of the construction of such machines, and, indeed, more elaborate ones, had

[83] Rope (2007).

[84] Agar (2003), p. 7.

[85] Turing (1948).

> 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.[86]

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 the year of 1947/1948, 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",[87] 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.[88]

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 and focused instead on the influence he expected the EDSAC to have on scientific research.[89]

A more detailed presentation can be found in the book *Calculating Instruments and Machines* published by Douglas Hartree in 1949.[90] In 1946 Hartree had visited the USA and made practical use of the ENIAC.[91] 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 described 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

[86] Department of Scientific and Industrial Research (1946).

[87] Huskey (1948), p. 976.

[88] Newman (1949), pp. 271–272.

[89] Wilkes (1949).

[90] Hartree (1949).

[91] Hartree (1946b).

then motivated the particular design of the computer by referring to the comparison that Turing had made with the procedures carried out by human computers. 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".[92]

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".[93] 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 ACE 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".[94] This rather contorted sentence suggests on the one hand that Hartree was aware of Turing's description 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 straightforward.

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".[95] Shannon did not define what he meant 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".[96]

It is not easy to extract a single consistent view on universality from Shannon's paper. On the one hand, the computer was described as 'general purpose' and the

[92] Hartree (1949), p. 97.

[93] Hartree (1949), p. 94.

[94] Hartree (1949), p. 97.

[95] Shannon (1950), p. 256.

[96] Shannon (1950), p. 265.

paper demonstrated 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",[97] and he made it clear that his preference would be to develop special-purpose machines. The paper made no reference to Turing's work, and it seems clear that Shannon viewed the machines being designed in 1948 as numerical calculators rather than as universal machines.

In September 1950, both Shannon and Turing were present at 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.[98]

During an informal discussion at this same symposium, Turing distinguished the construction of special-purpose machines for playing chess from the task of programming a computer to perform the same task,[99] but in 1950 his most significant contribution to the debate 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.[100]

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

[97] Shannon (1950), p. 265.

[98] Cherry (1950).

[99] Turing (1950b).

[100] Turing (1950a).

> which would be involved if they were to simulate human behaviour. This is not, however, a matter of fundamental importance.[101]

Although it is clear that Wilkes has here been influenced by Turing, he appears, like Shannon, to be envisaging classes of specialized machines for different tasks, while simultaneously recognizing the universality of particular machines within each class. The required specialization is found 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".[102] 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'.[103]

In the same year, Tony Oettinger spent a year with Wilkes in Cambridge working on programs which simulated learning. One of these simulated a machine with the ability 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".[104]

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".[105] Shannon, however, retained an interest in special-purpose machines and developed a physical machine to solve simple mazes rather than writing a program with the equivalent capability. 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",[106] but large parts of the paper are devoted to a consideration of 'machines' built for various

[101] Wilkes (1951b).

[102] Wilkes (1951a), p. 88.

[103] Prinz (1952), p. 261.

[104] Oettinger (1952), p. 1243.

[105] Wilkes (1953a), p. 1232.

[106] Shannon (1953), p. 1236.

purposes, not programmes. This preference for special-purpose machines has been noted to be a feature of the cybernetics community.[107]

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 same connection or think it important. Following 1950, however, Turing's paper was widely cited, and his characterization accepted and put into circulation.

6.9 General-Purpose Machines

The final 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.[108] Cybernetics suggested a wider range of applications: Wiener had originally been inspired by the problems posed by automated support for antiaircraft 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.[109] 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.

[107] Pickering (2002).

[108] Redmond and Smith (1980).

[109] Turing (1947).

6.10 Conclusions

The focus of this chapter has been on a particular episode in the development of the computer, namely the articulation of the stored-program principle in 1945. This has been given great prominence by historians of computing, and the fact of von Neumann's involvement 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, 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 varied 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 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 7
Machine Code Programming and Logic

The task of programming an automatic calculator or computer, usually referred to as 'coding', was understood to be that of specifying the sequence of operations that the machine would carry out in the course of a computation. The available operations were defined in a machine's 'order code', a list of the basic instructions out of which programs could be constructed. On the early relay machines, the operations were coded as a list of instructions and given to the machine through a medium such as punched cards or tape; for the most part operations were executed in the order in which the instructions were presented to the machine.

In stored-program machines, however, the instructions were held in internal memory. This approach was motivated by the need to make instructions available at high speed, but it also allowed two new coding techniques to be introduced. The designs proposed for the storage units of automatic computers enabled data to be retrieved from any storage location without too great a delay. If instructions were held in the same storage, therefore, they could also be accessed by the control unit in any order, making it feasible for programs to execute them in an order different from the sequence in which they were stored. Furthermore, programs could modify data in the store, a feature that made possible the writing of programs that could modify their own instructions to a potentially unlimited extent.

Many different order codes were possible for stored-program machines, however, and it would only be through practical experience that the features of a successful code could be identified, as von Neumann and his colleagues 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.[1]

Turing outlined his approach to these issues in the ACE report:

[1] Burks et al. (1946), p. 100.

M. Priestley, *A Science of Operations*, History of Computing,
DOI 10.1007/978-1-84882-555-0_7, © Springer-Verlag London Limited 2011

> A simple form of logical control would be a list of operations to be carried out in the order in which they are given. Such a scheme … lacks flexibility. We wish to be able to arrange that the sequence of orders can divide at various points, continuing in different ways according to the outcome of the calculations to date. We also wish to be able to arrange for the splitting up of operations into subsidiary operations.[2]

He went on to argue that the features of the stored-program design made it possible to meet these requirements, as they gave the machine the "possibility of constructing its own orders" by "taking a particular minor cycle out of storage and treating it as an order to be carried out", and made it easy to change the order in which instructions were obeyed. This, he claimed, would be sufficient.

By 1950, broad agreement had been reached about the features that a successful and usable code should provide, and in 1951 Maurice Wilkes and his colleagues in Cambridge published a book describing the programming system that they had devised for the EDSAC. Despite the machine-specific nature of the code presented, the authors believed that the ideas contained in the book were generally applicable, pointing out that "for the main part [the methods] may readily be translated into other order codes".[3] The book was widely read, and contributed greatly to the spread of this model of programming.

The first half of this chapter describes the key features of this model, with an emphasis on the process of experimentation and consideration of alternatives that preceded the acceptance of the 'standard model'. The second half of the chapter considers some more logical and philosophical aspects of this programming style.

7.1 Sequencing of Operations

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. For 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.[4] This requirement strongly influenced the design of the first large-scale, automatic digital calculators. Aiken and Grace Hopper wrote of the completed ASCC 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.[5]

[2] Turing (1946), p. 43.

[3] Wilkes et al. (1951), preface.

[4] Aiken (1937).

[5] 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.[6]

Specifying the required sequence of operations was therefore a basic aspect of coding problems for these machines. However, most of the calculating machines and installations designed before 1945 had units which were capable of operating in parallel. This meant that they could carry out more than one operation at the same time, a feature which 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, the ASCC 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 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 the ASCC possessed a number of specialized units for carrying out other operations, such as a unit to perform multiplications and divisions. 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".[7]

Once started, this dedicated unit carried out a multiplication independently of the main sequence mechanism. In general, a multiplication 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 that specified 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 the ASCC could not be read as a straightforward specification of the sequence of operations carried out by the machine, and the parallelism in its architecture was reflected to some extent in the way it was coded. Although it increased the efficiency of machine usage, however, the technique of interposing instructions created problems in writing and maintaining programs, as Richard Bloch, an early ASCC programmer, noted:

---

[6] Burks (1947), p. 756.

[7] Aiken and Hopper (1946), p. 449.

> 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.[8]

Hardware parallelism was also a feature of the ENIAC. The machine was built around 20 accumulators which, like the ASCC's storage registers, both stored a number and carried out simple operations on it. It also possessed separate units for carrying out operations such as 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 the transfer of information or the execution of an operation on a separate unit were enabled by connecting units together physically 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.[9]

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.[10]

As Eckert went on to note, the relay machine developed by the Bell Telephone Laboratories was programmed in a purely sequential manner.[11] In the *Draft Report*, a sequential, step-by-step approach emerged as a fundamental design principle; this was initially 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.[12]

---

[8] Bloch (1999), p. 87.

[9] Goldstine and Goldstine (1946). See Chap. 5 for more discussion.

[10] Eckert (1946), p. 114.

[11] Alt (1948a).

[12] von Neumann (1945), Sects. 5.6–5.7.

This principle was extended to all levels of the design, however. 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.

It has been suggested that the emphasis on sequential processing was principally motivated by the desire to increase reliability by using as little physical equipment as possible.[13] 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.[14] 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.[15]

In the context of hardware, Paul Ceruzzi has discussed the transition from "an architecture that processed data in parallel to one that processed data serially".[16] As we have seen, a similar transition took place in programming, but it took some time for the notion of sequencing to reach a stable form. The idea that did emerge, that a program should be thought of as a sequence of instructions each of which specifies a single operation which must be completed before the next instruction is executed, was influenced as much by experience in writing programs for the new machines as by considerations of their architecture. 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. Its adoption therefore reflected a decision to restrict the space of possible hardware designs in favour of simplifying the programming task.

Two main approaches were adopted to the question of specifying the sequence of instructions in a program. On machines which read instructions from an external tape, such as the ASCC, the sequence was simply defined by the order in which the instructions appeared on the tape. The stored-program machine described in the *Draft Report* adopted this approach by storing instructions in contiguous locations

[13] Marcus and Akera (1996), p. 23.

[14] Burks et al. (1946).

[15] Mauchly (1947), pp. 204–205.

[16] Ceruzzi (1997).

in memory: when an operation was complete, the next instruction was automatically read from the following memory location.

However, this introduced an inefficiency when the instructions were stored in the delay lines which were commonly used in the early machines. In this form of storage, data circulated continuously through long tanks of mercury, and were only accessible when they reached the end of the tank. There was no guarantee, of course, that the next instruction would be available when it was required, and this meant that computations could be significantly slowed down because of the time spent waiting for the necessary instructions to emerge.

To avoid these delays, an alternative approach, known as 'optimum coding' was sometimes adopted. In this approach, every instruction contained 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. A few machines adopted optimum coding,[17] but as random access memory became available the practical advantages of optimum coding became less crucial and the sequential placement of program instructions in memory became the norm.

7.2 Transfer 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.[18]

This model was shared by all the automatic calculators: a complete program for a computation was constructed from a number of distinct sequences of instructions. Normally, one particular sequence defined the structure of the entire computation: what was needed, therefore, was a method for executing the other sequences when necessary and causing a given sequence to be repeated as often as required. These requirements were met in different ways by different machines.

For example, the ASCC's sequence control unit read instructions from paper tape. Computations were normally split across multiple tapes, each containing a particular instruction sequence, but no mechanism was provided for automatically transferring from one tape to another. Instead, the programmer had to leave detailed instructions for the operators specifying which tapes should be loaded on to the machine and when, among other pieces of information. Aiken and Hopper described

[17] Bloch et al. (1948).

[18] Mauchly (1947), p. 204.

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".[19] The repetition of the instructions on the control tape in this process was achieved by making the tape "endless": in practice this was done by simply gluing the ends of the tape together, so forming a loop.[20]

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", containing 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, which allowed for the definition of highly complex program structures.[21] By 1946, Aiken recognized the need to supply the ASCC 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.[22]

In the early machines, then, the logical structure of a program was expressed physically by referring to some aspect of the machines' setup. In the subsequence mechanisms employed by the ASCC 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, however, a complete program was stored in a single, uniform memory and 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.[23]

[19] Aiken and Hopper (1946), p. 528.

[20] Aiken (1946), p. 156.

[21] Goldstine and Goldstine (1946).

[22] Bloch (1947).

[23] Eckert (1946), p. 116.

The EDVAC included in its code an order ζ which connected the control organ to a specified memory location where the next order could be found.[24] This location could contain a previously executed instruction, or mark the beginning of a distinct sequence, so this single transfer order supported the two operations of executing a new subsequence and repeating 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.)

Transfer orders of this type allowed the flow of control within a program to be specified without making reference to particular features of the machine on which the program was running. The stored-program approach therefore made possible a more abstract understanding of program structure, in which an entire program, including all the necessary subsequences of operations, was 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 was to run on. The generalized transfer instruction therefore marks a significant step towards a complete logic of control, or in other words a notation which was independent of any particular hardware configuration and yet capable of expressing both the elementary operations required and also the order in which they should be performed.

Two other points can be made about transfer instructions. Firstly, although they appeared in conjunction with the stored-program design, this was not a necessary condition of their use: 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 transfer instructions allowed more flexibility than expressing program structure through hardware, their use could also obscure that structure. On a machine with multiple tape readers, the distinction between the control sequence and subsequences of instructions was clear: it is harder to perceive this structure when both are merged into a single sequence of instructions and only implicitly distinguished by transfer orders.

7.3 Condition Testing

Simple transfer instructions were soon recognized to be insufficiently expressive to allow the easy coding of many common computations. In many cases, the future course of a computation depends on the results obtained so far: a frequently cited situation was the coding of iterative procedures, where it was necessary to repeat 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.

[24] von Neumann (1945), Sect. 15.

A variety of approaches were adopted to provide this capability. On the ASCC, for example, counter 72 was known as the 'automatic check counter', and an order 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 the 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.[25]

The original design of the ENIAC did not include any mechanism for testing the values currently held in the accumulators. This capability was provided 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 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.[26]

The situations in which it was thought necessary to interrupt the normal sequence of instructions were very closely linked to the application area of the machines, numerical computation. As von Neumann put it in the *Draft Report*:

> 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.[27]

The code presented in the *Draft Report* only provided this capability indirectly, however, relying on the fact that in the stored-program design instructions and their addresses are available to examined and perhaps modified, just as numeric data can be. Rather than providing a single conditional jump operation, the code defined a basic operation, s, of the arithmetic unit which could "sense the order relation between numbers". It had four arguments, x, y, u and v: if $x \geq y$ the operation would have as result u, and if $x < y$ the result would be v.

Von Neumann went on to claim 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". At the point where such a choice needed to be made, the machine would have to choose which of two instructions to carry out next. The addresses of these two instructions could be provided to s as the arguments u and v: once the relation between x and y had been tested, one of these addresses would be delivered as the output of s. This address could then 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.

[25] Aiken and Hopper (1946).

[26] Marcus and Akera (1996).

[27] von Neumann (1945), Sect. 11.3.

In the ACE report, Turing proposed a similar indirect approach. He did not put forward a specific operation to choose between two numbers, however, suggesting instead that the address of the next instruction could be calculated using ordinary arithmetical instructions before being copied into a transfer instruction. He gave the following example where the program should next carry out instruction either 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.[28]

By the middle of 1946, however, codes were being proposed that did not require programmers to construct alternative transfer instructions explicitly. Instead, these codes contained a single instruction to carry out a conditional transfer. For example, Eckert and Mauchly described an order code, known as 'Code A', which included a pair of 'comparison' orders. In Code A, each order could contain the addresses of up to three storage locations, *alpha*, *beta* and *gamma*. The effect of the order *c* was described as follows:

> If the number stored in register alpha is greater than the number stored in register beta, shift the control to register gamma.[29]

The code used by von Neumann's group at Princeton had similar instructions which carried out a transfer depending on whether the sign of the number stored in the accumulator was positive or negative.[30] Conditional transfer instructions of these or similar types were found in all subsequent codes.

It can seem striking in retrospect that conditional patterns of control such as "transfer to instruction 53 if the value of the number at address 256 is negative" were not immediately mapped onto a single instruction in an order code. Of the early machines, only the Bell Labs machine provided such an instruction,[31] and it has even been suggested 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".[32]

Rather than seeing this as a problem requiring explanation, however, it can be taken as evidence of the difficulty that can accompany innovations that later come to seem self-evident, and of the extended process of exploration and negotiation that often accompanies conceptual innovation. The experience of Paul Verzuh, who attended the Moore School course and took notes on the lectures, testifies to the

[28] Turing (1946), p. 35.

[29] Eckert (1946), p. 122.

[30] Burks et al. (1946).

[31] Alt (1948a), pp. 72–73.

[32] Carpenter and Doran (1977), p. 271.

potential difficulty of assimilating new ideas: according to the editors of the lectures, these notes indicate that "he had grave misunderstandings of the transfer of control orders" in Code A, as presented by Eckert.[33] Von Neumann and Turing, on the other hand, seem rather to have been exploring the possibilities provided by the technique of instruction modification: to describe the lack of conditional transfer instructions in their original codes as a stumbling block testifies to a rather narrow view of what was going on in this process of innovation.

7.4 Instruction Modification

In common with other fundamental programming concepts, the idea of instruction modification underwent considerable evolution before reaching a definitive form. In the *Draft Report*, as a result of a rather complex chain of design decisions, the order code provided 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.[34]

No theoretical principle was invoked to justify this decision, however. 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.[35]

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 that were 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".[36] 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 into memory, the way in which this transfer was carried out depended on whether the

[33] Campbell-Kelly and Williams (1985), p. 108.

[34] von Neumann (1945), Sect. 12.2.

[35] von Neumann (1945), Sect. 15.1.

[36] von Neumann (1945), Sect. 15.3.

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.[37] This facility for 'address modification' was the only method that the code provided to modify the orders making up a program, and was used among other things to provide conditional jumps, as discussed above.

In his description of the ACE, Turing defined its memory in essentially the same way as von Neumann, as "minor cycles" of 32 binary digits, 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". 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".[38]

However, in the report Turing assumed that programs would have an unrestricted ability to modify their own orders, using exactly the same basic operations as were provided for working with numeric data. Turing's approach therefore 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.[39]

Calvin Mooers made this point even more bluntly, stating that the modification of orders should be "a simple arithmetic operation between numbers and orders".[40] The 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 a 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,[41] Mooers used straightforward numerical addition, thus simplifying his code slightly.

Von Neumann and his collaborators also came to adopt a more flexible approach than that put forward in the *Draft Report*. In the computer built for the Institute of Advanced Studies, they distinguished "two different forms of memory: storage

[37] von Neumann (1945), Sect. 15.6.

[38] Turing (1946), pp. 24, 25.

[39] Mauchly (1946), p. 455.

[40] Mooers (1946), p. 470.

[41] Eckert (1946), p. 122.

of numbers and storage of orders",[42] 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 which would 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 in general, control could "modify any part of the coded sequence as it goes along".[43]

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.[44] Of particular importance was the situation where a loop was to be written to carry out a set of operations on data stored in a number of consecutive memory locations. This could easily be achieved by adding instructions to the loop to increment the relevant address fields at each iteration.

Some machines enabled this process to be partially automated. The increments that were to be applied to instructions were stored in special registers, often known as *B tubes* or *B lines*, following the terminology used on the Manchester computer which first introduced them.[45] Other instructions contained a field which was used to specify one of the available B tubes, and the contents of the selected tube would be added to the instruction before it was executed. Initially, an entire instruction could be modified in this way, but later implementations restricted this so that only the address field could be changed. This idea came into widespread use, B tubes becoming more commonly known as *index registers*.

A striking example of what could be achieved when instruction modification was employed without any restriction was provided by the so-called "Initial Orders" written by David Wheeler for the EDSAC. These orders read a symbolic form of a program from paper tape, and loaded the corresponding program in memory when the EDSAC was started up.[46] The initial orders included such techniques as the use of "ambiguous words", which at different times were treated as numbers or instructions, and they repeatedly constructed 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.

[42] Burks et al. (1946), p. 98.

[43] Goldstine and von Neumann (1947), p. 153.

[44] See for example Bloch et al. (1948), p. 293, and Bowden (1953), p. 29.

[45] Williams (1951), p. 176.

[46] Wheeler (1950).

7.5 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 ASCC 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 described above, subroutines, as reusable sequences of instructions, were held as physically distinct program tapes on the ASCC 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, the instructions making up a subroutine were located 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*, but 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.[47]

This approach requires the ability to transfer control to the start of a subsidiary operation and to return to the main operation on completion of the subsidiary. 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.[48]

The notes of the return addresses were to be "buried" in storage, and a record kept of the location 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 on different occasions of use a subroutine would be located at different places in memory. 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

[47] Turing (1946), p. 34.

[48] Turing (1946), p. 35.

a fixed address in the transfer instruction with the fact that address will vary in different programs, depending on where the subroutine is located in memory.

Turing's solution to this problem was to split the process of program assembly into two stages. Instructions were to be written on cards in a "popular" form that was more or less readable by humans, 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. To construct a program, all the cards required would be brought together 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",[49] but did not propose to do this in the first instance.

Goldstine and von Neumann considered the use of subroutines in more detail in a report circulated in 1948.[50] 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 the one already described 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. In contrast with Turing's approach, 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 could be handled by a preparatory routine, the problems presented by changing parameters and return locations can only be dealt with when a program is running. Goldstine and von Neumann did not describe in detail a method for doing this, but it is clear that they assumed that some form of instruction modification while the program is running would suffice.

It is worth noting that this approach is less flexible than Turing's proposal to store return addresses in a separate area of memory. Whereas Turing's idea would permit recursive calls to subroutines, this is impossible with an approach which physically modifies the return instruction at the end of a subroutine. The difference

[49] Turing (1946), p. 38.

[50] Goldstine and von Neumann (1948).

between the two proposals is perhaps accounted for by differing philosophies 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 seem to have had in mind a hierarchical structure in which the main routine of a program would call subroutines, but where references between subroutines would be rare.

In 1949, once the EDSAC was operational, a detailed scheme for handling all these aspects of subroutine usage 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.[51] These orders did not modify a complete program in the style of Goldstine and von Neumann's preparatory routine, however, but instead interpreted a coded version of the program read from the tape and constructed the complete program in memory. Wheeler also invented techniques for modifying the return addresses in subroutines and for enabling parameterized data to be used in subroutines. These were later described in the textbook issued by the Cambridge group, and became highly influential.[52]

The adaptation of the familiar idea of a subroutine for use on stored-program computers, then, can be characterized by two main features. Firstly, it turned out that subroutines could not be effectively used unless they were processed in some way prior to execution; during this stage, the complete program including correctly linked subroutines was constructed in some way. Secondly, in the complete program thus constructed, the capabilities provided by transfer instructions and instruction modification turned out to be 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.

7.6 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.

[51] Wheeler (1950).

[52] Wilkes et al. (1951).

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. Also, conditional transfer instructions were provided which allowed the sequence or orders executed to vary according to the current state of the computation.

Finally, programs could modify instructions in the course of a computation. There were a number of situations in which this was known to be necessary, but instead of providing special instructions for these situations, most codes allowed instructions to be treated as numeric data and allowed unrestricted manipulations to be performed on them.

Various extensions to this standard model had been proposed. For example, at the Moore School course Mauchly had presented a code which included "index counting instructions" to make the control of loops easier,[53] and Mooers described a change to the von Neumann design to include a device called a "sentinel" and a code which included so-called "stop order tags" to facilitate the detection of boundary conditions in certain applications.[54] None of these innovations was widely, if at all, adopted; for example, the very influential EDSAC code was essentially that of the standard model outlined above.

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.[55] 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".[56] 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.

As described above, all the 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".[57] This situation can easily be coded using a conditional transfer, but this same instruction can be used

[53] Campbell-Kelly and Williams (1985), p. 452.

[54] Mooers (1946).

[55] Turing (1946), p. 35.

[56] Turing (1947), p. 117.

[57] Turing (1947), p. 118.

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, but this meant that it was not easy to grasp the structure and design of a program simply by inspecting the code.

7.7 Machine Code and Logic

Both Turing and von Neumann 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.[58]

and a similar point was made by Goldstine and von Neumann:

> 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.[59]

However, these comments do not make it clear exactly what the intended force of this comparison was. As described in Chap. 4, one of the achievements of logic had been to demonstrate how important aspects of mathematical language could be captured by formal, or 'mechanical' rules. One 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 were 'mechanical'. Machine code programs and the instructions they contain bear little resemblance to the sentences of propositional and predicate logic, however, so it is worthwhile exploring in a bit more detail what was understood by the analogy.

The terms 'logic' and 'logical' were used in connection with computers in ways that did not imply a connection with mathematical logic. For example, a distinction was often drawn between the 'logical' and the 'physical' design of a machine, the difference being that the logical design made no reference to specific circuits or electronic devices.[60] 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 comment on Babbage's mechanical notation:

> 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.[61]

[58] Turing (1947), p. 122.

[59] Goldstine and von Neumann (1947), p. 154.

[60] Bloch et al. (1948).

[61] Bowden (1953), p. 17.

In this sense, then, 'logic' relates to the abstract structure of a device, and in particular to the static and dynamic relationships between its parts. This usage is closely related to that of von Neumann and others in discussing issues of coding and control. The *Draft Report* defined "[t]he logical control of the device" to be "the proper sequencing of its operations",[62] 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".[63] It was common to make the distinction between a computer's 'arithmetical' or 'mathematical' operations and its 'logical' operations explicit: for example, Goldstine and von Neumann described the "arithmetical operations and transfers of numbers" as the "properly mathematical (as distinguished from the logical) operations of the machine",[64] and Edmund Berkeley included among the set of 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.[65]

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 Rudolf Carnap's *Logical Syntax of Language*,[66] 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'.

A possible explanation for this is based on the observation that formalization had led to a more abstract view being taken of logic itself. Rather than thinking that rules of inference say something important about the notion of truth, it was now possible to think of them simply as defining a formal relationship, that of entailment, between sentences in a formal language. In a similar way, the 'logical' aspects of machine code could be thought of as describing 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",[67] 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 Friedrich Bauer, who states that the "*Plankalkül* is an instrument for reasoning

[62] von Neumann (1945), p. 2.

[63] Goldstine and von Neumann (1947), p. 113.

[64] Goldstine and von Neumann (1947), p. 115.

[65] Berkeley (1950).

[66] Carnap (1937).

[67] Giloi (1997), p. 18.

about programs—quite a modern point of view".[68] This comment does not seem to be valid if it is interpreted as meaning that Zuse was interested in proving or validating the functional properties of his programs: unlike Goldstine and von Neumann or Turing,[69] 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—one of his example programs was to check the well-formedness of a formula in propositional logic, for example—but his programming notation does not seem to have been specifically related to more traditional logical notions of proof and reasoning.

Once the analogy between logic and coding had been made, it became possible for researchers to consider what other features of logic could be fruitfully applied in the new area. Following Carnap and Morris, it had become common to structure the study of formal languages using the metalinguistic categories of syntax, semantics and, to a lesser extent, pragmatics. The following sections describe the ways in which these notions began to be applied to machine codes.

7.8 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; for example, the EDSAC team referred to it as "the device of expressing the orders in a numerical code".[70] However, this does not make explicit the fact that numbers also had to be coded before they could be stored, and in fact a variety of coding schemes had been used, even on the early relay computers.[71] 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".[72]

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.

[68] Bauer (2000), p. 278.

[69] See Turing (1949) for an early example of program proving.

[70] Wilkes et al. (1951), p. 3.

[71] Booth (1949).

[72] Huskey (1951).

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 sometimes presented in a variety of symbolic representations. For example, in the *Draft Report* von Neumann distinguished between the "short symbols" that were 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.[73] 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.[74] 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.[75] The alternative forms of code represented only its functional details, 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 individual instructions. The sequence of instructions making up a program was usually shown by listing actual or illustrative memory locations and showing the instruction stored in 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 the effect of a program 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.

[73] von Neumann (1945), Sect. 15.6.

[74] Turing (1946), Sect. 13.

[75] Wheeler (1950).

There appeared to be no clear advantage in terms of either execution time or overall code size 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.[76] 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.

An interesting attempt to apply logical syntax to computing at this time was made by George Patterson, who made explicit use of Carnap's work in sketching out a general theory of "syntactical machines".[77] He described a class of machines which he termed "linguistic transducers", which accepted input data and transformed them into output data. By viewing these data as symbolic expressions, Patterson hoped to use Carnap's approach to develop a logical theory of such machines. Analogue computers were not amenable to 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 in the application of the ideas of logical syntax to a type of language very different from conventional logic. For Carnap, the transformation rules were meant to capture aspects of the consequence relation between sentences. Patterson did not explicitly define a class of sentences, however, nor did he discuss the relationships between sentences. Instead, he was concerned with a class of "numerical expressions", or numerals in a specified base, and he gave a recursive definition of a "quasi successor" relation between numerical expressions. This definition was subsequently referred to as a "transformation rule", a usage which clearly marked 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 was 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.

[76] Elgot (1954).

[77] Patterson (1949).

7.9 Flow Diagrams and Program Semantics

A very natural account semantic account of programs takes as its starting point the idea that the meaning of a program text, or sequence of instructions, is in some way connected to the computation that it expresses or, in more concrete terms, the different sequences of operations that a computer might carry out while executing the program. In the course of their exploration of the possibilities of programming stored-program machines, Goldstine and von Neumann came up with a sophisticated account of this notion of semantics, together with a flow diagram notation which expressed such meanings and could be used as a way to develop programs.

They began by commenting on the relationship between a list of instructions and the sequence of operations performed when a program was executed. In the case of machines like the ASCC, where operations were by and large carried out in the same order as the instructions were written on the tape, this relationship was rather straightforward. Techniques such as making endless, looping instruction tapes or splitting a program among a number of different tapes did not seem to complicate this basic picture to any significant degree.

The situation with stored-program machines was very different, however, as a direct result of the two features that had been identified as typical of these new order codes: transfers of control meant that the sequence of operations executed could differ in arbitrary ways from the sequence of instructions, and instruction modification meant that the sequence of instructions itself could be expected to vary as the computation proceeded, making the very notion of a fixed program text rather problematic.

These differences meant that a semantic account of programs was going to differ in some ways from the account given for predicate logic. A semantic account of logic is typically presented as a mapping from a set of stable, syntactic expressions to some domain of meanings. The possibility of instruction modification, however, means that the situation with programs is more complex. If the execution of a program can cause the program text itself to change, then the operations subsequently carried out, and hence the program's meaning, 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]".[78]

Secondly, semantics for logic are typically compositional, in the sense that 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.

[78] Goldstine and von Neumann (1947), p. 154.

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 notation 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".[79] 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, 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.

The proposed flow diagrams 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 could contain a lot of information about the properties of the data being manipulated by the program. The notation maintained a strict distinction between the mathematical expression of the problem being coded, described in terms of *variables*, and the actual data being manipulated, which were referred to by reference to *storage locations*. The operations that were needed, together with numerical values stored at each point of the program's execution, were described mathematically. A distinction was drawn between free and bound variables, and the use of this terminology was explicitly linked with that of "formal logics",[80] 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 flow diagrams, the distinction being purely contextual.

Flow diagrams could also contain *assertion boxes*, which stated properties that were expected to hold at various times during program execution. This formed a bridge between programming and traditional logical notations, as an assertion could be any logical formula making use of the program variables. These assertions were to 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

[79] Goldstine and von Neumann (1947), p. 84.

[80] Goldstine and von Neumann (1947), p. 91.

an interpretation of the variables. The use of assertions was also adopted by Turing as a method for checking correctness of programs,[81] but then faded from view until the mid-1960s. A further difference between programs and conventional logic can be noted at this point: 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 can therefore be viewed as being in part an attempt to assimilate machine code programming to the theory and practice of contemporary formal logic. Flow diagrams provided a formal representation of the semantics of programs, and with the use of assertions, first-order logic was built in to the notation and to the methodology of program construction based upon it. Explicit analogies were drawn with logic even in relatively minor details of terminology, such as the distinction between free and bound variables.

Arthur Burks later described the use of "bound variables" in loops in flowcharts as being related to the bounded quantifiers introduced by Gödel.[82] 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, clearly related to the existing use of variables 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 degree of 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.[83] However, the diagrams showed only operation boxes and alternative boxes, and the connections between them: no distinction was made between variables and storage locations, and assertions were not used. With the exception of Turing's paper in 1949, these more 'logical' aspects of the notation were not on the whole taken up.

[81] Turing (1949).

[82] Aspray and Burks (1987).

[83] Renwick (1949).

7.10 Programs as Metalinguistic Expressions

Many years later, Arthur Burks reflected on the relationship between formal logic and the work that Goldstine and von Neumann did on programming in the 1940s, and he concluded that "I think it likely that, in his programming work, von Neumann was guided by his knowledge of Gödel's work, at least intuitively". In particular, Burks saw the fact that data in the memory of a stored-program computer can be interpreted as either numbers or instructions as pointing to "an instance of the metalanguage versus object language distinction".[84]

As described in Chap. 4, 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. Burks's distinction arises more from the way in which these data are used, however, than the way they are stored. In simple cases, machine code programs manipulate numerical data only, and can be viewed straightforwardly as expressions of the object language. A program that modifies its own instructions, however, is modifying expressions of the object language, and so the program can be interpreted as belonging to a syntactic metalanguage. Codes that permit instruction modification can therefore be seen as being simultaneously object and metalanguage.

The possibility that a language could, by making use of arithmetization, express its own syntax 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.[85] 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, however, the meaning of a program is considered to be the operations it carries out, 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 involved in computer development in the late 1940s and early 1950s, instruction modification, and 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 the ASCC, 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.

Interestingly, von Neumann consistently adopted a conservative attitude towards instruction modification. This conservatism can be understood as a side-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

[84] Aspray and Burks (1987), pp. 384–385.

[85] Carnap (1937).

of syntax and semantics. Further evidence in support of this line of thought will emerge in subsequent chapters, which describe the emergence of a theory of programming languages that was explicitly modelled on metalogic accompanied by 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. Metalanguages in logic provide the ability to describe aspects of the syntax and the semantics of another language. Machine code programs are not declarative, however, consisting of orders rather than statements, and so they cannot simply describe syntax and semantics in the way of logical metalanguages. Programs can be written, however, which manipulate instructions and program code in various ways, and it would seem reasonable to view such programs as being metalinguistic.

Examples of such 'metaprograms' 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.[86] This approach was quickly recognized as providing great benefits in programming productivity: a number of systems went on to define 'interpretive 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 interpretive program into the machine's native code. This approach to coding became known as 'automatic programming', and throughout the 1950s many such codes were produced, gradually evolving into what became known as programming languages. These developments are discussed in detail in the next chapter.

7.11 Conclusions

Von Neumann's *Draft Report* is widely recognized as marking a turning point in the history of computer hardware. It represents a moment of closure, where a number of elements, individually present for the most part 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 much greater continuity between the way in which the ASCC and the EDSAC were programmed, than there is in the design of their hardware. The nearest analogue to the *Draft Report* in the field of programming is perhaps the textbook written by the EDSAC group.[87] Like the *Draft Report*, this brought together in a particularly clear form the principles on which contemporary work was based, and served as a model for much later work.

[86] Wheeler (1950).

[87] Wilkes et al. (1951).

Chapter 8
The Invention of Programming Languages

It quickly became apparent that the task of creating machine code programs was one that most humans would find challenging, and techniques for simplifying and automating parts of this process were soon developed. Symbolic abbreviations for operation codes were often employed and several programming systems, following the example of 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 programs, but still required programmers to define the sequence of basic operations making up a program. A further stage of automation was envisaged in which this latter 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.[1]

The first application of such 'programming schemes' was in connection with mathematical formulae. During the 1950s, a number of systems were developed which allowed programmers to write programs which contained formulae written in some approximation to standard mathematical notation; these formulae would then be automatically translated into a sequence of basic machine instructions to perform the calculation defined by the formula. The most ambitious and successful of these systems was Fortran, which first became available in 1957 for the IBM 704 computer.[2]

At the same time, it became apparent that certain coding patterns were used over and over again to control the sequencing of operations in a program. The clearest example of this was the coding of loops, which were often controlled by an index variable counting the number of times the loop had been executed. In much the same

[1] Gill (1959).

[2] IBM (1956).

M. Priestley, *A Science of Operations*, History of Computing,
DOI 10.1007/978-1-84882-555-0_8,

way as formula translation systems automated the process of writing instructions to perform calculations, systems gradually began to provide ways of automating the writing of instructions implementing control structures such as loops. As there was no existing notation for control structures, however, these developments were rather more tentative than those to do with the translation of mathematical formulae.

By the end of the decade, many automated programming systems existed, mostly designed for and implemented on a particular type of computer. A number of groups had discovered the benefits of sharing code and the necessity of running programs on more that one type of computer, but the existence of machine-specific programming notations made these tasks 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.[3]

This chapter examines these technical developments and the parallel evolution of theoretical accounts of programming notations. At the start of this period, these were understood relative to a machine, whether real or imaginary; by the end, they were increasingly 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. On the whole, natural languages inspired developments in notations intended for use in data processing applications, whereas formal logic was taken as a model for programming languages intended for mathematical and theoretical uses.

8.1 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, 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 transliteration of the external symbols used to write the program in a form accessible to humans into the corresponding internal codes, but the use of subroutines made the task more complex. In order to make free use of subroutines, it must be possible to place their instructions at different locations in the store in different programs. This means that any instructions in the subroutine that refer to specific storage locations or jump to another instruction within the subroutine may 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

[3] Naur et al. (1960).

the programmer would not have to decide where in the store each subroutine should be placed.

These and related techniques, such as a method for attaching labels known as 'floating addresses' to selected instructions, involved the manipulation of a program before it was run.[4] 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 what became known as '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.

The EDSAC subroutine library contained interpretive routines to make it easier to code programs which performed 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. 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.[5] 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 gave an example where the instructions in the interpreted code were so small that two orders could be placed in a single machine word.[6]

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. Perhaps the earliest such system to have been implemented was John Mauchly's 'Brief Code', which was developed for the BINAC computer but first ran on the UNIVAC in 1950, by which time it was known as 'Short Code'.[7]

Short Code evolved from a proposal made by 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".[8] Mauchly's argument in favour of

[4] Wilkes (1953b).

[5] Campbell-Kelly (1980), p. 29.

[6] Wilkes et al. (1951), pp. 162–164.

[7] Schmitt (1988). As described below, Short Code was principally a formula translation system.

[8] Mauchly (1949).

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 interpreted codes 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".[9] 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 the work of Grace Hopper, were being carried out on the UNIVAC. Motivated like Mauchly by a growing awareness of the cost of programming, Hopper hoped that "[t]he programmer may return to being a mathematician". This was to be achieved by what Hopper called "compiling routines", or programs "designed to select and arrange subroutines according to information supplied by the mathematician or by the computer".[10] A series of such routines were developed and ran on the UNIVAC from 1952 onwards.

From 1950 on, then, a number of different 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 *pseudocode*. 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.

8.2 The Semantics of Pseudocodes

Machine codes were, naturally, understood as being notations for expressing or specifying the behaviour of particular computing machines. It could not be assumed, however, that an instruction in a pseudocode corresponded to a single instruction in the underlying machine, although sometimes this was of course the case. This made it necessary to come to a more complex understanding of the meaning of pseudocode programs, and a number of different interpretations emerged as codes became more common and more complex.

[9] Wilkes (1952).

[10] Hopper (1952), pp. 244, 248.

Extending Machine Code Some pseudocodes were syntactically very similar to the underlying machine code. This was particularly true of the interpretive orders that were developed for the EDSAC, of which Wilkes wrote that "[n]o example of a programme of interpretive orders need be given since it would look just like an ordinary programme".[11] In cases like this, it was possible to treat the pseudocode as if it was an extension of the original machine code: "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'".[12] Not only interpretive orders, but also subroutines could be thought of in this way, and Wilkes went on to state that "[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".[13]

If the syntactic relationship between machine code and pseudocode was very close, this could have implications for the ease with which pseudocode programs could be processed. Again with reference to the EDSAC, Wheeler wrote that "the [interpretive] sub-routine executes the 'orders' in the list in a similar fashion to the way that the machine obeys ordinary orders".[14]

Translation In cases where the pseudocode did not so closely resemble machine code, the notion of translation was often invoked to explain the relationship between the two. This was initially understood as a run-time process of interpretation; in his initial proposal for what developed into Short Code, Mauchly wrote that the computer on which the code ran "must be provided with routines for interpreting this special code, and for executing the indicated instructions".[15]

This alternative viewpoint was motivated by the fact that, unlike the EDSAC's interpretive orders, Short Code was understood to be a new linguistic formalism, one which was distinct from the underlying machine code. One consequence of this recognition was that the informal idea of interpretation developed into a more formal notation of translation, and became applied to a wider range of situations. In 1951, for example, 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".[16] In 1952, a group at Manchester described their work on developing a code for a new machine in precisely these terms, claiming that they were "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

[11] Wilkes (1952).

[12] Wilkes et al. (1951), p. 35.

[13] Wilkes (1952).

[14] Wheeler (1952).

[15] Mauchly (1949).

[16] Good (1951).

translates the new routine from the new code back into the code of the existing machine".[17]

Translation is usually thought of as a meaning-preserving relationship between expressions in distinct languages. Applying the translation metaphor to interpretive routines encouraged people to think of programming codes as languages in their own right: in the same year, Earl Isaac made this idea explicit, stating as a general principle that "[c]oding for digital computers is a process of translating from one language to another".[18]

Virtual Machines Yet another interpretation was available, however, according to which interpretive routines were understood to be extensions or modifications 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".[19] The change of perspective, from interpreting an extended code on a basic machine to programming on an extended machine, was made explicit by 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".[20]

This viewpoint gained some of its force from the desire or need to simulate on one machine the hardware of other, more powerful machines. For example, the floating-point interpretive routine designed by Brooker and Wheeler simulated in the EDSAC's memory both a floating-point accumulator and the B tubes available on the Manchester machine, and permitted recursive subroutine calls, even though these capabilities were not provided by the EDSAC's hardware.[21] 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".[22] 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".[23]

Compilers as well as interpretive routines were understood as creating synthetic machines. Some years later, Hopper characterized the compiling routines that she

[17] Bennett et al. (1952).

[18] Isaac (1952). 'Coding' is here used to refer the generation of machine code, not simply writing a program.

[19] Turing (1951), p. 192.

[20] Isaac (1952).

[21] Brooker and Wheeler (1953).

[22] Backus (1958), p. 234.

[23] Backus (1954), p. 4.

had developed by saying that "the compiler ... effectively converted the UNIVAC from a single-address, fixed-decimal computer into a three-address, floating-decimal computer".[24] The introduction of later pseudocodes 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 ...",[25] 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".[26]

Pseudocodes, then, came to be understood in the same way as machine codes, namely as being the instruction codes of particular computing machines. However, the machine corresponding to a given pseudocode would usually not have been built, but would be simulated on an existing machine. When a pseudocode program was run, the job of the interpreter or compiler writer was to ensure that the same results were produced that would have been obtained if the 'synthetic', or virtual, machine assumed by the writer of the pseudocode had been operational and the pseudocode 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.[27]

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.

It is worth noting, however, that the two interpretations of pseudocodes continued to coexist, as Gill noted at the end of the 1950s: "[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".[28] For codes that syntactically resembled machine code, the virtual machine interpretation continued to seem very natural;

[24] Hopper (1959), p. 167.

[25] Laning and Zierler (1954), p. 1.

[26] Robinson (1960), p. 115.

[27] Booth (1960), p. 1.

[28] Gill (1959), p. 111.

as more sophisticated notations evolved, however, the idea of translation was more frequently invoked.

Instruction Modification If the instructions of an interpreted pseudocode were held in a computer's memory, they were available for modification during program execution in the same way as machine code instructions. Some, but not all, systems supported the possibility of modifying pseudocode instructions. The Speedcoding system, for example, provided automatic address modification by means of three index registers, one for each of the three addresses appearing in an instruction. Each arithmetic operation in a Speedcoding program could specify which, if any, of the addresses should be modified by the contents of the corresponding index register before execution.

The situation was different in the case of compiled codes, however, where the pseudocode instructions were translated into machine code and so were no longer available for modification when the program was running. Compiled pseudocodes therefore had to provide alternative means to support the common purposes to which techniques like address modification were put.

A good example of this is provided by the PACT I system, which was developed in the mid-1950s for the IBM 701. Under the heading of "Automatic Loop Writing", Charles Baker wrote that:

> An important feature of a stored program computer lies in its ability to modify its own instructions. To take advantage of this, two logical instructions, "SET" and "TEST", are provided for instructing the compiler to write a loop.[29]

A PACT I loop to add together the values in two vectors is shown in Table 8.1. The first instruction sets the value of the subscript I to 1. This subscript is then used in the following three instructions, which add together the values in A_I and B_I and store the result in C_I. Finally, the TEST operation increments the value of I; if the resulting value is not greater than that given in the S_2 field, control returns to the instruction following the SET instruction.

This example illustrates a significant innovation, the *definition* of a computational structure, a loop, by specific syntactic features, the SET and TEST instructions. In machine code, by contrast, loops were *implemented* in terms of the jump, a simpler mechanism. It was suggested that the code in Table 8.1 could be treated as a whole and read as: "Take A sub I, add B sub I, and set the results equal to C sub I. Do this for $I = 1, 2, \ldots, 10$".

In some respects, then, the use of pseudocodes restricted the set of capabilities available to the programmer. In general, not every operation that could be carried out in machine code could be programmed in pseudocode. This restriction was felt to be acceptable if balanced by an increase in programming productivity, at least for the range of problems that the pseudocode was designed to address. To compensate for this, pseudocodes began to develop a greater linguistic richness by including features providing automatic coding of common programming patterns.

[29] Baker (1956), p. 275. Baker is referring here to the ability of a computer to modify machine code, not pseudocode, instructions, of course. The SET and TEST instructions belonged to the pseudocode and were not themselves modified.

Table 8.1 A loop to add vectors in PACT I

Step	Op.	Factor	S_1	S_2
0	SET		I	1
1		A	I	
2	+	B	I	
3	EQ	C	I	
4	TEST		I	10

8.3 Formula Translation

In some ways, programming in pseudocode was a similar experience to machine code programming. Problems still had to be broken down into steps which were small enough to be expressed as individual instructions in the code being used, whether or not it was that of a real machine. Low-level coding of this type was sometimes seen as a routine, but tricky and rather unrewarding task; as computers became more widely used there was an increasing demand for programmers, giving rise to some anxiety as to whether this demand could easily be met. One strategy that was adopted to address this situation was to attempt to make programming more interesting and accessible by developing programming notations that were directly 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 formulae. 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 formulae can be interpreted as expressing algorithms, specifying what arithmetical operations have to be carried out and in what order. For example, the simple formula

$$x = z + ab$$

specifies that the values of a and b should be multiplied together, the value of z added to this result, and then the final answer stored in the variables x. If machine code was being used, this sequence of operations would have to be written by hand. It seemed, however, that it should be possible to have the computer itself interpret the formula and generate the required machine code 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 to be more technically challenging than the interpretation of pseudocodes, primarily because mathematical formulae were syntactically more complex than the orders in a typical pseudocode. Earl Isaac broke the task of translation 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.[30]

The developers of formula translation systems in the early 1950s made different decisions about the 'grammar' of formulae. Some assumptions were widely shared: it was a common goal to allow formulae to 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 it was widely felt that it should be possible to include standard functions, such as trigonometric and exponential functions, in formulae. 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 formulae.

An early example of a formula translation system is provided by Mauchly's Brief and Short codes. According to William Schmitt, who developed a program which implemented the Brief Code proposals for the BINAC:

> A BRIEF CODE statement consisted of a destination variable symbol, an EQUALS operator and an expression made up of arithmetic symbols, variable symbols and parentheses.[31]

Initially, expressions could use the only the four basic arithmetical operators and parentheses, but other operators and functions were soon added. The interpreter associated numerical values with variables, and a statement allowed these values to be used to calculate a new value.

Because of the limited character sets available, formulae were transliterated by hand into a machine-readable representation. For example, a programmer might wish to code the statement represented symbolically as $X = Y + Z \times W$. On the BINAC, the variable symbols X, Y, Z and W were represented by the numerals 40, 41, 42 and 43, and the $=$ and $+$ operators by 03 and 02. To save space in the BINAC's small memory, the multiplication symbol was not represented explicitly; instead, consecutive variable symbols were interpreted as expressing an implicit multiplication. The formula above would therefore have been represented in Brief Code by the sequence 40 03 41 02 42 42.

The Laning and Zierler Program A more sophisticated system was written for the Whirlwind I computer at MIT by Laning and Zierler, and described by them as "a program for translation of mathematical equations".[32] They introduced their program and their understanding of it as follows:

> Nearly everyone who has had a problem to solve on a large-scale digital computing machine has probably felt that it would be indeed convenient if one could give the machine his problem in ordinary mathematical language with perhaps a suggestion for a method of solution. ... The effect of our program is to create a computer within a computer and the purpose of the present report is to describe this computer and provide a programmer's manual for its use.

[30] Isaac (1952).

[31] Schmitt (1988), p. 10.

[32] Laning and Zierler (1954), p. 1.

A basic goal of the program was to allow users to write formulae in a notation that was as close to that of normal mathematics as possible. The basic statements in programs were equations of the following form:

$$a = 5,$$
$$y = -6.3a,$$
$$b = 0.0053(a - y)/2ay,$$

which were interpreted as instructions to the computer to perform the computations indicated by the formulae on the right-hand side, and to store the results in the registers denoted by the letters on the left-hand side. Letters had to appear on the left-hand side of an equation and be assigned a value before they could be used in a formula on the right-hand side.

The program reflected normal mathematical notation to a considerable degree; for example, multiplication was represented by the juxtaposition of variables rather than by an explicit operator symbol. Parentheses could be used to group operations in a natural way, and a number of common trigonometric and exponential functions were available as function subroutines.

Although the formulae on the right-hand sides of equations could be interpreted conventionally, however, the same was not true of equations taken as a whole, and Laning and Zierler pointed out that examples such as

$$n = n + 2 \quad \text{and}$$
$$w = -w$$

should be interpreted not as expressing the equality of two terms, but as instructions to update the value of particular registers.

By default, the equations in a program were executed in the sequence they were written, but instructions for conditional and unconditional transfers of control were also provided. For example, the instruction SP n transferred control unconditionally to the equation numbered n, and CP n did the same if the last computed quantity was negative. Table 8.2 shows a complete example program to compute and print the value of $\cos x$ from 0 to 1 radian in steps of 0.1.

In common with other systems, Laning and Zierler's program allowed the use of subscripted variables, used in mathematical notation to represent the elements of vectors or sequences. Because of the limitations of the Whirlwind's input devices, subscripted variables were represented by superscripts; both constants and variables could be used as superscripts. Variables could be allocated to the Whirlwind's drum storage, and it was planned to make special provision for assigning tables of values to the drum variables, as in the following examples.

$$g|N = 2, 4, 6, 0,$$
$$a|N = 1.0(0.5), 2.0(0.25), 2.5(1), 4.5.$$

The first of these would assign the values 2, 4, 6 and 0 to the superscripted variables $g|^1$, $g|^2$, $g|^3$ and $g|^4$, respectively. The second denoted a sequence of values formed by applying the bracketed increments in the specified ranges. This is a notable proposal for automatic code generation: a single line of source code would be translated

Table 8.2 A program in Laning and Zierler's notation

	$x = 0$
1	$y = 10$
	$z = 1$
2	$z = 1 - zx^2/y(y-1)$
	$y = y - 2$
	$e = 1 - y$
	CP 2
	PRINT x, z
	$x = x + 0.1$
	$a = x - 1.05$
	CP 1
	STOP

into a complex loop to calculate a sequence of numbers. In this case the variables $a|^i$ would be assigned the values 1, 1.5, 2, 2.25, 2.5, 3.5 and 4.5.

Like other early formula translation systems, Laning and Zierler's program is as striking for what it leaves out as for what it includes. Viewed as 'a computer within a computer', it prevented the user from having access to the host computer, the Whirlwind. The virtual machine was very simple and, in common with other compiled pseudocodes, it did not provide any facility for instruction modification. The use of superscripted variables, however, provided a means of satisfying one of the common tasks that instruction modification was used for, namely providing convenient access to the items in a table of data.

Notational Diversity The different formula translation systems proposed in the 1950s varied considerably in the grammatical forms that they allowed programmers to use. At one extreme, even at the end of the decade the autocode developed 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.[33]

By contrast, Laning and Zierler's program went beyond the evaluation of single expressions, and even allowed certain systems of differential equations to 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:

$$Dy|^1 = y|^2 + 1,$$
$$Dy|^2 = -y|^1.$$

The basic idea underlying formula translation systems was the recognition that mathematical formulae encode algorithms which can be translated into machine code. A number of people 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

[33] Felton (1960).

the equivalent 'explicit' formula $y = 3x + 2$. There seemed to be nothing to prevent translators being written to generate machine code directly from implicit formulae, and techniques for doing this were proposed.[34]

This diversity of approach did not last long, however, and later programming languages show much less variety, typically supporting little, if anything, more than functional expressions written using explicitly specified arithmetical operators, parentheses and calls to subroutines. This 'standard form' first appeared in Fortran in the mid-1950s, subsequently spreading to other languages.

8.4 Fortran and Increasing Linguistic Complexity

Fortran, the most successful automatic programming system of the mid-1950s, was conceived by John Backus at IBM at the end of 1953, and a team of up to a dozen programmers developed the system over the next few years for the IBM 704. The first version of the system, Fortran I, was delivered to customers at the beginning of 1957.

Many earlier automatic programming systems had been rather inefficient, and had to a large extent gained support because they simulated hardware features that were not supported by existing machines. The 704 changed matters, however, by providing support for floating-point arithmetic and index registers. In this situation, Backus and his colleagues believed that the crucial factors governing the success of Fortran would be the speed at which the translation from source code into object code could be carried out, and above all the efficiency of the resulting object code. A preliminary report of 1954 describing the specifications for the system stated that:

> programming techniques have been devised which can be applied by an automatic coding system in such a way that an automatically coded problem, which has been concisely stated in a language that does not resemble a machine language, will be executed in about the same time that would be required had the problem been laboriously hand coded. Heretofore, systems which have sought to reduce the job of coding and debugging problems have offered the choice of easy coding and slow execution or laborious coding and fast execution.[35]

Aiming to provide easy coding and fast execution, the Fortran designers reportedly

> did not regard language design as a difficult problem, merely a simple prelude to the real problem: designing a compiler which would produce efficient programs.[36]

Nevertheless, the language they came up with was significantly more complex and sophisticated than its predecessors, bringing together two of the principal trends in automatic coding in the mid-1950s, namely formula translation and the automation of the coding of control structures.

Like the earlier formula translation systems, Fortran was not designed to be a universal language capable of completely replacing the use of machine code:

---

[34] Cleave (1960).

[35] IBM (1954), p. 1.

[36] Backus (1981b), p. 30.

> The FORTRAN language is intended to be capable of expressing any problem of numerical computation. In particular, it deals easily with problems containing large sets of formulae and many variables, and it permits any variable to have up to three independent subscripts. However, for problems in which machine words have a logical rather than a numerical meaning it is less satisfactory, and it may fail entirely to express some such problems.[37]

Talking about the language many years later, Backus was of the opinion that the original key ideas of the Fortran language had been the assignment statement, subscripted variables and the DO statement. There was much more to the language than this, however, and compared with other pseudocodes and formula translation systems, Fortran was described in great detail and with considerable precision. The metalinguistic terminology used to described the language changed over the years. The key construct in Fortran was the arithmetic formula, which computed the value of an expression and assigned it to a variable. In 1954, the word 'formula' was used to describe other procedural elements of the language, such as control and input–output formulae, and ancillary information was provided by so-called specification sentences. By 1956, however, Fortran programs were described as consisting of a sequence of statements. There were 32 different types of statement: as well as the arithmetic formulae, these were 15 control statements, 13 input–output statements and three specification statements.

An example Fortran I program is given in Table 8.3.[38] Fortran programs were written on special coding sheets and formatted in four columns which contained an indication of whether a particular line was a comment, an optional statement number, an indication of whether a line was a continuation of the previous line and lastly, a single Fortran statement.

Constants, Variables and Subscripts Higher level programming notations, such as pseudocodes and Fortran, cannot, of course, be completely insulated from details of the underlying machine that will ultimately execute the programs. One prominent area of interaction arises from the representation of numbers.

In order to generate correct code, the compiler of a pseudocode needs to know what kind of numbers are represented by the constants and variables in a program, so that the appropriate integer or floating-point routines can be called. This issue was finessed in the earliest systems by only supporting one representation, typically by using floating-point numbers exclusively. This had the consequence that integer values might not be held exactly, leading to problems when integers were required for control purposes such as checking whether a loop index had reached a particular value. In Laning and Zierler's system, for example, a loop that was to be executed ten times did not check whether an index variables had reached exactly 10, but instead subtracted 9.5 from it. If the result was negative, the loop would be repeated, terminating only when the result of the subtraction became non-negative.[39]

[37] IBM (1956), pp. 2–3.

[38] IBM (1956), p. 46.

[39] Laning and Zierler (1954), p. 8.

Table 8.3 A Fortran I program

```
C       PROGRAM FOR FINDING THE LARGEST VALUE
C     X    ATTAINED BY A SET OF NUMBERS
        DIMENSION A(999)
        FREQUENCY 30(20,1,10), 5(100)
        READ 1, N, (A(I), I = 1, N)
    1   FORMAT (I3/(12F6.2))
        BIGA = A(1)
    5   DO 20 I = 2, N
   30   IF (BIGA-A(I)) 10, 20, 20
   10   BIGA = A(I)
   20   CONTINUE
        PRINT 2, N, BIGA
    2   FORMAT (22H1 THE LARGEST OF THESE I3, 12H NUMBERS IS F7.2)
        STOP 77777
```

In PACT I, information about the types of numbers stored in particular variables was "supplied to the compiler by means of the variable definition sheet which is loaded along with the arithmetic and logical instructions".[40] In Fortran, however, all this information was presented in a single linguistic framework. Fortran supported both fixed- and floating-point numbers; fixed-point numbers were in fact restricted to integers, and constants of the two types could be told apart by the presence or absence of a decimal point. Variables could store one or the other type of number, depending on the initial letter of the variable name: those beginning with I, J, K, L, M, or N stored integer values, and the rest floating point.

Many pseudocodes supported the mathematical convention of using subscripted variables to represent elements of a sequence or vector of data values. Subscripted variables were placed in an array in memory, so that the subscript could be used to modify the base address of the variable and so access each element of the array in turn. Fortran required programmers to declare the use of subscripted variables in a DIMENSION statement which stated the size of an array, and so the highest possible subscript value.

The use of symbolic variables names allowed programmers to leave details of storage allocation up to the compiler. However, many codes allowed programmers to specify when different variables could be safely allocated to the same memory location at different times in the program. This technique provided a useful way of enabling optimal use of scarce memory resources, but it did require programmers to be certain that conflicts would not arise. In Fortran, another specification statement, the EQUIVALENCE statement, allowed the programmer to specify these details of storage allocation.

[40] Baker (1956), p. 279.

Expressions and Arithmetic Formulae A primary purpose of Fortran was, of course, formula translation. This was carried out in statements known as *arithmetic formulae*.

> A FORTRAN arithmetic formula resembles very closely a conventional arithmetic formula; it consists of the variable to be computed, followed by $an = \operatorname{sig} n$, followed by an arithmetic *expression*. For example, the arithmetic formula
>
> $$Y = A - \mathrm{SINF}(B - C)$$
>
> means "replace the value of y by the value of $a - \sin(b - c)$".[41]

This definition clearly distinguishes two different 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. Like some earlier systems, Fortran continued to use the standard equality symbol = in formulae, despite the potential for confusion between its conventional meaning and the new usage. The Fortran reference manual explained its new, computational meaning as follows.

> 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.[42]

Notice that this interpretation of what was apparently an equation ruled out any possibility of the system handling implicit computations such as $y - 2 = 3x$.

A striking feature of the definition of arithmetic formulae was the way in which the syntactic form of expressions was specified. As well as an informal definition, expressions were given a formal recursive definition which was very similar in style to the definitions given in logic texts of the terms of formal languages. In the original specification for the system, the following informal definition was given:

> Any sequence of variables and functions separated by operation symbols and parentheses which forms a meaningful mathematical expression in the normal way. Note that every adjacent pair of variables or functions must be separated by an operation symbol. [43]

This definition emphasises both the similarity between Fortran and mathematics, and also the possibility of difference: the last sentence, for example, rules out the use in Fortran of juxtaposition to express multiplication. It relies entirely on readers' intuition, however, about what constituted 'normal' mathematical expressions.

The formal definition removed any such ambiguities:

> By repeated use of the following rules, all legal expressions may be derived and all expressions so derived are legal provided they have less than 750 characters.
>
> (i) Any constant or variable is an expression.
> (ii) If E is an expression not of the form $+F$ or $-F$, then $+E$ and $-E$ are expressions.
> (iii) If xxx denotes a function of n arguments, and if $E_1, E_2, \ldots, E_n$ are expressions, then in general $\mathrm{xxx}(E_1, E_2, \ldots, E_n)$ is an expression. ...
> (iv) If E is an expression, so is (E).

[41] IBM (1956), p. 12.

[42] IBM (1956), p. 16.

[43] IBM (1954), p. 6.

(v) If E and F are expressions where F is not of the form $+G$ or $-G$ and o is one of the permissible binary operations, then EoF is an expression.
(vi) If E and F are expressions, so is $E \times \times F$.[44]

The 'permissible binary operations' referred to in clause (iv) were $+$, $-$, $\times$ and $/$, and $\times\times$ represented exponentiation. In general, adjacent operation symbols were not permitted; clause (vi), however, permitted the formation of expressions such as $1.53 \times 10 \times \times - 14$.

It is striking that a very similar definition was given in the programmer's manual of 1956: this brought a level of formality to the general description of programming notations that was at the time unusual. The later definition was completely general, removing the restriction on the maximum length of expressions, and also made it clear that expressions could refer to fixed- or floating-point quantities; this property was referred to as the *mode* of the expression. The key clauses in the later definition read as follows:

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, and if the first character of F is not $+$ or $-$, then

$$\begin{array}{c} \mathsf{E}+\mathsf{F} \\ \mathsf{E}-\mathsf{F} \\ \mathsf{E}*\mathsf{F} \\ \mathsf{E}/\mathsf{F} \end{array}$$

are expressions of the same mode. Thus $A - +B$ and $A/ + B$ are not expressions.[45]

By and large, expressions in later languages were defined in a similar way to Fortran, and experiments like the inclusion of differential equations in Laning and Zierler's system are no longer seen. This raises the question of why this particular definition 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 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 formulae given by Fortran that might account for its differential success and persistence?

One possible answer is that it was the style of the definition that gave rise to its success. 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

[44] IBM (1954), pp. 6–7. Clause (iii) also stated that certain functions might place restrictions on the syntactic forms of their arguments.

[45] IBM (1956), p. 14.

approach to the design of programming languages which appealed to the authority and established results and techniques of the discipline of logic. Later developments, and in particular the approach taken towards Algol, as described later in this chapter, provide some support for this idea.

It is worth noting that 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 such as programming notations, mechanical theorem proving, and the formal representation of computing machines. Among these was a talk by Peter Sheridan, who had designed the parts of the compiler responsible for interpreting expressions, in which he briefly described the Fortran system.[46]

Statements The use of formal techniques of language specification in the Fortran manual was limited to the definition of expressions, however, and the remaining statements of the language were only defined informally.

A number of different statements were provided to define the flow of control through the program. The GO TO statement specified an unconditional jump to the statement labelled with a particular statement number, and variants allowed the destination of the jump to be determined by a previously assigned or computed value. Conditional transfer was provided by the IF statement, which took the form:

> "IF (a) n_1, n_2, n_3" where a is any expression and n_1, n_2, n_3 are statement numbers.[47]

Following the execution of an IF statement, control was transferred to the statement numbered n_1, n_2 or n_3 depending on whether the value of the expression a was less than, equal to or greater than zero, respectively. A number of special forms of IF statement tested various machine conditions of the 704.

Automatic coding of loops was primarily supported by the DO statement, which performed a similar role to the SET and TEST instructions of PACT I. A statement of the following form:

$$\text{DO } 30\ I = 1,\ M, 3$$

would repeatedly execute the statements in the *range* of the DO statement, in this case those following it up to and including the statement with label 30. The variable I would first take the value 1, and would be incremented by 3 each time the statements in the range were completed. Once the next increment would cause the value of I to exceed the value of M, execution of the loop would finish.

The occurrence of a DO statement in a Fortran program therefore explicitly marked the beginning of a loop. The end of the loop was only marked implicitly, however, by the statement label given in the DO statement. The last statement of a DO loop could not be a GO TO or IF statement, however, so a dummy statement was defined for use in these cases:

[46] Sheridan (1957).

[47] IBM (1956), p. 18.

> CONTINUE is a dummy statement which gives rise to no instructions in the object program. Its most frequent use is as the last statement in the range of a DO, where it provides a statement number which can be referred to in transfers which are desired to become, in the object program, transfers to the indexing instructions at the end of the loop.[48]

In many cases, CONTINUE statements were regularly used, however, and became a *de facto* marker of the end of loops, as in the program of Table 8.3.

The authors of the Fortran manual did not find it straightforward to specify the exact details of the sequencing of DO statements and the interactions between DO statements and other transfer statements. An indication of this is given in the quote above: it is striking that an explanation of the behaviour of a dummy statement seems to require reference to properties of the object code that the compiler would produce. The first issue concerned the extent to which the ranges of two different DO statements could overlap:

> **Rule 1.** If the range of one DO includes another DO, then all of the statements in the range of the latter must also be in the range of the former.

In this case, the statements were said to be *nested*. Nested DO statements could specify the same statement number as the end of their ranges; it would therefore be possible in Fortran for a single CONTINUE statement to mark the end of more than one loop. A second rule concerned the interaction between various transfer statements:

> **Rule 2.** No transfer is permitted into the range of any DO from outside its range.[49]

There was an exception to this rule, however, which permitted transfer from a nest of DO statements to an unconnected piece of program which made no change to any of the index variables in the nest; this was provided to make "it possible to exit temporarily from the range of a DO to execute a subroutine". Further rules concerned the calculations that could be performed on the index variables of loops, and the values that these variables had on exit from a loop.

Automatic loop writing also featured in a number of input and output statements. For example, the statement

$$\text{READ } 1, N, (A(I), I = 1, N)$$

would first read a value into the variable N. Further values would then be read into the subscripted variables $A(1), A(2), \ldots$, input stopping once N values had been read. The '1' preceding N is a reference to a FORMAT statement which provided details of the expected format of the data being read, as shown in Table 8.3. The details of format statements are not important here, however.

Semantics of Fortran Linguistically, then, Fortran can be viewed as an attempt to combine the benefits of formula translation with the features for automatic coding of control statements introduced by pseudocodes such as PACT I. This dual approach

[48] IBM (1956), p. 22.

[49] IBM (1956), p. 21.

produced a language that was significantly more complex and sophisticated than its contemporaries. The design of specific features of the language seems to have been influenced by logic, but a more significant motivation in many cases was the desire to produce object code that was as efficient as possible.[50]

Contemporary accounts of the semantics of the language made reference to both the established accounts of pseudocode semantics:

> The IBM Mathematical Formula Translating System FORTRAN ... is a 704 program which accepts a *source program* written in a language—the FORTRAN language—closely resembling the ordinary language of mathematics, and which produces an *object program* in 704 machine language.
> FORTRAN therefore in effect transforms the 704 into a machine with which communication can be made in a language more concise and more familiar than the 704 language itself, The result should be a considerable reduction in the training required to program, as well as in the time consumed in writing programs and eliminating their errors.[51]

Some features of the language definition, and in particular the formal account given of the syntax of expressions, mark the introduction of established metalogical techniques into the definition of programming languages. However, it did not prove possible to maintain a strict distinction between syntax and semantics throughout. For example, the question of whether something as simple as GO TO 3 was a legal statement depended on whether the statement labelled 3 was in the range of a DO statement, and if it was, whether the overall behaviour of the program satisfied the complex semantic rules laid down for the DO statement.

8.5 Universal Languages

By the end of the 1950s, a large number of automatic programming systems of various kinds were in existence. Most of them had very restricted use, however, and the overall picture was one of fragmentation. At the beginning of 1961, the *Communications of the ACM* used the image of the tower of Babel to reflect the confusion and lack of communication that was felt to exist at the time.

One reason for this diversity was the fact that most systems were only designed for and usable on a single type of computer. Machine code programs were obviously machine specific, but surveys at the end of the 1950s showed that in addition most automatic programming systems could only be used on a single type of machine.[52] Even Fortran was, by the end of the decade, only available on two machines, the IBM 704 and the IBM 709.

Another factor leading to diversity was the perception that different notations were required for different application areas. Fortran quickly became the *de facto* standard for scientific programming, but it was felt that it was too mathematically

[50] Backus and Heising (1964).

[51] IBM (1956), p. 2.

[52] Bemer (1959).

oriented for business purposes. Commercial uses of computers centred around data processing rather than mathematical calculation, and a number of language designs were made aimed specifically at this market. 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.

Research also led to new languages: by the latter half of the 1950s it was 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 he decided 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",[53] 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,[54] Hamblin's ideas for a so-called 'zero-address' computer were implemented in the architecture of a later computer, the KDF9.

Attempts at Standardization During the 1950s, a number of groups were formed to enable the users of particular computers to communicate with each other and share experiences and knowledge. Perhaps the best known of these was SHARE, founded in 1955 as a "cooperative programming group for IBM 704 users".[55] As the name implies, one aim of the group was to enable programmers of the 704 to benefit from each other's work. 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.[56]

and the difficulties involved in sharing code between users of different machines were even greater.

There was therefore a growing recognition of the desirability of establishing common standards and languages that would enable results and experience to be shared between different groups. Mathematical formulae provided a standard way of expressing simple computational procedures, but traditional mathematics did not define a universally accepted notation for expressing the sequencing of operations

[53] Hamblin (1957), p. 135.

[54] Hamblin (1958).

[55] SHARE (1958a).

[56] SHARE (1958b).

in more complex algorithms. The notations that did exist were machine specific: Fortran marked a significant step forward, but had not yet been implemented on a significant number of machines.

In fact, a number of machine-independent programming notations had already 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 an account of his *Plankalkül* notation, based on work he had carried out immediately after the war.[57] These proposals do not appear to have influenced the development of programming notations, however, in part because no compilers or interpreters for the notations had yet been developed.

Algol 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 ACM, the American *Association for Computing Machinery* proposing a conference which would have the aim of defining a common formula translation language.[58] 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). In the light of subsequent developments, this language is often referred to as Algol 58; Table 8.4 shows a sample Algol 58 program.[59]

Algol was not intended to be a universal language that would replace all others: the aim rather was to produce a language for scientific programming that would unify existing proposals and become a standard in this particular field. It was not viewed simply as a programming language, but also as a medium for the expression and communication of algorithms. The first two objectives defined in the Algol 58 report were that the language should be "as close as possible to mathematical notation" and usable for "the description of computing processes in publications", and in support of this ambition, in 1960 the *Communications of the ACM* started a "new editorial department … to publish algorithms consisting of 'procedures' and programs in the ALGOL language".[60]

The third objective of Algol 58 was that it "should be mechanically translatable into machine programs", but unusually for a language proposal of that time, the report made no reference to the features of any particular computer. Ironically, this perhaps made it harder for Algol to achieve its goal of universality. Attempts to implement the language on different machines and for different purposes quickly

[57] Zuse (1948).

[58] Bauer et al. (1957).

[59] Perlis and Samelson (1958). It is interesting to note that, even though Algol 58 contained no formatting rules, this text was presented in columns, in a manner reminiscent of Fortran.

[60] Wegstein (1960).

Table 8.4 An Algol 58 program to integrate $F(x)$ by Simpson's rule

procedure	Simps($F(\,)$, a, b, delta, V);
begin	
Simps:	Ibar $:= V \times (b - a)$
	$n := 1$
	$h := (b - a)/2$
	$J := h \times (F(a) + F(b))$
J1:	$S := 0$;
for	$k := 1(1)n$
	$S := S + F(a + (2 \times k - 1) \times h)$
	$I := J + 4 \times h \times S$
if	(delta $<$ abs(I − Ibar))
begin	Ibar $:= I$
	$J := (I + J)/4$
	$n := 2 \times n$; $h := h/2$
	go to J1 *end*
	Simps $:= I/3$
return	
integer	(k, n)
end	Simps

led to the creation of dialects such as NELIAC and JOVIAL, which soon took on a life of their own as, effectively, separate new languages.[61]

Following extensive discussion of the Algol 58 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.[62] Even more than its predecessor, Algol 60 was presented as a language for communication, rather than as a notation to control the behaviour of computers: the report stated that "[t]he purpose of the algorithmic language is to describe computational processes". Table 8.5 shows a sample Algol 60 program, and following sections examine the Algol languages in greater detail.

UNCOL 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 so-called UNCOL project, sponsored by SHARE. 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".[63] At a meeting in February, 1958, discussion took place on "ways to

[61] See Shaw (1963), for example, for a description of JOVIAL. Development of both NELIAC and JOVIAL began in 1959.

[62] Naur et al. (1960).

[63] SHARE (1958a).

Table 8.5 An Algol 60 procedure to sum fct(i) from 0 to infinity

```
procedure euler (fct, sum, eps, tim); value eps, tim;
integer tim; real procedure fct; real sum, eps;
begin integer i, k, n, t; array m[0:15]; real mn, mp, ds;
i := n := t := 0; m[0] := fct(0); sum := m[0]/2;
nextterm: i := i + 1; mn := fct(1);
        for k := 0 step 1 until n do
          begin mp := (mn + m[k])/2; m[k] := mn; mn := mp end means;
        if (abs(mn) < abs(m[n])) ∧ (n < 15) then
          begin ds := mn/2; n := n + 1; m[n] := mn end accept
        else ds := mn;
        sum := sum + ds;
        if abs(ds) < eps then t := t + 1 else t := 0;
        if t < tim then go to nextterm
end euler
```

develop a universal language for the computing field",[64] and over the coming year a sub-committee of SHARE developed proposals to address this need.

The UNCOL project drew a distinction between machine or computer-oriented languages (COLs) and problem-oriented languages (POLs). Rather than trying to define a single problem-oriented language, like Algol, the idea behind the project was to define a universal computer-oriented language, or UNCOL. This language would be at a higher level than machine code, but at a lower level than languages like Fortran or Algol. An implementation of UNCOL was to be written for every target machine, but because of its lower level, it was felt that this would be easier than implementing a language like Algol across a range of hardware.

This approach was believed to promise a number of benefits. Firstly, there was a belief that the Algol approach was in principle wrong, and that there would be a need for many different problem-oriented languages, each tailored to the needs of programmers in particular application areas. Secondly, it was anticipated that the implementation of a new problem-oriented language would require only a POL-to-UNCOL translator to be written, and that this would be much easier that writing 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.[65] It did not prove possible 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.

[64] SHARE (1958b).

[65] Steel (1961).

8.6 Algol 60 as a Formal Language

The influence of Algol 60 on the development of programming and programming languages in the 1960s is described in the following chapter. The extent of this influence has less to do with the practical success of the language than with the way in which it was defined. Unlike 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's discussions of logical syntax:

> For this universal language, it will be necessary, just as for programming systems, to begin by defining its elements: firstly the elementary symbols and their different roles, then the formation rules for obtaining terms out of aggregates of these symbols, and finally the rules for constructing expressions out of terms or simpler expressions. It would seem that experience of current automatic programming could accelerate the purely logical study of the language as a formal system.[66]

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.[67] 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 required that the set of symbols used for constructing expressions in a language be clearly defined. The symbols used for expressing machine-specific order codes and pseudocodes were not usually explicitly listed. Instead, it was assumed that programs would be written using a subset of the characters available on the input devices of the computer that the code was to run on. It took 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. The Fortran I programmer's manual contained a "table of Fortran characters".[68] This showed the 48 characters available on the IBM 704 together with the different ways they were coded on punched cards, on paper tape and within the 704 itself. 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 that could appear in program output. The '$' symbol, meanwhile, could only be used in a program within textual data that were to be output.

---

[66] Puyen and Vauquois (1960), p. 134. In French in the original.

[67] Woodger (1960).

[68] IBM (1956), p. 49.

Sheridan attempted to formalize this situation by defining an explicit 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 '⊣' which was "not a character explicitly indicated in any FORTRAN statement, serving solely as a statement endmark on the executive level",[69] or in other words, not a symbol of the Fortran language at all. Neither of these approaches, then, succeeded in defining correctly the set of characters that could appear in legal Fortran source 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*.[70]

Of these, the reference language was the one used to defined the language, and consisted of "only one unique set of characters". The publication language had to ensure "univocal correspondence" with the reference language, but would allow for the use of formatting conventions used in print, such as subscripts and superscripts; different publication languages could be defined to accommodate such things as varying national conventions for showing decimal points. Finally, each Algol 58 implementation would require a machine-specific hardware representation, whose details would depend on the capabilities of the target machine. The correspondence between the different notations was to be specified by transliteration rules.

The reference language of Algol 58 was defined in terms of a set of basic symbols rather than a simple character set. The set of basic symbols contained a mixture 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 for 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. It therefore became possible to think of the alphabet of the language as 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 representa-

[69] Sheridan (1959), p. 11.

[70] Perlis and Samelson (1958), p. 9.

> tion of an internal symbol. We can therefore say that the processing of internal symbols can be independent of the external language.[71]

In this respect, Algol differed from traditional accounts of formal languages which treated the expressions of a language as sequences of characters drawn from a fixed set. The variety of representations considered focused attention on the abstract structure of expressions rather than a particular representation, and this structure, rather than the sequence of characters, came to be seen as defining the expression.

One consequence of this was the way in which white space was treated. Every input device included characters representing spaces or new lines, and these could be included in Algol programs. However, there was no basic symbol corresponding to either of these characters, and the Algol 60 report stated that:

> Typographical features such as blank space or change to a new line have no significance in the reference language. They may, however, be used freely for facilitating reading.[72]

Object Language 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 use of "syntactical Gothic symbols",[73] the Algol 58 report had used 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 was defined by

> *Figures* ζ (arabic numerals $0, \ldots, 9$)

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 were defined as follows:

> Form: $N \sim G.G_{10} \pm G$ where each G is an integer as defined above.[74]

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, "[t]here must exist a precise definition of those sequences of symbols which constitute legal IAL [i.e. Algol 58] programs". He went on to provide a formal syntactic metalanguage sufficiently powerful to express such a definition; it was explained as follows:

> To begin with, we shall need *metalinguistic formulae*. Their interpretation is best explained by an example:
>
> $$\langle ab \rangle :\equiv (\text{or } [\text{or } \langle ab \rangle \ (\text{or } \langle ab \rangle \langle d \rangle .$$

[71] Green et al. (1959).

[72] Naur et al. (1960), p. 301.

[73] Carnap (1937), p. 15.

[74] Perlis and Samelson (1958), p. 11.

> Sequences of characters enclosed in "⟨⟩" 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 ⟨*ab*⟩. It indicates that ⟨*ab*⟩ may have the value "(" or "[" or that given some legitimate value of ⟨*ab*⟩, another may be formed by following it with the character "(" or by following it with some value of the variable ⟨*d*⟩.[75]

Backus' notation was used by Peter Naur in the Algol 60 report. By the time this report was produced, this notation had been adapted slightly: "::=" replaced ":≡" and "|" replaced "or". In addition, the report made it explicit that "the symbols used for distinguishing the metalinguistic variables (i.e. the sequences of characters appearing within the brackets ⟨⟩ ...) have been chosen to be words describing approximately the nature of the corresponding variable"; Naur commented later that "this was intended to provide an "immediate link between syntax and semantics".[76] In the final notation, the definition of the syntax of integer constants appeared as follows:

⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9,

⟨unsigned integer⟩ ::= ⟨digit⟩ | ⟨unsigned integer⟩⟨digit⟩,

⟨integer⟩ ::= ⟨unsigned integer⟩ | +⟨unsigned integer⟩ | −⟨unsigned integer⟩.

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. Davis, however, 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.[77] There was some awareness within the computing community of Post's work, however: Rosenbloom's textbook of 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.[78] 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.[79]

Whatever the origins of the notation, however, the fact that the Algol 60 report made explicit use of the logical distinction between object and metalanguage drew attention to the importance of giving an unambiguous definition of the syntax of a programming language. Furthermore, the specific formal notation introduced for syntactic specification was highly influential and widely emulated, and was even applied retrospectively to existing languages such as Fortran.[80]

---

[75] Backus (1959), p. 129.

[76] Naur et al. (1960), p. 301, Naur (1981).

[77] Backus (1980, 1981a), Davis (1988).

[78] Rosenbloom (1950), Elgot (1954), for example.

[79] Bauer (1981), Samelson (1981).

[80] 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

⟨string⟩ ::= ⟨any sequence of basic symbols not containing 'or'⟩,

where the 'metalinguistic variable' on the right-hand side is actually an informal description of the intended set of strings, and does not appear on the left-hand side of any other production. To a much greater extent than any previous document, however, the Algol 60 report succeeded in formally defining which texts were legal programs of the language.

Textually, Fortran programs were considered to be sequences of statements. There were many different types of statement which played very different roles in a program. Algol employed rather different terminology, and identified three major syntactic categories, known as expressions, statements and declarations.

These categories were distinguished by the different semantic roles played by their elements. Expressions denoted values, and as in Fortran, algebraic formulae were treated as expressions which denoted numbers. Expressions could also denote different types of value, however: boolean expressions denoted one of the values *true* and *false*, and designational expressions denoted program labels. Statements described the operations to be performed by a program: simple operations, such as the assignment of a value to a variable or an unconditional transfer of control, were defined by basic statements, but statements could also define complex actions which involved multiple basic operations, and also control structures such as loops. Finally, declarations defined properties of entities that would be used elsewhere in the program, such as variables and subprograms.

Recursive definitions were used to specify the syntax of both expressions and statements, as in the following example from Algol 58:

> Strings of one or more statements may be combined into a single (compound) statement by enclosing them within the "statement parentheses" *begin* and *end*. Single statements are separated by the statement separator ";".
>
> Form: $\Sigma \sim$ *begin* Σ; Σ; ...; Σ *end*.[81]

The use of the term 'statement parentheses' highlighted the similarity between this definition and the recursive definition of expressions that Fortran had made familiar. Rule 4 of that definition stated that a parenthesized expression was itself an expression, and this allowed parentheses to be used to construct nested expressions of any required degree of complexity. In exactly the same way, Algol's definition of compound statements allowed them to be nested to arbitrary depths.

In Algol 58, declarations could be written at any point in the program and appear in any order. The facts they stated applied throughout the program; for example, the declaration of the type of a particular variable could appear after all occurrences of the variable. In Algol 60, however, declarations could only appear at the beginning of compound statements, and a compound statement which included declarations

[81] Perlis and Samelson (1958), pp. 13–14.

was known as a block. The definition of blocks and compound statements in Algol 60 was given by the following productions.

⟨unlabelled basic statement⟩ ::= ⟨assignment statement⟩ | ⟨go to statement⟩
| ⟨dummy statement⟩ | ⟨procedure statement⟩,
⟨basic statement⟩ ::= ⟨unlabelled basic statement⟩ | ⟨label⟩ : ⟨basic statement⟩,

⟨unconditional statement⟩ ::= ⟨basic statement⟩ | ⟨for statement⟩
| ⟨compound statement⟩ | ⟨block⟩,
⟨statement⟩ ::= ⟨unconditional statement⟩ | ⟨conditional statement⟩,

⟨compound tail⟩ ::= ⟨statement⟩ **end** | ⟨statement⟩; ⟨compound tail⟩,

⟨block head⟩ ::= **begin**⟨declaration⟩ | ⟨block head⟩; ⟨declaration⟩,

⟨unlabelled compound⟩ ::= **begin**⟨compound tail⟩,

⟨unlabelled block⟩ ::= ⟨block head⟩; ⟨compound tail⟩,

⟨compound statement⟩ ::= ⟨unlabelled compound⟩
| ⟨label⟩ : ⟨compound statement⟩,
⟨block⟩ ::= ⟨unlabelled block⟩ | ⟨label⟩ : ⟨block⟩.

As this definition shows, compound statements could include any other statements, including nested compound statements, and blocks could contain declarations and statements. As the report went on to explain,

> This syntax may be illustrated as follows: Denoting arbitrary statements, declarations, and labels, by the letters S, D, and L, respectively, the basic syntactic units take the forms:
>
> *Compound statement:*
> L: L: ... **begin** S; S; ... S; S; **end**
>
> *Block:*
> L: L: ... **begin** D; D; ... D; S; S; ... S; S; **end**
>
> It should be kept in mind that each of the statements S may again be a complete compound statement or block.[82]

The recursive approach to the definition of statements also changed the way in which control structures were defined. A DO statement in Fortran was the single line defining the properties of the index variable and the number of last statement in the range of the DO statement. In Algol 58, by contrast, the range of the analogous *for* statement was defined to be the single statement following the *for* statement. If a range of more than one statement was required, this could be achieved by using a compound statement. Among other things, this simplified the language definition, for example by removing the need to consider cases where the ranges of two loops overlapped: the form of the syntactic definition ruled this out as a possibility.

In Algol 58, however, a loop still consisted syntactically of two statements, one of which defined the index properties and the other the range. Algol 60 simplified this by defining the *for* statement as a single statement which included both the index information and a nested statement corresponding to the range.

[82] Naur et al. (1960), p. 306.

⟨for clause⟩ ::= **for** ⟨variable⟩ := ⟨for list⟩ **do**,

⟨for statement⟩ ::= ⟨for clause⟩⟨statement⟩ | ⟨label⟩ : ⟨for statement⟩.

As shown above, for statements were themselves statements, so this definition rather elegantly allowed for unrestricted nesting of for statements. Conditional statements were similarly defined, and the various forms could be combined, allowing for a great flexibility in program structure.

It was argued above that one major innovation of Fortran was to give a recursive definition of the structure of arithmetical expressions. The Algol 60 report gave a similar account of statements, giving them the same combination of simplicity and potential complexity as mathematical formulae. Thus an Algol program text could 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 formulae is identified. These are the formulae 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 formulae 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 is a program and what is not. This approach took a while to evolve, however. The Algol 58 report gave 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.[83]

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.[84]

It should be noted, however, that even this definition could not be captured in BNF, and that no production was given defining the metalinguistic variable 'program'.

[83] Perlis and Samelson (1958), p. 10.

[84] Naur et al. (1960), p. 200.

Semantics In 1959, Backus stated what he thought was be required of a semantic account of Algol.

> 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).[85]

Compared with earlier accounts of the semantics of pseudocodes, this is notable for putting the emphasis on the meaning of individual programs rather than of the language as a whole, and also for giving an informal description of the domain of program meanings. This marks a further alignment of the Algol approach with that of logic, where accounts such as Tarski's gave a formal definition of the meaning of each sentence in the language. Backus went on to promise that "the formal treatment of the semantics of legal programs will be included in a subsequent paper". No such paper appeared, however, and in the Algol 60 report the semantics of the language were defined informally.

In the Algol 58 report, the major syntactic categories of expressions, statements and declarations had been distinguished by their differing semantic roles. In more detail, 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"; 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", and declarations "state certain facts about entities referred to within the program".[86]

A very similar approach was adopted for Algol 60. The descriptions of many syntactic categories were accompanied by an account of the meaning the formulae of that category, suggesting the intention to produce a compositional account of the semantics. 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 made explicit. This was probably a consequence both of the machine-independent aspirations of the language, and also its complexity. 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.

---

[85] Backus (1959), p. 129.

[86] Perlis and Samelson (1958), pp. 13, 17.

8.7 The Influence of Logic on Algol

The previous section described how Algol was presented as a formal language, using the metalinguistic 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, by contrast, 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 Fortran's IF statement was basically just 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 first statement depend on the truth-value of a given condition. For example, in

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

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 led 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.[87]

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

$$\textbf{if}\ a > 0 \quad \textbf{then}\ c := a \uparrow 2 \times b \uparrow 2,$$

defined 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.

[87] Perlis and Samelson (1958), p. 14.

Substitution Substitution of an expression for a variable in a formula is a technique widely used in logic and the λ-calculus as a means of generating new formulae from old. Algol 58 defined a mechanism for textual substitution, 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 arbitrary string of symbols $S_{\rightarrow}$, provided that the resulting text was syntactically legal. The effect was defined to be the same as that of executing the resulting code in place of the *do* statement.

Although having a completely different meaning, this proposal shared with the similarly-named DO statement in Fortran the property of referring to a range of statements. This made it necessary to state a number of obscure rules about the required properties of the statements within the rules, and the effect of nested *do* statements. Perhaps because it did not fit well with the evolving recursive concept of statements, the *do* statement was removed from Algol 60. However, the notion of textual substitution was preserved in the 'call by name' mechanism.

Subroutines and Parameter Passing Subroutines had been a prominent feature of machine code programming, and the methodological advantages of splitting a large program into a number of independent and reusable components were well known. Integrating the subroutine concept with autocodes and formula translation languages proved not to be straightforward, however. Fortran I defined a number of functions which could be called in expressions, but provided no way for programmers to create their own function or subroutine definitions.

In 1958, both Fortran II[88] 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 was to decide how data are 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

[88] IBM (1958).

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,[89] but there appears to be no evidence of mutual influence between this work and thee 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 formal subroutine parameters 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)".[90] The use of the term 'local' here makes a connection between variable binding and the Algol notion of 'block'.

In Algol 60, a sequence of statements enclosed within the special brackets *begin* and *end* was a *compound statement*, and a *block* was a defined to be 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 that:

> 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.[91]

Table 8.6 shows a rather artificial example of one block nested inside another. Blocks implied a particular mechanism for the allocation of storage to variables. In this example, 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

[89] Barcan Marcus (1962).

[90] Strachey and Wilkes (1961), p. 489.

[91] Naur et al. (1960), p. 9.

Table 8.6 Blocks in Algol 60

```
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
```

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, provided that they are not referred to by a identifier defined in the inner block; the assignment to i above illustrates this. 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 the bound variables of logic: they are inaccessible from 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.[92] 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".[93]

8.8 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 example of mathematical 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,

[92] Naur et al. (1960), p. 12.

[93] Wells (1980).

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[94] 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.[95] 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 applications in the new area of artificial intelligence. Experience had indicated that programs in this area needed 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, Allen Newell and Herbert 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".[96] They therefore developed a pseudocode designed specifically to support the operations required in this application. Initially known as the "logic language" and described as a "formal language", even though it was not defined in a particularly formal manner, this code evolved into a family of notations known collectively as 'Information Processing Language' (IPL).

The key data structures for this class of problem became known as *lists*, and Newell and Simon explained that "IPL-V allows two kinds of expressions: *data list structures*, which contain the information to be processed, and *routines*, which define information processes".[97] The system was conceived of as a virtual machine, the "IPL Computer", whose memory was designed to be suitable for storing list structures and which provided a number of primitive processes, analogous to the arithmetic operations found 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 pseudocodes.

Lisp itself combined the algebraic approach adopted by Fortran with the data structures used in IPL, and was characterized by McCarthy as an "algebraic list-processing language".[98] Whereas IPL, like machine code, only permitted sequences of the basic list operations to be written, McCarthy's approach allowed complex expressions to be formed, analogous to the conventional algebraic expressions that were supported by Fortran.

Like Fortran, Lisp was originally referred to as a programming system, one which was "based on a scheme for representing the partial recursive functions of a certain

[94] Taylor (1960).

[95] McCarthy (1960).

[96] Newell and Simon (1956), p. 62.

[97] Newell and Tonge (1960), pp. 205–206.

[98] McCarthy (1981), p. 174.

class of symbolic expressions".[99] 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 notation for describing partial functions. As well as the standard means of forming new function from old by using substitution and definition by recursion, a new notation for *conditional expressions* 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 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*, denoted 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 $(a_1 \cdot (a_2 \cdot (\cdots (a_n \cdot \text{NIL}) \cdots)))$, where NIL was a distinguished atom representing the empty list.

Having defined *S*-expressions and lists, the next step was to define some primitive functions to manipulate them. McCarthy defined five elementary functions: *atom* tested whether or not an *S*-expression was an atom, *eq* tested whether two atoms were equal, *cons* constructed an *S*-expression to be constructed out of two atoms or *S*-expressions, and *car* and *cdr* retrieved the two components making up a non-atomic *S*-expression. 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 then defined to represent functions over *S*-expressions. To distinguish these, from *S*-expressions, they were written using lower-case letters and different forms of punctuation. For example, a function 'ff' which returned the first atomic symbol in an *S*-expressions could be defined by the following *M*-expression:

$$\text{ff}[x] = \big[\text{atom}[x] \to x; T \to \text{ff}\big[\text{car}[x]\big]\big]$$

At 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".[100]

[99] McCarthy (1960), p. 184.

[100] McCarthy (1960), p. 189.

Although McCarthy did not make this explicit, this was a form of arithmetization. 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".[101] *apply* is universal in the following sense:

> if f is an *S*-expression for an *S*-function f' ... then apply[f; arg s] and f'[arg 1; ...; arg n] are defined for the same values of arg 1, ..., arg n, and are equal when defined.[102]

apply, therefore, is capable of 'simulating' every other *S*-function, once 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 which represent 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. Instead, Algol and Lisp can 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.

[101] McCarthy (1960), p. 184.

[102] McCarthy (1960), p. 189.

8.9 Conclusions

This chapter has followed one path through the complex history of the development of programming languages in the 1950s, and argued that the aim of automating parts of the programming and coding processes led, through the development of systems for formula translation, to an understanding of programming notations themselves as being formal languages, a view that was made most explicit in the Algol 60 proposal. In Sect. 8.6 it was argued that Algol was explicitly defined as a formal language, in the same way as logical notations, and Sect. 8.7 described the way in which specific features of Algol were influenced by logic.

This is not the only story that could be told, of course. Some languages, and in particular those intended for data processing applications, such as FLOW-MATIC and Cobol, emphasized instead the extent to which programming notations could be made to resemble natural languages, as a way of making them easier to read, and also, perhaps, to write. A third approach, originating in the needs of artificial intelligence, also made use of 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 inspired the developments in programming languages development and theory in the next decade. These developments are the subject of the following chapters.

Chapter 9
The Algol Research Programme

Compared to some of the 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".[1]

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 was made explicit, but not answered, in a detailed history of the development of Algol 60 published by Robert Bemer in 1969; in his introduction Bemer quoted a comment made by Andrey 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".[2]

This chapter suggests an answer to this question, arguing that what changed the face of programming was not simply the Algol 60 language, but rather a coherent and comprehensive research programme within which the Algol 60 report had the status of a paradigmatic achievement, in the sense defined by the historian of science Thomas Kuhn. This research programme established the first theoretical framework for studying not only the design of programming languages, but also the process of software development, the subject of the next chapter.

[1] Wexelblat (1981), p. xviii.

[2] Bemer (1969), p. 151.

M. Priestley, *A Science of Operations*, History of Computing,
DOI 10.1007/978-1-84882-555-0_9, © Springer-Verlag London Limited 2011

9.1 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".[3] In particular, the language acted as a catalyst for the formation of a number of new groups and institutional initiatives. There were earlier examples of social groups forming around particular computing technologies, such as the SHARE group formed by users of the IBM 704 computer as a vehicle to enable the sharing of code examples and working practices.[4] 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.

Inventing Institutions Prior to Algol 58, most programming languages had been developed by small groups, using processes in which the design of the language and the implementation of a compiler proceeded more or less in parallel. By contrast, the Algol 58 report was produced by a committee representing many different groups on two continents, and serious attempts to write compilers did not begin until after the publication of the language specification. This open structure meant that compiler writers were no longer able to resolve difficulties locally and 'make the language up as they went along', as Backus claimed had been done with Fortran.

In 1959, a number of academic and commercial computing centres in Europe began to investigate the possible use of Algol. In February, representatives from these centres met in Copenhagen and agreed to start a newsletter, the 'ALGOL-Bulletin' (AB), to support continued collaboration and communication. The first bulletin was circulated by its editor, Peter Naur from the Regnecentralen in Copenhagen, in March 1959.[5]

As well as being a medium for the exchange of information, the AB aimed to be an instrument for "making agreements on the policy to be followed in developing ALGOL generator programs";[6] if programs were to be shared, for example, it was important that each group implemented the same subset of the language. A detailed procedure was outlined for voting on and arriving at such agreements. The AB was not simply a means of sharing information, therefore, but also reflected a desire to create some kind of institutional structure within which the future development of the Algol language could be managed.

The activities of the all-European AB group had, of course, to fit into the wider context of the US-European collaboration within which Algol had been developed.

---

[3] See Bemer (1969), for example, and also the material on Algol collected in Wexelblat (1981).

[4] Akera (2001).

[5] Naur (1959). The bulletin had a rather bureaucratic and formal flavour. It was specified, for example, that each contribution to future issues should begin with a 'type declaration', stating whether it was, among other possibilities, a question, an answer, a communication or a request.

[6] Naur (1959), p. 4.

Representatives from both continents met at the UNESCO conference on data processing held in Paris in July 1959, and a joint committee was formed to consider proposed changes and extensions to the language.[7] A further all-European meeting was held in Paris in November, prior to the joint meeting in January 1960 at which the Algol 60 report was finalized. During 1959, therefore, the main role of the AB was as a forum where interested Europeans could conduct an extensive and wide-ranging discussion on the definition of the language and its implementation, and many of the issues discussed were addressed in Algol 60. The *Communications of the ACM*, although formally independent of the Algol development effort, played a similar role in the US.

It soon became apparent, however, that the Algol 60 report was not the last word, and that ongoing work would be required to address remaining ambiguities in and proposed changes to the language. In August 1960, the Americans formed an 'Algol Maintenance Group' to consider these issues, and suggested that a parallel group be set up in Europe. Fearing that the existence of two groups would lead to two diverging dialects of Algol, however, several members of the AB group applied for membership in the Maintenance Group rather than forming a parallel European group. Meanwhile, discussions continued in the AB until 1962, when a meeting in Rome produced a revised version of the Algol 60 report.

In parallel with this Algol-specific activity, more general institutional support for work on programming languages began to emerge. This was initially supported by the International Federation for Information Processing (IFIP), who in 1962 set up a technical committee on programming languages (TC-2). TC-2 had 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". At the same time, a sub-committee of TC-2, known as "Working group 2.1 (WG2.1)", was established to "assume responsibility for the development, specification and refinement of ALGOL".[8] This strongly suggests that Algol had played a significant part in focusing interest on a more systematic approach to the study of programming languages.

To a large extent, the 'politico-social' history of Algol can be read as a process of discovering what kind of institutional authority was required to create and maintain a programming language standard in the face of demands from implementers and users to create subsets and dialects. It appeared that informal groups like the AB group were not able to maintain and enforce a common language definition, and that by 1962 such informal groupings had been made obsolete by the creation of WG2.1. Among other factors, these developments led to the AB temporarily ceasing publication, as Naur explained in 1962:

> in encouraging the ALGOL community to make use of the ALGOL Bulletin for expressing their views the editor must feel convinced that the views contained therein will indeed be taken properly into account when official action is taken. The developments mentioned

[7] Reported in AB-4, 13 August 1959.

[8] Bemer (1969), pp. 197–198.

above and the meeting in Rome have annihilated this conviction. Consequently the ALGOL Bulletin must cease to exist.[9]

Scientific Publications The publication of the Algol 60 report was followed by a flurry of journal articles.[10] Three topics were 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 publication of the Algol definition preceded any implementation of the language, 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.[11]

At the same time, computing conferences and symposia began to take a greater interest in issues related to programming languages. The *Information Processing 1962* conference, organized by IFIP, was described in the magazine *Datamation* as being "[i]n virtually all respects … a programming-oriented conference".[12] This assertion was based on a perception of the interest shown in sessions on various topics, however, rather than the comprehensive range of papers appearing in the conference proceedings.[13]

More specialized events were also organized. In March 1962, a symposium on "Symbolic Languages in Data Processing" was held in Rome at the International Computation Centre,[14] and in 1964 the IFIP committee on programming languages organized a working conference on "Formal Language Description Languages".[15] As the name of this later event indicates, attention was paid not only to the details of the programming languages themselves, but also to 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, Edsger 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'".[16]

Paradigm Formation In the years immediately following 1960, then, the study of programming languages became well enough established to attract considerable

[9] Naur (1962), p. 3. The Bulletin was, however, revived in 1964 and continued publication, albeit intermittently at times, until 1988.

[10] Bemer (1969), pp. 219–234, gives a comprehensive list.

[11] Floyd (1964).

[12] Quoted in Bemer (1969), p. 202.

[13] Popplewell (1963).

[14] International Computation Centre (1962).

[15] Steel (1966).

[16] Dijkstra (1962b), p. 537.

institutional support and recognition. The events described in this section support the assertion made by many computer scientists, both at the time and subsequently, that the Algol development, and in particular the publication of the Algol 60 report, played a crucial part in this process.

In his famous book, *The Structure of Scientific Revolutions*, the philosopher and historian of science Thomas Kuhn introduced the notion of a *paradigm*, initially defined as an exemplary scientific 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 resolve".[17] Both these properties were certainly true of Algol, and it seems natural to describe the Algol 60 report in particular as a paradigmatic achievement in the field of programming languages.

9.2 Normal Science in the Algol Research Programme

In Kuhn's account, the acquisition of a paradigm marks the coming of age of a scientific field, and enables a transition to a period of '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.[18]

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". 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". 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".[19]

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

[17] Kuhn (1962).

[18] McCarthy (1963a), p. 69.

[19] McCarthy (1962), p. 21, McCarthy (1965), p. 219.

to give the right answers to an equivalent form guaranteed to give the same answers, but which has other advantages such as speed".[20]

A prerequisite for the development of a theory of the desired type was that there should exist convenient notations for describing the "entities with which computer science deals", namely "problems, procedures, data spaces, programs representing procedures in particular programming languages, and computers".[21] 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".[22]

As a illustration of what he meant, McCarthy described a formalism similar to Lisp, based on the λ-calculus, which not only included ways of recursively defining functions computable on the basis of a given set of primitive functions, but also 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 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 tradition, and in the following chapter the project of replacing debugging with proof is examined in more depth.

9.3 The Description of Programming Languages

As described in Chap. 4, a distinction between syntax and semantics in logic had been articulated by Tarski and Carnap in the 1930s. This distinction was drawn upon by Backus in his treatment of Algol 58,[23] and general 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 rather artificial English. The success of this approach to the definition of Algol provided a powerful motivation for further exploration of the application of these metalogical notions to programming languages.

The link between this application and 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" produced a glossary of fundamental terms in the area. Gorn made use of Morris's threefold division between pragmatics, semantics and syntax, but gave revised definitions of

[20] McCarthy (1961), p. 225.

[21] McCarthy (1962), p. 22.

[22] McCarthy (1961), p. 225.

[23] Backus (1959).

the key terms which reflected the fact that programming languages were intended to be processed by machine, rather than interpreted by human agents. Thus whereas Morris had characterized pragmatics as being concerned with "the relation of signs to interpreters",[24] Gorn glossed this as follows: "We will 'interpret' the words *user* and *interpreter* to have a mechanical sense, i.e. to mean 'processor"'.[25]

The application of Morris's scheme to programming languages provides a clear example of 'bridging', to use Andrew Pickering's terminology for describing the process of conceptual innovation.[26] Bridging refers to a recognition of similarities between two domains, in this case logic and programming languages, which makes it reasonable to consider using the existing domain as a guide for exploration of the new. According to Pickering, bridging is followed by a phase of 'transcription', where results from the older domain are relatively straightforwardly reinterpreted to give insight into the new. Much of the theoretical work in programming languages in the 1960s can be seen as an attempt at transcription from logic in this sense. However, the differences between programming languages and logic meant that this process 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 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 would 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". 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".[27]

Strachey was here describing a situation where the syntax of a programming language permitted 'nonsense', or non-terminating, programs to be written, but where

[24] Morris (1938), p. 6.

[25] Gorn (1962), Gorn (1961), p. 337.

[26] Pickering (1995).

[27] Strachey (1962), pp. 102–103.

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 languages used in programming and logic, and suggested that the work of transcription might not be completely 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 normal regulatory function. As van Wijngaarden put it, "[t]he main idea in constructing a general language, I think, is that the language should not be burdened by syntactical rules which define meaningful texts".[28]

Dijkstra later gave an account of the philosophy of language underlying this view, in which meaning was viewed as being inextricable from the act of communication, in the sense that "the reaction of my listener determines what my utterances mean". 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 described 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", the point being that in the case of programming languages we can tell from the text alone what process will be executed.[29]

It followed from this account, according to Dijkstra, that "syntax does not have a defining function". 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 might 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 the so-called "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

[28] van Wijngaarden (1962), p. 409.

[29] Dijkstra (1963), pp. 33–34.

object language in terms of the target language".[30] 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.[31]

This work formed an approach to the question of how to define the semantics of programming languages which was rooted in the very practical problem of writing compilers for the large number of new high-level languages that were emerging. The hope was that a single 'compiler-compiler' could be written, a program which would be able to 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. Similarly, the view that semantics consisted primarily in translation also came under direct attack: in 1963, at an ACM workshop on mechanical languages for example, McCarthy stated bluntly that "to describe semantics by means of a translation rule is an incorrect thing to do",[32] and similar views were expressed by Ken Iverson and Maurice Wilkes.

An alternative approach was related to the existing practice of specifying the meaning of programming codes by describing the real or virtual machine on which the code ran. This technique was applied to the new programming languages of the 1960s, but with a emphasis on giving a formal definition of the interpreting machine. McCarthy had hinted at this, 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",[33] and both van Wijngaarden and Dijkstra described abstract machines which were, according to Dijkstra, "suitable means for the formalization of the semantic definition of an algebraic language".[34] McCarthy

[30] Irons (1961), p. 51.

[31] Feldman (1966), p. 3.

[32] McCarthy (1963b), p. 134.

[33] McCarthy (1961), p. 225.

[34] Dijkstra (1962a), van Wijngaarden (1962).

had himself explained the meaning of Lisp programs by giving the definition of the 'apply' function which evaluated them. 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 developed further during the 1960s. In 1963, Gilmore described a "Lisp-like" language, writing 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",[35] 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".[36] This general approach was later adopted in a project to define the semantics of the new 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".[37] By the end of the decade, this general strategy to programming language semantics was being referred to as the *operational* approach.[38]

However, in 1962 McCarthy had proposed a more abstract approach in which the details of the computation performed by a program would drop out of the semantic account altogether, leaving just the relationship between the initial data presented to the program and the results it 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 $\xi' = \text{algol}(\pi, \xi)$ which gives the value ξ' of the state vector after the ALGOL program π has stopped.[39]

In a later paper, this approach was applied to a small subset of Algol, and McCarthy explained that the state vector would include "the value currently assigned to each variable and also the statement number about to be executed". In this same paper, McCarthy asserted that his approach to semantics "corresponds to the notions of Tarski et al., that are current in mathematical logic".[40]

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, a computation does not finish, and so there is no final state. The semantic function is therefore undefined for such programs. For some, this was an objection to McCarthy's semantic account: because it did not include any account of the actions performed by programs, it could not distinguish between,

[35] Gilmore (1963), p. 73.

[36] Elgot and Robinson (1964), p. 365.

[37] Lucas and Walk (1969), p. 105.

[38] Lucas (1972), Wegner (1972).

[39] McCarthy (1962), p. 27.

[40] McCarthy (1964), pp. 3, 6.

for example, two non-terminating programs which were nevertheless performing very different computations.[41]

In giving this account, McCarthy appears to have been trying to align the theory of programming languages with established recursive function theory, much as he had done with the definition of languages in the case 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, in which a program text would be translated into a single expression defining the function computed by the program.

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. This was soon followed by an explicit proposal for a programming language based on AEs.[42] Landin gave an operational definition of the semantics of this language by describing a machine which would evaluate AEs. For Algol 60, however, he 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.[43]

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".[44]

Achieving 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,[45] 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.

[41] McCarthy (1964), p. 10.

[42] Landin (1964, 1966).

[43] Landin (1965a, 1965b).

[44] Strachey (1964), p. 201.

[45] Barron et al. (1963).

Strachey developed his ideas in the following years into a wide-ranging theory that became known as *denotational* semantics.[46] 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.[47] Following McCarthy, he viewed the textual details of a language's syntax as being irrelevant, and worked instead with 'abstract syntax' which was 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 described below.

The application of the metalogical distinction between syntax and semantics to programming languages therefore resulted in the early 1960s in the development of at least three distinct approaches to the problem of giving their semantics. The translation-based account was from the beginning associated with the practical task of writing compilers; however, despite occasional proposals "to define languages by their compilers",[48] it became less frequently referred to as a semantic account, compared with the operational and denotational techniques, in part because of the "inscrutable" nature of the semantic description that a compiler embodied.[49]

Pragmatics Compared with syntax and semantics, the concept of pragmatics put forward by semioticians was rather underdeveloped in mathematical logic, and it 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":[50] 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. 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".[51]

[46] Tennent (1976).

[47] Strachey (1967).

[48] Garwick (1964).

[49] Rochester and Goldfinger (1964).

[50] ACM (1966).

[51] Zemanek (1966).

One widely discussed aspect of pragmatics concerned the question of whether different programming languages were required for different application areas. It was common to draw distinctions between, for example, '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.

9.4 Different Philosophies of Programming Language Design

Investigation into new programming language concepts, and the development of new languages, continued throughout the 1960s. However, different assumptions were made by different groups, and these disagreements 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".[52]

NPL was intended to be a language that would be easy to program in. To this end, its designers believed that it should not be necessary to know every detail of the language before making productive use of it, and that the language specification should not place obstacles in the way of programmers. Two specific design criteria suggested a route to this goal.[53] First, "*Anything goes*. If a particular combination of symbols has a reasonably sensible meaning, that meaning will be made official". Secondly, a form of '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". 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 that would be usable in different application areas and the concern showed for the experience of programmers working with the language suggest that the designers of NPL were strongly influenced by pragmatic considerations, as defined above. In contrast with this, and pursuing McCarthy's overall goal of eliminating debugging, the Algol research programme placed more

[52] Radin and Rogoway (1965), p. 9.

[53] Radin and Rogoway (1965), pp. 9–10.

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.[54] 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. Many programs function as potential sources of knowledge, whether concerning the primality of integers or the size of a gas bill, and Dijkstra asked under 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 he 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.[55]

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).

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 the nature of a successor language to Algol 60. The dominant tendency within the group was in

[54] Dijkstra (1965).

[55] Dijkstra (1965), p. 5.

favour of a form of generalization known as 'orthogonality', where "all possible combinations of two or more independent concepts were allowed". Given even a relatively small number of basic concepts, this approach would quickly lead via a combinatorial explosion to a large and complex language, as proved the case when this approach was used as the basis for the language Algol 68. The alternative to orthogonality was "only to insert those possibilities in the language as were seen fit for some purpose",[56] and this was adopted in a paper by Tony Hoare and Niklaus Wirth which 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.[57]

The remainder of this chapter describes how this principle was applied 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.

9.5 Logic and the Design of Control Structures

The debate over program language design became particularly heated in connection with the design of control structures, and in particular a controversy about the role of jumps in programming. It had been always been recognized, of course, that the conditional execution of code and the repeated execution of blocks of code were essential coding patterns, and in machine code these were implemented using jump instructions to navigate around a program. In higher-level programming languages, unconditional jumps were still available, implemented by special statements usually known as 'go to' statements. Some apparently more sophisticated statements, such as the IF statement of Fortran I, were really little more than convenient ways of writing multiple jumps.

In addition to jump statements, programming languages gradually introduced specialized statements, later known as control structures, which encapsulated these common patterns of control. For example, Fortran's DO statement provided a basic iteration facility, and the conditional expressions of Lisp made it explicit that certain pieces of code would only be executed in certain circumstances. Algol 60 included both a *for* statement for writing loops, and an *if* statement for conditional execution.

A number of people had commented on the anticipated benefits of specialized notation for describing the flow of control. For example, Hamblin, recognizing that

[56] van der Poel (1986).

[57] Wirth and Hoare (1966), p. 414.

describing transfers of control presented a problem for his proposed programming notation, wrote that "there are some hopes that control transfer may be unnecessary in other cases if a sufficiently flexible system of conditional instructions can be found", and McCarthy said of the techniques used in traditional notations for recursive functions that "controlling the flow in this way is less natural than using conditional expressions which control the flow directly".[58] 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.[59]

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

Dijkstra's comments about the benefits of programming without jumps raised the question of whether it was in fact always possible to eliminate go to statements. In 1966 Corrado Böhm and Giuseppe Jacopini published a paper about normal forms in an artificial flowchart language; their results were widely interpreted as showing that it was possible to write a program for any algorithm using only conditional and iterative control structures, and hence showing the dispensability of the go to statement.[60]

The theoretical possibility of doing without go to statements did not necessarily resolve Dijkstra's requirement for lucidity, however. As he later pointed out, there was no reason in principle to suppose that a program without go to statements that was produced by means of Böhm and Jacopini's method would be significantly more comprehensible or convincing than one that used go to 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).[61]

Dijkstra gave two arguments for this recommendation. The first was related to a comment made by Wirth and Hoare in which they claimed that "[t]he notational

[58] Hamblin (1957), pp. 138–139, McCarthy (1961), p. 237.

[59] Dijkstra (1965), p. 216.

[60] Böhm and Jacopini (1966).

[61] Dijkstra (1968b).

structure of programs expressed in the language should correspond closely with the dynamic structure of the processes they describe". 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".[62]

This argument is related to the desire to give a compositional style of semantic explanation for programming languages, with the additional demand that languages should not contain statements which do not permit of such an explanation. Consider, for example, a control structure, such as a *for* statement, 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 solely by the syntax of the *for* 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 the use of a go to 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 go to statement. The use of go to 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 the interest in the relationship between programs and proof that had been generated by papers by Floyd and Hoare.[63] This work is considered in detail in the following chapter; in 1968 Dijkstra made relatively informal use of it, simply pointing out that in order 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".[64] 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 would 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 process could be given by simply giving the number of the currently executing statement. The numbering scheme would need to be more complex in order to cope with iteration constructs and subroutine calls, and Dijkstra demonstrated how this could be achieved. If a program included go to statements, however, the graph of potential paths through the program could become arbitrarily complex, and there would be no possibility of identifying the state of an executing program by any number of textual indices. Dijkstra later summarized this argument 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

[62] Wirth and Hoare (1966), p. 414, Dijkstra (1968b), p. 147.

[63] Floyd (1967), Hoare (1968).

[64] Dijkstra (1968b), p. 147.

> and repetitive clauses and procedure calls, rather than by statements transferring control to labelled points.[65]

The elimination of the go to statement did not, however, settle the question of what control structures should be provided by a language. On the basis of both practical experience and Böhm and Jacopini's theoretical result, it was accepted that it would be sufficient if programmers the means to express the sequencing and conditional and repeated execution of basic statements, but there were many ways to do this, and a wide variety of control statements were defined in contemporary language proposals.

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",[66] 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.[67]

For Dijkstra, a more principled way to evaluate different control structures was to consider their proof-related properties. The choice between control structures should be made by considering the ease with which they enabled convincing arguments for the correctness of programs to be made, 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 question 'for what program structures can we give correctness proofs without undue labour, even if the programs get large?'"[68] The same point was 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.[69]

Hoare confirmed this point, noting that "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".[70]

In order to think 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

[65] Dijkstra (1969b), pp. 85–86.

[66] Dijkstra (1965).

[67] Wirth and Hoare (1966), p. 415.

[68] Dijkstra (1969b), p. 85.

[69] Dijkstra (1972), p. 20.

[70] Hoare (1972a), p. 336.

to show that execution of the following statements will preserve the truth of the relation $0 \leq r < dd$:

$$dd := dd/2;$$
$$\textbf{if } dd \leq r \textbf{ do } r := r - dd.$$

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.[71]

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:

$$d := D;$$
$$\textbf{while non } \text{prop}(d) \textbf{ do } d := f(d).$$

He also provided a proof that the loop terminates after the kth iteration.[72]

Dijkstra also gave an example of how enumerative and inductive reasoning could be used to prove the correctness of a small program.[73] 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. Nevertheless Dijkstra was "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 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.[74]

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

[71] Dijkstra (1972), p. 7.

[72] Dijkstra (1972), p. 8.

[73] Dijkstra (1972), pp. 12–14.

[74] Dijkstra (1972), p. 10.

syntactic and semantic structure of program texts prompted a move away from the go to statement to more specialized control structures, the specific forms of which were motivated by a desire to make the correctness of programs accessible to logical reasoning, even if programmers made only informal and second-hand use of the logical results.

9.6 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, it was widely acknowledged that the so-called scientific languages such as Fortran and Algol 60 were 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 were 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 and Lisp.[75]

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 stored in a variety of formats, either textual or numeric. 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 three different approaches. Practical proposals made included a number of ad hoc suggestions to incorporate the features from one area into a language of a different type; it was, for example, suggested to add support for strings and lists to Algol 60, and some proposals for new languages, such as NPL, attempted to include features from all areas.[76]

[75] Newell and Tonge (1960), McCarthy (1960).

[76] Green et al. (1959), Radin and Rogoway (1965).

The general problem was summarized by Douglas Ross who was involved in the development 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.[77]

Ross's solution envisaged "problems as being composed of interconnected *n*-component elements of a general type".[78] 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".[79] 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":[80] 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 when they were needed as 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 proposal provided a way of satisfying the needs of 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*" [81] 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.

---

[77] Ross and Rodriguez (1963), p. 306.

[78] Ross (1961), p. 147.

[79] Ross and Rodriguez (1963), p. 306.

[80] Wirth and Hoare (1966), p. 416.

[81] Wirth and Hoare (1966), p. 417.

These proposals about records and record classes were incorporated into Wirth's new language, Pascal.[82] Pascal defined a number of *scalar* types, representing atomic data values such as numbers, characters, user-defined symbols and references (now called 'pointers'), 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 the theoretical work on data structures that had been carried out in parallel with the practical language developments.

An early 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 the 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".[83] 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".[84]

This approach was further developed by Hoare, who proposed that data types in programming languages could be understood as denoting sets of data values. Given a number of basic types, whose values were 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".[85] 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 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.[86] 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 impractical, however, for Pascal to implement fully Hoare's theoretical account. For example, the powerset structure was limited so that only powersets

[82] Wirth (1971b).

[83] McCarthy (1962), p. 21.

[84] McCarthy (1961), pp. 231–232.

[85] Hoare (1972b), p. 93.

[86] Wirth (1971b), p. 37.

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 defined to be 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 a central aspect of programming languages was profoundly influenced by theoretical considerations drawn from logic and set theory.

9.7 Modelling Data for Information Retrieval

At the beginning of the 1960s, programming systems in the two areas of scientific and data processing were to a large extent developing independently of each other. 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 them, and it was assumed that it would be a relatively straightforward 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".[87] 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.[88]

Cobol encapsulated a model where information 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.[89]

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.[90] 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.[91]

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

[87] Colilla and Sams (1962), p. 11.

[88] Sammet (1962).

[89] Lombardi (1960).

[90] Bosak et al. (1962).

[91] Bosak et al. (1962), p. 190.

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 applying mathematics and logic to 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".[92] 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 an 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 is usually achieved by defining properties such as 'employee payroll number' which are guaranteed to be distinct for each entity modelled by the system.

As described 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 representations of distinct entities which happened to share the same properties could easily coexist at different locations in the store. They would be distin-

[92] Bosak et al. (1962), p. 191.

guishable 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 reference values, that had no real-world analogue in 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 were held could be a significant economic asset and have a lifetime much longer than that of the programs which manipulated 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 them to take on a life of their 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.[93]

A number of different database models were put forward, but they all shared a number of characteristics. Firstly, databases were based on a model consisting of files and records, 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 them cannot in general use the address of a data item in memory as a way to access it. 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 might 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.

[93] Bachman (1973).

A prominent proposal 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 language.[94] Network databases were based on two primitive concepts, the file and the 'set'. Sets were the vehicles for representing relationships within a network database: all the records which were related in a particular way to a given record, such as the set of employees that work in a particular department, constituted 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", and a similar metaphor was used by others, such as Jay Earley who referred to "access paths" through data structures.[95] Programs could have various 'current locations' within a database, and by making use of commands embedded in a programming language could update the current location and thereby move from one data item to another. For example, suppose a program had to process all the employees working in a particular department. The relevant records would be physically linked in the database as part of a set; the program would record the current position within this set, and a programming operation was provided to move to the next item in the set. This could be repeated until every record in the set had been processed. Thus programs accessed network databases 'from the inside', as it were, a record at a time.

An alternative database model, the 'relational' model, was introduced by Ted Codd in 1970.[96] There were two significant differences between the relational and network models. 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 the sets in a network database, 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 had one primitive concept, the relation, which modelled both. One advantage claimed for this was that it kept the logical structure of the data 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 did not proceed by means of record-at-a-time navigation through the database. Rather, a number of high-level operations on relations were provided, the most significant of which was known as a 'join', an operation combining two relations into one. These operations were defined to work on whole relations, rather than on individual records, and to return new relations as their results. These resulting relations were virtual, and not stored

[94] CODASYL Data Base Task Group (1969).

[95] Earley (1971).

[96] Codd (1970).

physically in the database, but as data structures they were identical to the relations defined in the database schema. This meant that they could 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.

9.8 Conclusions

This chapter has discussed the use of logic and algebra in computer science in the 1960s in the related areas of programming language design and the development of theoretical models for databases. It has been argued that the publication of the Algol 60 report catalyzed the formation of a coherent research programme which aimed to use 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 application of logic was not simply a case of drawing straightforward consequences for programming languages from theoretical results in logic. Even the role of concepts as fundamental as syntax and semantics could be contested, and significant amounts of work were required to establish what they might mean in the context 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 scientific languages and in 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 based on higher-level operators which were described in terms of their overall effect. In the area of programming, this can be seen in the introduction of control structures which embodied common patterns of computation to replace the go to statement. In the database world, the relational model with its set of general algebraic operations would in the coming years become much more widespread than the network model with its reliance on navigation from record to record in the database, a procedure itself very reminiscent of a jump.

However, this transition was incompletely carried out in programming languages themselves. 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 Chap. 11; 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 10
The Logic of Correctness in Software Engineering

This chapter describes the approach taken by those working in the Algol research programme to the problem of how to improve the quality of software development and in particular to ensure that software systems met their users' expectations and were completed economically and on schedule. These concerns came to prominence in the mid-1960s in response to a perceived 'software crisis', and were extensively discussed at the well-known NATO conference in 1968 which brought the term 'software engineering' to prominence.[1]

In response to the goals set by McCarthy for the Algol programme, two major results emerged from this work. Firstly, a novel 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 put forward which, it was hoped, 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.

10.1 Checking Computations

The introduction of large-scale automatic computers to perform calculations which had hitherto been carried out by hand raised the question of how the correctness of the results produced could be guaranteed. What was meant by 'correctness' in this context changed as computing and programming technology evolved and brought different issues to prominence. The reliable functioning of the earliest machines,

[1] Naur and Randell (1969).

M. Priestley, *A Science of Operations*, History of Computing,
DOI 10.1007/978-1-84882-555-0_10, © Springer-Verlag London Limited 2011

at a purely mechanical or electronic level, could not be taken for granted, and the primary 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 the ASCC that "of paramount importance in the design of a sequence control tape, are the checks on the computation".[2]

Aiken and Hopper identified three distinct sources of error. Firstly, errors could be made in the mathematical formulation of the problem being solved. These did not differ in principle from the kinds of mistake that were 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 source of error was new, however, and was introduced by the processes involved in transferring the mathematical solution of a problem onto the computer. Aiken and Hopper simply put these down to human factors: "two major sources of human error, incorrect switch settings and incorrect plugging, are perhaps the most serious of all", because in cases where there was no feasible mathematical check on the final results of a computation, these errors could easily go undetected. They offered no methodological solution to avoiding such errors, however, beyond taking such obvious precautions as requiring "meticulous precision on the operator's part and careful checking of all manual operations".[3]

With the introduction of stored-program machines, the manual operations that were involved in setting up a machine to perform a calculation were significantly simplified: instead of rewiring parts of the machine, or setting arrays of switches, all that was required was to feed in a tape containing the instructions for the program. As a result, attention became increasingly focused on the design of the sequence of operations to be carried out. At the conference held in Cambridge in 1949, J.C.P. Miller discussed the errors that could arise from "[p]rogramming and coding the [mathematical] solution for the machine".[4]

At this time, 'programming' referred to the process of designing an algorithm to solve a problem and 'coding' to the process of translating the operations required by an algorithm into a particular machine code. The 'programming errors' identified by Miller therefore corresponded to the 'mathematical errors' of Aiken and Hopper, who had not themselves identified a separate category of coding 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". Only two years later, however, Maurice Wilkes and his colleagues reported that "such mistakes are much more difficult to avoid than might be expected", and similar comments were made by others.[5]

[2] Aiken and Hopper (1946), p. 525.

[3] Aiken and Hopper (1946), p. 525.

[4] Miller (1949).

[5] Wilkes et al. (1951), p. 38, see also Brooker et al. (1952).

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, or taking particular care over the execution of computations. A variety of techniques for preventing such errors were proposed and put into practice, including the inspection of program code 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 contained 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 a lot of emphasis was placed on the avoidance of error by uncovering mistakes before a program was executed, and there is little mention of the practice 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 was reported as having had "very little patience for an error-infested trial session",[6] and Wilkes referred to the amount of machine time that could be lost running erroneous programs.

At first, 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".[7] 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".[8]

10.2 Debugging and Testing

As the 1950s progressed, increasing practical experience of programming and of the problems involved in developing larger software systems led to the development of new techniques and approaches to the question of program correctness. According to Stanley Gill, 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". Initial optimism had given way to a belief that such errors were not "a temporary evil, due to lack of experience", and "some

[6] Bloch (1999), p. 97.

[7] Wilkes et al. (1951), p. 41.

[8] Miller (1949), emphasis in original.

attention has, therefore, been given to the problem of dealing with mistakes after the programme has be tried and found to fail".[9]

Gill did not feel that the significant problem was that of detecting that an error had occurred, at least for programs performing computations: "If its presence is not immediately apparent, it will be detected by the arithmetical checks which must be incorporated in every calculation". 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 more 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 automated and carried out by machine. Gill described the 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 in the programs run on it by means of techniques such as the use of "breakpoints" at which program execution could be interrupted, and an "automonitor" checking routine, among others.[10]

The development of large systems raised further unanticipated problems, and in 1956 Herbert Benington described the lessons that had been drawn from experience gained on the SAGE air defence system.[11] 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 activities of detecting errors, known as *testing*, and locating and correcting them 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

[9] Gill (1951).

[10] Diehm (1952).

[11] Benington (1956).

will be sampling only ... many sad experiences have shown that the program-testing effort is seldom adequate". Like debugging, the testing process could be automated by programs which performed "test instrumentation" using simulated live inputs.

As well as increasing productivity, it was widely expected that the development of automatic programming and the increased use of pseudocodes would lead to a reduction in 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", 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".[12] This expectation turned out to be overly optimistic, however: Robert Bemer later 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.[13]

The use of pseudocodes raised the question of how best to carry out debugging: at first, debugging efforts were directed towards the machine code generated from the pseudocode, but it was recognized that it would be more convenient to debug a program written in a symbolic code by examining the pseudocode itself rather than the machine code generated from it. Charles Katz discussed the issues raised by this proposal; 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".[14] At the end of the decade, Gill suggested 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".[15]

As hardware reliability increased, then, coding and programming became widely recognized to be the most serious sources of errors in computer programs, despite repeated expressions of optimism that improvements in the design of pseudocodes 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.

10.3 Correctness Proofs

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

[12] Diehm (1952), p. 19, Gill (1953), p. 291.

[13] Bemer (1984).

[14] Katz (1957), p. 21.

[15] Gill (1959).

program. As McCarthy had 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".[16] 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 Benington had been pointed out were articulated further, 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",[17] and does not permit us to conclude that the machine will work correctly when presented with other programs. This is an application of a familiar argument against an inductive approach to gaining knowledge, the point being that from a finite set of observations nothing can be inferred about future or unobserved events. Applied to software, the point is that 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 in a famous epigram, "Program testing can be used to show the presence of bugs, but never to show their absence!"[18]

In 1966, Peter Naur described it as "deplorable … that the regular use of proof procedures … is unknown to the large majority of programmers". 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". 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", and finally prove that the algorithm had the required properties.[19]

One difficulty facing attempts to construct proofs of programs arose from the semantic difference between the assertions describing the transformation carried out by an program, and the imperative code describing the algorithm to 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".[20] 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

[16] McCarthy (1962), p. 22.

[17] Dijkstra (1962b), p. 537.

[18] Dijkstra (1969b), p. 85.

[19] Naur (1966), pp. 310–311.

[20] Naur (1966), p. 312.

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.

Robert Floyd took a similar line, proposing "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".[21] The correctness of a program could then be obtained "by an induction on the number of commands executed", 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 further made an explicit connection between proof and semantics, referring to his technique as a way of "assigning meanings to programs".

Even before 1966, the technique of using propositions to make assertions about properties of program executions had quite a long history. For example, when reminiscing about programming the ASCC in 1944, Richard Bloch described his approach 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".[22]

Similarly, while developing their technique of using flow diagrams for program development, von Neumann and Goldstine 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".[23] 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, asking "[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".[24] However, this suggestive early work was not followed up, and it was only in the context provided by the Algol research programme in the mid-1960s that the use of assertions was systematically investigated and serious attempts made to apply it to programming practice.

The use of assertions in proofs of the correctness of programs relied on a clear understanding of the effect of the execution of individual statements. Naur had 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 assign-

[21] Floyd (1967), p. 19.

[22] Bloch (1999), p. 94.

[23] Goldstine and von Neumann (1947), p. 92.

[24] Turing (1949).

ment then makes it again true to say that $A[r]$ is the greatest".[25] The formalization of such arguments, however, required a definition of the statements of a programming language 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".[26] 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 began by making 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".[27]

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". 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". In his other examples, Hoare used the same control statements that Dijkstra had identified as giving rise to particularly simple patterns of proof. Thus for the sequence of two consecutive statements he gave the rule

$$\text{If } \vdash P\{Q_1\}R_1 \quad \text{and} \quad \vdash R_1\{Q_2\}P, \quad \text{then } \vdash P\{Q_1; Q_2\}R,$$

and for iteration the following rule:

$$\text{If } \vdash P \wedge B\{S\}P, \quad \text{then } \vdash P\{\textbf{while } B \textbf{ do } S\}\neg B \wedge P.$$

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 paper thus provided an existence proof, showing in outline at least how the project of embedding at least some program language constructs in the familiar logic of propositions could be completed. In theory, this would enable proofs about algorithms and programs to be carried out using the machinery of formal logic, though Hoare did point out that such proofs were "excessively tedious" and that the development of practical techniques for program proofs would be needed if the method was to become widely used.

---

[25] Naur (1966), p. 324.

[26] Floyd (1967), p. 19.

[27] Hoare (1969), p. 576.

10.4 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".[28]

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".[29] Rather than proving the correctness of an existing program, Dijkstra tackled the problem the other way round, demonstrating methods to derive 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 were satisfied at the appropriate times. The style of the presentation 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 put forward an approach which would combine the earlier work on assertions and correctness with the constructive approach suggested by Dijkstra.[30] He proposed to identify the 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

[28] McCarthy (1961), p. 225.

[29] Dijkstra (1968a).

[30] Naur (1969).

cluster was a sequence of program statements which would always be performed as a whole and whose effect could be characterized by assertions. The correctness of the final program could then be checked by examining the relationship between the assertions defining the action clusters.

Hoare then took this one step further by combining the ideas of constructive development with the formal logic of programs, giving as an example an account of the development of an algorithm to perform certain manipulations on an array of data.[31] He began by giving an informal explanation of the algorithm, but the development of a corresponding program was carried out completely formally.

The formal process started by providing 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 the approaches of 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.[32]

By iterating this procedure, Hoare could 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 this method 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, some aspects of the program 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"; a similar approach was adopted, though in a less formal manner, by Niklaus Wirth.[33] For Wirth, "[i]n each step, one or several instructions of the given program are decomposed into

[31] Hoare (1971).

[32] Hoare (1971), p. 41.

[33] Hoare (1971), p. 45, Wirth (1971a).

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, such as the introduction of a new variable. 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, however, 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.[34]

10.5 Specifications and Correctness

In parallel to the development of constructive methods, the concept of correctness underwent a change from being thought of as a property of individual programs, to being considered as a meaning-preserving relationship between two programs which performed the same computation, or between a specification and a program which implemented the stated functionality.[35] Program specifications were still only stated informally, however, 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; it was perhaps inevitable that ways of removing these remaining traces of informality would be investigated.

In his attempt to produce a completely formalized program derivation, Hoare had stated the requirements for the example program by defining predicates which stated

[34] Floyd (1971).

[35] Programs had always been expected to do what their specifications stated, or course. The point is that theoretical accounts of correctness were increasingly phrased in terns of a program "meeting its specification" rather than "being correct".

the assumptions made about the data at the start of the process and also the properties that it was required to have at the end. This particular account of what a specification was tied in nicely with the suggestion made by McCarthy in 1962 that the meaning of a program should be considered to be its effect on the state vector, which included the program variables, and quickly became widespread. For example, Manna and Waldinger suggested using automatic theorem provers to automate the process of program development. They formalized the "specifications for the program to be written" as pairs of predicates, an "*input predicate* $\phi(\bar{x})$ and an *output predicate* $\psi(\bar{x}, \bar{z})$" and claimed that "[i]n order to construct such a program, we prove the theorem $(\forall \bar{x})[\phi(\bar{x}) \supset (\exists \bar{z})\psi(\bar{x}, \bar{z})]$".[36]

The relationship between programs and specifications had up to this point been assumed to be implicitly obvious, but it was soon given a more explicit treatment. Barbara Liskov and Stephen 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". 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".[37] While recognizing that the intended 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". 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.[38]

Another effect was an increase in the importance of specifications: if correctness amounted to correctness with respect to a specification, a program could only be said to be correct insofar as its specification was clearly stated and understood. If that correctness was to be provable, furthermore, 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 further consequence was that the analogy between software development and logical deduction was enriched. A formal specification was sometimes thought of as defining a set of axioms or postulates; from the specification other expressions were deduced, by correctness preserving refinement steps, culminating in the production

[36] Manna and Waldinger (1971), p. 152.

[37] Liskov and Zilles (1975), pp. 7–8.

[38] Manna and Waldinger (1978), p. 201.

of conclusions in the form of 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 therefore became viewed as analogous 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 were viewed as postulates did not imply that they were immune from revision, of course. A program could be correct with respect to a specification that is completely inadequate from the users' point of view. The activity of specification revision, however, played no role in the quasi-deductive model: the overall development process was split between an initial phase in which a specification was created, and a subsequent phase of refinement and implementation based on the assumption that an adequate specification existed.

This understanding of the process of software development was widely adopted, even when development was not being carried out in a formal manner. Very many '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 they were expressed in a mixture of informal textual and graphical notations, preserved the essence of the quasi-deductive approach, namely a process of refinement leading from a specification to a conforming implementation.

10.6 Structured Programming

In the early 1970s, many of the concerns of the Algol research programme moved from the research community to the practical world of commercial and industrial software development. They were widely thought to represent a new approach to the problems of programming, one which became widely referred to as 'structured programming'. This section examines this process, and the different ways in which the term was understood. The term 'structured programming' appears to have been first used by Dijkstra, who in August 1969 wrote some widely circulated 'Notes on structured programming', and in the same year presented a working paper titled simply 'Structured programming' at the NATO conference on software engineering techniques.

For Dijkstra, the central issue was how to be assured of the correctness of what he called "intrinsically large programs", 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 made for the use of specific control structures rather than jumps. Dijkstra also emphasized the use of abstract data structures in the program development process: using a curious metaphor, he described a program as "an ordered set of pearls, a 'necklace'", where each pearl

represented a program module written in terms of the facilities provided by the module below it in the string.[39]

The notion of 'pearls' derived, like the idea of constructive programming, from Dijkstra's experience in designing and writing a multiprogramming system.[40] 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".[41]

The 'Notes on structured programming' contained a more leisurely treatment 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" and that as a result, they could be treated as a single indivisible action in a sequential program.[42] 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 the book *Structured Programming*, which also contained contributions from Tony Hoare and Ole-Johan Dahl. An essay by Hoare on data structuring argued that set theory could be used to define a range of data structures, and an joint contribution on 'hierarchical program structures' by Hoare and Dahl described how programs in the Simula 67 programming language were structured. Simula 67 is considered in more detail in the following chapter; its relevance in the context of structured programming was the claim 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 together with the constructive method by which such programs could be produced, and finally a general scheme for program modules. This rich collection of ideas provided a great deal of 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". However, 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

[39] Dijkstra (1969a), p. 222, Dijkstra (1969b), p. 225.

[40] Dijkstra (1968c).

[41] Dijkstra (1969b), p. 255.

[42] Dijkstra (1972), p. 19.

apply formal methods".[43] Henry Ledgard later concluded in a response to this paper that, "[t]he method used ... is strictly speaking not really 'structured programming', at least as conceived by Dijkstra"; the reason given for this was that Henderson and Snowdon had not "*formalized* and debugged each of the levels".[44] To add to the confusion, Ledgard, despite claiming to make a case for structured programming, defined his own methodology which not only combined the ideas of structured and top-down programming and stepwise refinement, but also adopted a programming style which made use of the go to statement.

In 1973 Barbara Liskov returned to Dijkstra's emphasis on correctness when she characterized structured programming as "a programming discipline intended to support the production of correct, understandable programs which are easy to modify and maintain".[45] 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, she went further, saying that "[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".[46] Liskov downplayed the significance of proof, however, 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 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.[47] 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. Inspired by the make-up of surgical teams, a 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.

As well as adopting a new style of project management, the team used a top-down approach to design and implementation in conjunction with a version of structured programming. This was characterized in primarily syntactic terms 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", namely sequence, if-then-else statements and do-while loops.

---

[43] Henderson and Snowdon (1972), pp. 38, 51.

[44] Ledgard (1974), p. 49.

[45] Liskov (1973), p. 5.

[46] Liskov (1973), p. 6.

[47] Baker (1972a, 1972b).

The New York Times project was highly successful, and it was claimed that "[s]tructured programming, and the organization and tools used to achieve it, were key factors in developing this kind of system". Even though it was admitted that the use of chief programmer teams was not an essential part of structured programming, the widespread interest in this project meant that 'structured programming' became understood to be a general approach to software projects and not simply a technical approach to the organization of programs.

In 1973 the ideas of structured programming were brought to a wider audience when the popular magazine *Datamation* published a special issue on the topic. An introductory article entitled 'Revolution in Programming' 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".[48] The articles in this issue, however, characterized structured programming in terms that had more in common with Baker's description of IBM's experience on the New York Times project than from Dijkstra's theoretical writings.

In more theoretical treatments, 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".[49] 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". 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".[50]

For a more practical audience, the key point about the adoption of a particular set of control structures was the ensuing increase in 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". 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.

[48] McCracken (1973).

[49] Wirth (1974), p. 252.

[50] Donaldson (1973).

There can be no doubt that structured programming made a significant and long-lasting contribution to programming language design and programming practice. Older languages, such as Fortran, which did not include the recommended control structures, were soon revised to include them, and they have been a constant part of all languages developed since. The controversy over the go to 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.[51]

In its original form, then, structured programming was closely bound up with the Algol research programme, and in particular with 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 could be immediately applied by programmers, such as rules about the indentation of code.[52]

In particular, it is noticeable that the positive 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". 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.

10.7 Proof and Testing

As noted above, one of the central goals of the research programme articulated by McCarthy in 1962 was to replace testing and debugging by proof, and the first half of this chapter 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 prophesied that "the practical advantages of program proving will eventually outweigh the difficulties, in view of the increasing costs of programming error".[53] 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",[54] as well as offering a much stronger guarantee of reliability.

[51] Stevens et al. (1974).

[52] McCracken (1973).

[53] Hoare (1969), p. 579.

[54] Hoare (1971), p. 39.

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 most important means 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 observed that "the problem of program correctness has turned out to be far less serious than predicted" and went on to suggest that the systematic application of traditional engineering techniques to the development of software was largely responsible for the observed increase in software reliability.[55]

This raises the question of the extent to which 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, made earlier, that 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, who modelled scientific and mathematical theories as deductive systems which identify 'basic statements' as the final conclusions drawn by a theory.[56] 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,

[55] Hoare (1996).

[56] Lakatos (1967).

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 earlier in this chapter that the overall view of software engineering that was taken by the Algol research programme was that it was Euclidean, in Lakatos' sense. The adoption of a fixed specification is equivalent to an injection of correctness at the top of the system. 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. It was expected that approaches involving testing and debugging would become obsolete once techniques had been developed to carry out such derivations effectively:

> 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.[57]

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.[58]

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.

Stuart Shapiro described the use of proof and testing on the same project as a pragmatic approach which employed two independent verification techniques to maximize the chances of producing a correct system.[59] One motivation for such an approach might be an acknowledgement of the possibility of errors even in a proof of a program. In 1976, Gerhart and Yelowitz published a list of errors that they had detected in published examples of formal program derivations. Although they were sympathetic to mathematical approaches to program development, their evidence pointed out the fallibility of mathematical proof. While accepting that the formal verification of programs could help ensure that a program was "substantially correct", they concluded that "we must simply learn to live with fallibility".[60]

[57] Dijkstra (1968a), p. 185.

[58] Mills (1976), p. 269.

[59] Shapiro (1997).

[60] Gerhart and Yelowitz (1976), p. 206.

This suggests that a mixture of the Euclidean and quasi-empirical approaches to software engineering could be used, in which the typical response to the detection of an error in testing would be to correct the faulty code. Supporters of a purely 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".[61]

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, 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.[62]

Despite the use of the quasi-empirical technique of testing, however, these mixed 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 often 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 in the literature of software engineering are rare, no doubt because of the prominence of Dijkstra's attack on this position. Ironically, a mixed position which included elements of an inductive approach was employed by Dijkstra in the development of 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".[63]

---

[61] Hoare (1996).

[62] Fetzer (1988).

[63] 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".[64] 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 In the scheme being elaborated here, a quasi-empirical account of software engineering would 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)",[65] and Dasgupta put forward 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".[66]

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 of software that was absolutely correct, but instead accepted the inherent fallibility of software. In 1971, Friedrich Bauer wrote in an overview of the 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".[67] It is noteworthy that Bauer refers not to the correctness of software, but to its reliability; unlike correctness, reliability is not considered in engineering to be 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".[68] 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.[69]

A further characteristic that we might expect to find in quasi-empirical software engineering is 'bold hypotheses, followed by dramatic refutations', as described in

[64] Goodenough and Gerhart (1975), p. 157.

[65] Fetzer (1988), p. 1062.

[66] Dasgupta (1991), p. 353.

[67] Bauer and Wössner (1972).

[68] Schick and Wolverton (1978), p. 105.

[69] Randell (2000).

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.[70]

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.[71]

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 don't *know* them in advance",[72] an argument based on the observation that the development process itself frequently changed, among other things, users' perceptions of their 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 a software engineering artefact—either specification or program—meets the actual requirements of its users became known as *validation*.[73]

Responses to this situation took the form of proposals for software development methods that would involve the user extensively throughout. Originally known by

[70] 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"; See Basili (1996), for example. 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.

[71] Ross (1968).

[72] McCracken and Jackson (1982), p. 31.

[73] Boehm (1984).

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.[74]

Such approaches do not take a quasi-deductive view of the software development process; 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 instead emerges in the course of a rather 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, and it has been taken up and refined in Pickering’s notion of the ‘mangle’.[75] It is beyond the scope of this book to explore further the connections between this work and evolutionary approaches to software development, however, though some interesting attempts to link software development with the ideas of post-modernism have been made.[76]

10.8 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.[77]

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 program maintenance. It was in this context that 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.[78]

---

[74] Larman and Basili (2003).

[75] Callon (1987), Pickering (1995).

[76] See Robinson et al. (1998), for example.

[77] Mills (1986).

[78] Boehm (1976).

Structured programming was frequently described as a 'revolution',[79] 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 schema in this case is not 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 historiographical model for the take-up of structured programming can be found in the traditional account according to which new ideas are 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 altered significantly 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 were central goals of researchers in the Algol paradigm, but presentations aimed at industry downplayed these aspects, emphasizing instead issues to do with project management, for example, even though these were not part of the theoretical notion of structured programming.

[79] See McCracken (1973), Knuth (1974), for example.

Chapter 11
The Unification of Data and Algorithms

In the 1970s it was common to describe programs as having two main aspects, namely the data structures that the program required and the algorithms used to manipulate that data. This point of view was clearly reflected in the title of Niklaus Wirth's well-known book *Data Structures + Algorithms = Programs*, and also in the design of Wirth's language, Pascal, in which control and data structures were defined largely independently of each other.[1]

Investigations into programming methodology, however, had made it clear that there was a close relationship between these two areas. A common technique in top-down refinement methods was to introduce new variables and data values that would allow a high-level algorithm or specification to be expressed in more detail. Along with the data structures, the necessary operations were defined in terms of these new data items so that higher-level code could be insulated from the lower-level details of data representation.

This approach became known as *data abstraction*, and it was a natural extension of the Algol research programme to investigate ways in which logic could be used to help to understand the properties of data types defined in this way. In the early 1970s, and number of languages were developed which provided direct support for data abstraction in various ways. These included the definition of modules which implemented in a fairly direct manner the 'pearls' of Dijkstra's necklace, a more abstract notion, of abstract data type, and a more comprehensive notion, of object-oriented programming.

This chapter examines in more detail the development of novel programming language support for data abstraction, and it suggests that a stable configuration of ideas, one that has profoundly influenced the design of programming languages up to the present day, was first achieved by the Smalltalk language. Although it drew heavily on work from the Algol research programme, the design of Smalltalk was also strongly influenced by ideas from completely different areas, and the chapter concludes by arguing that in various ways Smalltalk marks a limit to the influence of logic on programming language design.

[1] Wirth (1971b; 1976).

M. Priestley, *A Science of Operations*, History of Computing,
DOI 10.1007/978-1-84882-555-0_11, © Springer-Verlag London Limited 2011

11.1 Simulation Languages

A number of important ideas about the unification of data and algorithms emerged from the experience gained in writing programs to carry out simulations. This had always been an important application of digital computers, and as experience was gained in programming simulations, common features of these applications became codified in special notations and languages. These languages included SIMSCRIPT and GPSS and two slightly later languages, Simula and SOL.[2]

Simulations were used to model the behaviour of complex systems whose global properties could not be determined easily, or at all, through analytic means. The particular type of simulation suitable for programming on digital computers became known as *discrete event simulation*. The system to be simulated would be modelled as a collection of entities, and interactions between these entities were thought of as discrete or non-overlapping events, each of which could change the overall state of the system in some way.

A typical example given of this type of simulation was the flow of traffic through a network of streets. In this case the entities in the system might include vehicles, traffic signals and so on. A vehicle reaching a traffic signal would represent an event, and the system state would be affected by which way the vehicle turned. Simulation programs usually needed to take account of the time at which events occurred, and often also included a probabilistic element, to model the fact that 70% of vehicles turned left at a particular junction, for example.

The basic assumption behind languages like SOL and Simula was clearly stated by Knuth and McNeley:

> A complex system can be represented as a number of individual processes, each of which follows a *program* very much like a computer program.[3]

A simulation program therefore had to keep track of a potentially very large number of processes, each of which was capable of acting in some sense independently, and the interactions between them.

> Because of the necessary list processing, complex data structures and program sequencing demands, simulation programs are comparatively difficult to write in machine language or in ALGOL or FORTRAN. This alone calls for the introduction of simulation languages.[4]

A primary goal of the early simulation languages was to provide users with the concepts and notation needed to describe complex systems, and the generation of a simulation program was almost seen as secondary to this. The designer of GPSS, Geoffrey Gordon, later stated that "[w]hat the GPSS user has to do is to draw a block diagram to represent the system, using some very specific block types"; the blocks were then translated into textual statements for input to the code generator.[5]

---

[2] See Markowitz et al. (1963), Gordon (1961), Dahl and Nygaard (1965) and Knuth and McNeley (1964) for details of these languages.

[3] Knuth and McNeley (1964), p. 401.

[4] Dahl and Nygaard (1966), p. 671.

[5] Gordon (1981).

The designers of Simula made this point very explicit:

> Simulation is now a widely used tool for analysis of a variety of phenomena: nerve networks, communication systems, traffic flow, production systems, administrative systems, social systems, etc. ... still more important is the need for a set of basic concepts in terms of which it is possible to approach, understand and describe all the apparently very different phenomena listed above. A simulation language should be built around such a set of basic concepts and allow a formal description which may generate a computer program.[6]

Simulation languages, then, were not simply alternatives to algorithmic languages like Algol, but had an entirely different goal: their purpose was to describe not just computations, but whole systems. In this respect, they had a different significance from languages for other application areas, such as data processing languages like Cobol, whose goal was still to describe computations, albeit with different data and in a different style from the scientific languages.

Simula Of the simulation languages developed in the early 1960s, Simula had the most influence on later work on general-purpose languages. It was developed at the Norwegian Computing Centre by Kristen Nygaard and Ole-Johan Dahl, and had its roots in Nygaard's work in operational research. This work led to a recognition of "the necessity of using simulation, the need of concepts and a languages for system description, lack of tools for generating simulation programs".[7]

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, however, the two concepts of 'station' and 'customer' had been merged into a more general notion of 'process' which combined the data associated with the customers and the operations carried out by the 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.

The need to describe the active behaviour of processes meant that a simulation language had to contain at least some elements of an algorithmic language. Rather than creating a new notation, Dahl and Nygaard chose to base Simula on Algol 60; they did this in such a way that it was an extension of Algol, in the sense that any feature or structure defined by Algol 60 could be used in a Simula program.

Key to this was the Algol notion of a block. A Simula process was both a passive carrier of data and an active element of the system. Precisely this combination of properties was defined by 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

[6] Dahl and Nygaard (1966), p. 671.

[7] Nygaard and Dahl (1981), p. 440.

Table 11.1 A Simula program

```
SIMULA begin
integer population, nr uninfected, U1, U2;
real p; array mortality [4:10];
set dead, cured;
activity infected person;
begin integer day;
      nr uninfected := nr uninfected −1;
      hold(3);
      for day := 4 step 1 until 10 do
      if draw(mortality[day], U1) then
      begin include(current,dead); terminate(current) end
      else hold(1);
      include(current, cured);
end;
read(population, U1, U2, p, mortality);
nr uninfected := population;
infect: activate new infected person;
hold(negexp(p, U2));
if nr uninfected >0 then go to infect;
hold(11);
write(population, cardinal(dead), cardinal(cured));
end;
```

the notion of a collection of such programs, called 'processes,' conceptually operating in parallel.[8]

This extension changed the lifetimes of blocks: in Algol, blocks could be nested, with the consequence that data defined in the inner block could only be maintained as 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. This made the Algol discipline too restrictive, and 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".[9]

Processes were defined by *activity* declarations, which were rather like procedure declarations. Whereas procedures were called, however, processes were created, or *generated*, from activities using a symbol *new*. This returned a pointer to the new process which could be stored to allow it to be referenced throughout its life. Simula defined special data structures, called *sets*, to manage collections of processes and the queue of events waiting to be dealt with.

Table 11.1 shows a simple Simula program modelling the spread of an infection through a population.[10] Simulations were included within a special *SIMULA* block; this could be the outermost block in a program, as here, or nested within a normal Algol block. The program defines one activity, representing an infected person,

[8] Dahl and Nygaard (1966), p. 671.

[9] Dahl and Nygaard (1965), p. 14.

[10] Dahl and Nygaard (1965), pp. 95–96.

which applies the probabilities of death on days 4 to 10 of an infection, given in the *mortality* array, and assigns the person to one of the sets *dead* or *cured*, depending on the outcome.

The declarations in the *SIMULA* block hold the global data required for the simulation, and the statements following the declaration of the activity handle input and output, and create processes representing infected people which are then added to the simulation. Once a process is created, the subsequent course of the infection is modelled solely by the code appearing in the activity. Statements such as *activate*, *hold* and *terminate* provided means for controlling the progress of the computations performed by individual processes.

11.2 Modelling the Real World

Because of the nature of simulation systems, languages like Simula were naturally described as providing the ability to *model* certain aspects of the real world. This capability was supported primarily by giving programmers 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 the area of business data processing where Cobol, for example, provided the ability to exhaustively describe the structure of data stored in files.

The ability to work with structured data was not supported by scientific languages such as Fortran and Algol, however. In the mid-1960s proposals were put forward, most notably by Wirth and Hoare, for adding a general record handling capability to such languages, as described in Sect. 9.6. These proposals made the connection with modelling explicit: 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*".[11]

The most well-developed proposals for record handling defined the relationship between the real world and the computer model in terms of four properties. Firstly, objects were considered to possess a number of *attributes* represented by *fields* in a record, each of which could hold a data value describing the object. Secondly, similar objects would naturally have the same kinds of attributes, though normally holding different values. Objects could therefore be grouped into classes, and in a program the attributes belonging to a particular class of objects would be defined by a *record class*. 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,

[11] Hoare (1968), p. 294.

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 ways in which Algol 60 could be extended to permit the manipulation of structured data which modelled real-world entities. Simula achieved this by generalizing the Algol notion of a block; as Hoare pointed out, however, this confused the record concept with that of the process, thought of as "a rule of behaviour as specified by procedural statements",[12] and brought with it the complexities of parallel processing. Records, on the other hand, provided a new language feature which isolated the central problem of handling structured data, and therefore reinforced such characteristic themes of the Algol research programme as clarity and the need to be able to understand and control the behaviour of programs.

11.3 Simula 67

In 1967, Dahl and Nygaard defined a revised version of Simula, which drew on the experience they had gained with the use of Simula, but also responded to proposals made for using records to handle structured data. Unlike its predecessor, Simula 67 was intended to be 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"[13] 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";[14] 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. In a terminological change which was 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".[15] 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.

[12] Hoare (1968), p. 330.

[13] Dahl et al. (1968).

[14] Dahl et al. (1968), p. 1.

[15] Dahl et al. (1968), p. 5.

A significant innovation in Simula 67 was the introduction of *prefix classes*, which were intended to be an alternative to the record subclasses that Hoare had described.[16] 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 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 the vehicle class. The importance of this notion lies in the ability to define the capacity field differently in different subclasses of vehicle. A programmer would then be able to refer to the capacity of a vehicle without knowing in detail what sort of vehicle was referred to at run-time. Record handling proposals contained no similar capability: the emphasis on the concept of data types in the Algol research programme made it highly desirable that every field in a record was fully defined when a program was compiled.

11.4 Data Abstraction

In Sect. 10.6 it was argued that as structured programming made the transition from academia to industry, it became principally identified with two of the ideas that Dijkstra had put forward, namely the use of a restricted repertoire of control structures and the employment of a top-down approach to program development. A third idea,

[16] Dahl and Nygaard (1968).

that programs could be structured as a layered hierarchy of *machines*, was in comparison rather overlooked.

Descriptions of top-down methods did, of course, often recognize the very close relationship between a program's data structures and the operations that acted on them, and the need to consider both together. For example, Wirth stated that "[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".[17] However, top-down methods were often taken to consist of a process of specifically *functional* decomposition, or analysis. Wirth later made this clear, writing 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".[18]

This approach reinforced, and was itself perhaps supported by, the traditional ideas that subroutines, or functions, were the fundamental components out of which programs were built, and that the structure of a program should be conceived of as a hierarchy of functions. As discussed above, Simula and Simula 67 employed a different model, in which at the top-level programs were composed of blocks rather than functions. However, for reasons that will be considered in more detail below, proposals such as Simula 67's classes were not adopted in other languages, despite the inclusion of an extended discussion of the ideas underlying Simula in the widely read book *Structured Programming*, published in 1972.[19] Instead, the early 1970s saw extensive discussion of new language mechanisms intended to provide support for more data-oriented program modules, and the development of languages based on these new ideas.

One important theme in this discussion was the idea of isolating data and placing restrictions on the ways in which it could be accessed directly. 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 systems programs, data structures frequently needed to be changed. This made it plausible 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".[20] This was achieved by enabling access to the elements of a data structure through a function-like interface and, along with the data structure, defining 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 approach was taken further by James Morris, who introduced a notion of data 'protection'. As well as describing programming language mechanisms which

[17] Wirth (1971a), p. 221.

[18] Wirth (1974), p. 249.

[19] Dahl et al. (1972).

[20] Wulf et al. (1971), p. 787.

would enable a data structure and procedures for accessing and manipulating the data to be closely associated, Morris described methods for preventing other parts of the program from accessing the data structure directly.[21] Stephen Zilles described a related approach, which he called 'procedural abstraction', here defined as "the technique of representing system components in terms of one or more procedures such that interactions among components are limited to procedure calls".[22] As this description suggests, it was widely assumed that the procedure 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 functions were in fact the best type of component to base the design of software on was 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". His alternative proposals were based on a principle of 'information hiding', and he 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".[23] This idea is clearly related to that of Dijkstra, who had 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)".[24]

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, data structures are commonly accessed by many different operations. If the subroutines representing these operations are written in such a way that they rely on detailed knowledge of the data structure, then when the data structure changes, all the dependent modules will need to be modified. An alternative approach would be to conceal the choice of representation in a single module, which would then make a more abstract representation of the data available to other modules, together with the ability to manipulate it. Dijkstra proposed 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.[25]

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".[26]

A concrete language proposal based on these ideas was proposed in a session on structured programming at an ACM meeting in 1973, where Barbara Liskov

[21] Morris Jr. (1973).

[22] Zilles (1973).

[23] Parnas (1972), p. 1058.

[24] Dijkstra (1969b), p. 87.

[25] Dijkstra (1969b), p. 87.

[26] Parnas (1972), p. 1056, emphasis in original.

put forward the notion 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".[27] She noted that classes as defined in Simula 67 provided a similar feature, but also that they did not fully support the desired notion of abstraction, as they made the data representation accessible to other modules.

A new language meeting these requirements was more fully described in the following year by Liskov and Zilles. They again emphasized the connection between this work and 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". This language, later named CLU, defined a new program structure, the *cluster*. Clusters provided a way of implementing an abstract data type, defined as "a class of abstract objects which is completely characterized by the operations available on those objects". An example cluster is given in Table 11.2.[28]

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. A program using the stack cluster in Table 11.2 would therefore be able to declare variables that held stacks, but only manipulate them 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. This was defined in the *rep* clause of the cluster, and the operations were then implemented 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.

Clusters were not simply a proposal for a new programming language feature, however. Liskov and Zilles also saw a strong connection between the use of abstract data types and giving correctness proofs of programs. They argued that the use of abstraction enabled the task of proving the correctness of a program to be split into two independent parts, first proving the correctness of the abstract program that used the data abstraction, and then proving the correctness of the implementation of the data abstraction itself. This division of labour could be expected to make program proving simpler and more effective.

In order to carry out such proofs formally, however, it would be 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" was suitable for specifying a

[27] Liskov (1973), pp. 6–7.

[28] Liskov and Zilles (1974), pp. 50, 51, 54.

Table 11.2 A cluster defining an abstract data type of stacks

```
stack: cluster(element_type: type) is push, pop, top, erasetop, empty;

    rep(type_param: type) = (tp: integer;
                             e_type: type;
                             stk: array[1..] of type_param);

    create
        s: rep(element_type);
        s.tp := 0;
        s.e_type := element_type;
        return s;
        end

    push: operation(s: rep, v: s.e_type);
        s.tp := s.tp + 1;
        s.stk[s.tp] := v;
        return;
        end

    pop: operation(s: rep) returns s.e_type;
        if s.tp = 0 then error;
        s.tp := s.tp −1;
        return s.stk[s.tp+1];
        end

    top: operation(s: rep) returns s.e_type;
        if s.tp = 0 then error;
        return s.stk[s.tp];
        end

    erasetop: operation(s: rep) returns s.e_type;
        if s.tp = 0 then error;
        s.tp := s.tp − 1;
        return;
        end

    empty: operation(s: rep) returns boolean;
        return s.tp = 0;
        end

    end stack
```

procedural abstraction, but not a data abstraction.[29] They considered various ways in which formal specifications of data abstractions could be given, including the use of existing mathematical models such as graph theory, state machine models, and the possibility of giving more abstract axiomatic or algebraic specifications, before concluding that none of the existing techniques were adequate.

From the mid-1970s on, 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

[29] Liskov and Zilles (1975), p. 10.

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 an 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. An initial difficulty was that 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 is returned.

Another important difference was connected with the idea of protection: abstract data types provided a barrier which allowed programmers to manipulate data only by means of the provided operations. Simula 67, by contrast, allowed programs to have 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.

11.5 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 types. A number of new languages were based on the idea, pioneered by CLU, of a program module which defined and encapsulated an abstract data type. Perhaps the most significant of these languages was Ada, developed in the late 1970s and early 1980s by the US Department of Defense.[30]

Later languages such as C++ and Java did not follow this approach, however, adopting instead a form of program module derived from the Simula 67 notion of

[30]Department of Defense (1983). It should be noted that the way in which Ada supported data abstraction was different in certain technical respects from CLU. The basic principles of data protection and an operational interface were preserved, however.

a class. Languages based on this kind of module 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 onwards.

Smalltalk was intended to be the programming language that would 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". 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".[31]

The origins of Algol lay in the tradition of scientific programming carried out in the 1950s. A typical problem in this tradition was to devise an algorithm to carry out a particular computation, and Algol was conceived as a language for expressing and communicating such algorithms. The context in which Smalltalk was developed was very different, however: it was part of a project to develop a highly interactive device which would be usable by a wide range of people, including young children. Crucially, 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 not thought of as the production of code by following a process modelled on engineering process, but as an ongoing interaction with a complex and reactive system. This outlook profoundly shaped the design of Smalltalk, which Kay and Goldberg tended to describe as a "communications system ... implemented on small computers" rather than as a programming language.

Kay was inspired by Simula's notion of an object, and in particular by the idea of integrating 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 ...?" However, Kay was not interested in taking the existing ideas of simulation programming or abstract data types further; rather, "[i]t was the promise of an entirely new way to structure computations that took my fancy". 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 the use of 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, but

[31] Kay and Goldberg (1977), pp. 31, 40.

based on a different primitive notion, namely the idea of message passing between objects.[32]

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. 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".[33] Despite the Simula-like terminology, however, it was recognized that there were important differences. 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".[34] 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.[35] By contrast, Smalltalk made it a fundamental feature of the language, applying to all inter-object communication.

Smalltalk-72 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 it is strikingly different from the manuals written for languages in the Algol tradition.

The first significant difference arose directly from the purpose of the language, to be a medium for communicating with the Dynabook. Programmers, or users, did not submit the texts of Smalltalk programs to a compiler for subsequent execution; instead, expressions were entered into a "Smalltalk dialog window" and evaluated immediately by the Smalltalk system. In the instruction manual, this process was described as "talking to Smalltalk". In the simplest cases, arithmetic expressions could be entered, and the system would respond with the value of the expression, thus functioning as a simple calculator.

A number of introductory examples involved drawing shapes on the screen of the Dynabook, using a 'turtle'. A turtle could draw lines of various lengths and directions on the screen; the name was inspired by the appearance of a pen-holding robot that drew lines on paper on the floor. The Smalltalk system predefined one instance of the turtle class, denoted by the 'smiley' symbol ☺. Messages could be

[32] Kay (1996), pp. 516, 517, 531.

[33] Goldberg and Kay (1976), p. 6.

[34] Shoch (1979), p. 72.

[35] Dahl et al. (1968), p. 24.

sent to the turtle to move it around the screen and draw lines by entering expressions such as the following:

☺ *go* 100
☺ *go* 100 *turn* 90

As these examples illustrate, a Smalltalk statement consisted of the name of an object, ☺ in this case, followed by the name of a message, such as *go*. Message parameters could follow the message name, and more than one message could be sent to an object in one statement.

The language also defined statements which allowed simple command structures to be expressed. For example, the *do* statement could be used to repeat a statement a given number of times, so the following example would cause the turtle to draw a square, starting at its current position:

do 4 (☺ *go* 100 *turn* 90)

Iteration could also be expressed using the *for* statement. The following example clears the screen, resets the turtle to its default starting position, and then draws a spiral:

☺ *erase home*
for i ← 1 *to* 200 *do* (☺ *go i**2 *turn* 89)

Unusually, Smalltalk also provided a means for expressing unbounded iteration, whereby a statement could be repeated indefinitely. In the following example, the message *goto* causes the turtle to draw a line from its current position to the point given by the coordinates following, in this case the coordinates of the mouse cursor which were always available in the variables *mx* and *my*. The effect of this was to cause the turtle to follow the mouse, thus allowing the mouse to be used as a simple drawing tool.

repeat (☺ *goto mx my*)

This is a striking example of the effect that a different set of priorities could have on programming style. In scientific programming, non-terminating loops were often taken to be a severe error: Strachey had declared that programs which contained them were meaningless, and methods for proving that a program terminated when expected were intensively investigated. In Smalltalk, however, unbounded iteration simply provided for an open-ending and ongoing interaction between the user and the computer, and non-termination was not seen as a problem: "[t]o escape from the loop and get Smalltalk to listen to you again, press the key marked 'ESC'".[36]

New methods and classes were defined by providing suitably formatted textual definitions. For example, a parameterized method to draw a square could be defined as follows:

```
to square size
  (☞ size ← :.
    do 4
      (☺ go size turn 90))
```

[36] Goldberg and Kay (1976), p. 5.

Table 11.3 A class definition in Smalltalk-72

to box	*var* | *x y size tilt*	
(◁ *draw*	⇒	(☺ *place x y turn tilt. square size*)
◁ *undraw*	⇒	(☺ *white*. SELF *draw*. ☺ *black*)
◁ *turn*	⇒	(SELF *undraw*. ☞ *tilt* ← *tilt* + :. SELF *draw*)
◁ *grow*	⇒	(SELF *undraw*. ☞ *size* ← *size* + :. SELF *draw*)
isnew	⇒	(☞ *x* ← ☞ *y* ← 256. ☞ *size* ← 50. ☞ *tilt* ← 0. SELF *draw*))

In this example, both the word *to* and the pointing hand symbol ☞ are kinds of quoting devices, which indicate that what follows is a definition of the literal following the quote. The method *square* is defined with one parameter, *size*, which is initialized with the next value in the message, provided by the symbol :. The body of the method simply repeats the code for drawing a square shown above. To use this method to draw a square, the user would simply type something like:

square 50

The definition of a new class, *box*, is given in Table 11.3.[37] The first line defines a temporary variable *var* and four instance variables, which hold the data values which give the individual properties of each instance of the class. The remainder of the definition lists the four messages that individual boxes know how to respond to. The special 'question' *isnew* was required in every class definition, and defined how a new instance of the class should be created.

The details of the Smalltalk language used in this example are not important for the current discussion. A new instance of the box class could be created by typing something like the following:

☞ *joe* → *box*

and the new box could then be sent messages in the normal way.

Later Smalltalk Developments Subsequent versions of Smalltalk introduced new language features, most notably subclassing, the Smalltalk equivalent of Simula 67's prefix classes, in Smalltalk-76.[38] 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 different classes 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 drew a distinction between the "object oriented" approach of Smalltalk and the 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

[37] Goldberg and Kay (1976), p. 18.

[38] Ingalls (1978).

for programmers to use, Smalltalk provided "a set of well developed superclasses from which most of the system classes are derived".[39] 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 book, however, and the relationship between Smalltalk and the Algol research programme will now be considered.

11.6 The Relationship Between Smalltalk and Logic

The previous section described some of the key features of Smalltalk-72, suggesting that its origins and the motivations of its designers were very different from those of Algol 60. Although the influence of Simula was important in its development, Smalltalk appears to represent an approach to the design of programming languages that is quite different from what was familiar in the Algol research programme. This section supports this claim by describing ways in which Smalltalk differs from the notion of a programming language which was based on Carnap's notion of a formal language.

Smalltalk as a Formal Language In one sense, every programming notation 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 Chap. 8, 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, 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 methods and 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, Ingalls described the situation when a class text was brought up for editing by saying that "[t]he class has thus

[39] Ingalls (1978), p. 9.

provided a simulation of itself as structured text".[40] 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 separate 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 therefore not 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 Chap. 4. 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 to do with more than simply syntax, however, and extend to the general understanding of what computation is.

As described in Chaps. 5 and 6, the principal motivation for the development of digital computers was to automate calculation, and the canonical design that

[40] Ingalls (1978), p. 10.

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 one set of features for expressing algorithms and a largely separate set for describing data structures. Within this tradition, programs were fundamentally seen as a way of 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 principle of computation. In this understanding, a Smalltalk system constitutes a complete 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.[41] 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 striking that, like the Algol model, this overall approach reflects aspects of the target 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 artefacts: 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 related 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

[41] Shoch (1979).

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 foretold 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 Finally, 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 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, one which is still in evidence to the present day.

11.7 Conclusions

By the early 1970s, then, programming language researchers had identified that a key issue was 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 has been suggested that Smalltalk, in contrast with many languages that had been developed in the preceding decade, owed little to the influence of logic, and so marked a significant break with the Algol research programme. 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.

Chapter 12
Conclusions

The overarching theme of this book has been the story of how computational agency has, over a period of several centuries, migrated from humans to machines. From the sixteenth century onwards, it became increasingly possible, although often controversial, to see mathematics as a mechanical process of symbol manipulation. Chapter 2 described how Babbage's engines can be seen as an attempt to take this idea literally, to conceive of physical machines which could simulate, or perhaps carry out, mental activities. This theme continued to be important in the later history of the computer, and is particularly evident in the relationship between cybernetics and computer development in the 1930s and 1940s.

This migration brought with it the need to find a new kind of language, one which was adequate to express computational processes in such a way that they could be carried out 'mechanically', either literally, or at least in such a way as to make minimal demands on human faculties of interpretation and judgement. New forms of mechanical language first appeared in mathematics, as described in Chap. 1, and then in mathematical logic. As recounted in Chap. 4, this in turn gave rise to a well-developed metalinguistic account of the relevant notion of formal, or mechanical language.

To begin with, mathematical logic only formalized the language of imperative statements and proof, however, and it was only in the mid-1930s that completely formal notations for the expression of algorithms and computational processes were developed. These particular notations turned out not to be ideally suited for use with the technology of automatic computation that was emerging in the same period, however, and several decades were to pass before developments converged on a stable set of basic concepts and ideas.

It is a central proposal of this book that this convergence was the achievement of the Algol research programme, the tradition of work that was inspired by the Algol 60 language proposals and culminated in the widespread adoption of the ideas about programming notation associated with the term 'structured programming', as described in Chap. 9. These ideas have a permanence and centrality in both the theory and practice of computation that is comparable to that of the notation of the predicate calculus in logic.

M. Priestley, *A Science of Operations*, History of Computing,
DOI 10.1007/978-1-84882-555-0_12, © Springer-Verlag London Limited 2011

12.1 Paradigms and Revolutions

This book has provided an account of the influence that the established discipline of mathematical logic had 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. The significance of this report was explained, however, by proposing that it played the role of a concrete paradigm, in Kuhn's term, for what has been described here as the Algol research programme.

The key characteristics of the Algol research programme were highlighted by making use of the concept of a research programme as defined by Imre 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*)".[1] The negative heuristic tells researchers to preserve at all costs certain propositions, the "hard core" of the programme. In the case of the Algol research programme, it was suggested in Chap. 8 that the hard core was roughly the proposition that programming languages should be understood to be formal languages in the sense established by mathematical logicians in the 1930s. This in turn was described, making use of Andrew Pickering's schematic account of conceptual innovation, as an example of *bridging*.[2]

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".[3] The positive heuristic of the Algol research programme was expounded most influentially by John McCarthy in the early 1960s.[4] A large part of McCarthy's suggestions 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 Chaps. 9 and 10, 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, only reinforces this picture.

By the early 1970s, the Algol research programme had made significant progress and it was argued in Chap. 10 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 an approach 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.

---

[1] Lakatos (1970), p. 132.

[2] Pickering (1995).

[3] Lakatos (1970), p. 135.

[4] McCarthy (1961, 1962).

The use of structural concepts from the philosophy of science, such as 'paradigm' and 'research programme', in this account suggests further possible observations about the history of programming languages. For example, it could be argued that the Algol programme in fact provided the first paradigm in the field of programming language design, a claim based not on an isolated evaluation of the merits of Algol, 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 Chap. 9.

One 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 description does not underplay the importance of earlier achievements, in particular Fortran, but draws attention to the fact that it 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 developments made in automatic programming during the 1950s were driven by practical motives, such as 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 Chap. 8, 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.[5] To some extent a verdict on 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 preliminary work in 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 cover all subsequent work on programming languages. Chapter 11 described the early development of object-oriented languages and concluded that the Smalltalk project represented a new and independent development that, despite the existence of certain historical links, differed profoundly from approaches to program language design that were more directly influenced by logic. It is outside the scope of this book to describe in detail the later 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.[6] However, they do not differ as radically

[5] Wilkes et al. (1951).

[6] Stroustrup (1994).

from Algol-like languages as Smalltalk did, and the currently dominant form of programming language can most 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 not viewed as a single process, but as a network of intercommunicating objects.

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 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 encouraged by the frequent use of the Kuhnian term 'paradigm' to describe different approaches to programming language design. There does seem to have been a significant change in the programming languages used in industry, which until the late 1980s consisted largely of languages which supported abstract data types, but which are now more widely based on object-oriented ideas. In a development reminiscent of the way in which structured programming constructs were introduced into older languages like Fortran, furthermore, some older 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 has been 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 are still present in the new paradigm. Specifically, 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 conservative revolutions are characteristic of progress in mathematics and logic,[7] and it is therefore possible

[7] Gillies (1992).

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 still 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 application of object-oriented ideas primarily to the overall structure or programs, and also in program design. This, however, is a topic for future research.

A further interesting question is why object-oriented languages have proved more successful than those based on abstract data types. 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. Furthermore, 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 Chap. 11, it is likely that this was an important influence in the development of the ideas of object-oriented programming.

12.2 Relating Theory and Practice

Another way of looking at the influence of logic on programming is to see it as an example the *application* of theoretical ideas to practice, a distinction that is often 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.[8]

This process of application is often seen as being unproblematic: for example, Chap. 6 considered claims that the invention of the computer involved the simple application of ideas from logic, and that the computer emerged as a byproduct of theoretical research in logic. However, it appears that a far less certain and much more exploratory process took place than the simple term 'application' suggests, and that within this process ideas derived from logic 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

[8] See Hoare (1982) or Shaw (1990), for example.

later, suggesting that the interaction between theory and application in this case was 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 observations can be made about the relationship between programming and logic. Throughout the 1950s, there were several explicit attempts made 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.[9] 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:[10] 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 Chap. 10 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, the importance of program proving being 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 has been much more problematic. Informal or semi-formal top-down design became a widespread practice in practical programming, but formal program proving was not, and has not been, widely accepted. The belief that a proof-based approach to programming can guarantee the correctness of programs has been repeatedly criticized,[11] and a number of writers have commented on the lack of practical application of theoretical results.[12] 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] Turing (1949), Elgot (1954), Hamblin (1957).

[10] Jones (2003).

[11] See De Millo et al. (1979) and Fetzer (1988), for example.

[12] See Arden (1980) and Mahoney (1997).

12.3 Methodological Conclusions

Internal and External Accounts The subject matter of this book belongs to what it usually characterized as internal history, which tends to describe historical episodes in terms of their relationship with the current state of knowledge and to ignore 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. However, it is not clear that such external factors will always be sufficient to explain all the internal features of a particular subject matter, and the approach adopted here has been to remain agnostic about what kinds of contextual factors might be relevant to an explanation in any particular case.

For example, in Chap. 4 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 its role in the origins of modern computing. However, there are striking similarities between the machine table notation and that of recursive function theory and the λ-calculus, and drawing attention to these makes 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 6, 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,[13] the Second World War provides a context which cannot be ignored in discussing the development of computers and the uses to which they were put, but this broad historical background 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 Chap. 8: the principal motivation for the development of automatic programming in the 1950s was the need to make programming less laborious and time-consuming and so to enable 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 passing in Chap. 11 that some aspects of the high-level structure of programming languages might be explained by reference to 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

[13] Edwards (1996).

which a given language was intended to be used, such as scientific or commercial applications, might affect the design and style of the resulting language, and the features included in it.

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 Internal accounts of technical invention often treat episodes of innovation as pure moments of inspiration which are not amenable to analysis and explanation. This is a consequence of a perspective, known as the Whig interpretation of history,[14] which sees in the past only those aspects relevant to the present. In contrast, this book 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 4 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 Chap. 6. 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.

Such processes were also involved in the development of more technical details. Chapter 7 described how even such a basic feature of programming as performing two operations in sequence went through a period of exploration 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 Chap. 8. The interest in formula translation was prompted by a desire to widen the field of people who could program computers, but it was

[14] Butterfield (1931).

suggested in that chapter 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 formulae could be defined and processed.

The work involved in conceptual innovation provides an explanation for the not infrequent 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.

Further Directions The story of programming told in this book reflects traditional accounts in that it focuses on innovations rather than on the use of technology.[15] For example, according to the account given in Chap. 8, the early 1950s were significant for the early development of autocodes, leading up to the development of Fortran. However, as Saul 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".[16] 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.[17]

The relationship between the needs and characteristics of different application areas and the programming languages and techniques developed for them deserves more detailed attention than was possible in this book, which has focused primarily on scientific computing. As well as the major 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.[18]

[15] See (Edgerton 2006) for discussion of this distinction.

[16] Rosen (1964).

[17] Rosen (1972).

[18] Wexelblat (1981).

There are a number of technical aspects of programming languages that have only been mentioned in passing in the text, but which also deserve more detailed study. Significant among these are 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 Chap. 11, 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, and whether the nested structure of Algol 60's blocks was sufficient for all needs. It would be interesting 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.[19] This idea 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.

[19] Gillies and Zheng (2001).

Appendix
Turing's Universal Machine

This appendix gives the complete table for the universal Turing machine $\mathcal{U}$. There were a number of errors in Turing's original paper, as pointed out by Emil Post and Donald Davies.[1] These errors have been corrected in the tables below.

A.1 General Purpose *m*-functions

The tape of the universal machine uses the convention of alternating F-squares and E-squares, described in Sect. 4.5, where the E-squares are used to mark, or label, symbols on the preceding F-squares. The leftmost point of the portion of the tape used in the computation is identified by means of the symbol ə placed in a pair of adjacent squares.

$\mathfrak{f}(\mathfrak{C}, \mathfrak{B}, \alpha)$**: find** The *m*-function $\mathfrak{f}(\mathfrak{C}, \mathfrak{B}, \alpha)$ finds the leftmost occurrence on the tape of the symbol α, and then moves to *m*-configuration $\mathfrak{C}$. If there is no occurrence of α on the tape, the machine moves to *m*-configuration $\mathfrak{B}$.

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	$\mathfrak{B}$

[1] See Post (1947) and Davies (2004). Davies describes Turing's "impatience" when these errors were pointed out, remembering that he "made it clear that I was wasting my time and his by my worthless endeavors".

M. Priestley, *A Science of Operations*, History of Computing,
DOI 10.1007/978-1-84882-555-0, © Springer-Verlag London Limited 2011

$\mathfrak{e}(\mathfrak{C}, \mathfrak{B}, \alpha)$**: erase** The *m*-function $\mathfrak{e}(\mathfrak{C}, \mathfrak{B}, \alpha)$ erases the leftmost occurrence on the tape of the symbol α, and then moves to *m*-configuration $\mathfrak{C}$. If there is no occurrence of α on the tape, the machine moves to *m*-configuration $\mathfrak{B}$.

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	$\mathfrak{C}$

$\mathfrak{e}(\mathfrak{B}, \alpha)$**: erase all** The *m*-function $\mathfrak{e}(\mathfrak{B}, \alpha)$ erases all occurrences on the tape of the symbol α, and then moves to *m*-configuration $\mathfrak{B}$.

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

$\mathfrak{pe}(\mathfrak{C}, \beta)$**: print at end** The *m*-function $\mathfrak{pe}(\mathfrak{C}, \beta)$ prints the symbol β at the end of the sequence of symbols on the tape, and then moves to *m*-configuration $\mathfrak{C}$.

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

$\mathfrak{pe}(\mathfrak{C}, \beta)$ moves to the start of the tape, and then $\mathfrak{pe}_1(\mathfrak{C}, \beta)$ searches for the first blank F-square before printing β.

A variant of this *m*-function prints two symbols at the end of the tape:

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{pe}_2(\mathfrak{C}, \alpha, \beta)$			$\mathfrak{pe}(\mathfrak{pe}(\mathfrak{C}, \beta), \alpha)$

$\mathfrak{l}(\mathfrak{C})$**: move left** The *m*-function $\mathfrak{l}(\mathfrak{C})$ performs the basic operation L, and then moves to *m*-configuration $\mathfrak{C}$.

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{l}(\mathfrak{C})$		L	$\mathfrak{C}$

$\mathfrak{f}'(\mathfrak{C}, \mathfrak{B}, \alpha)$**: find and move left** The *m*-function $\mathfrak{f}'(\mathfrak{C}, \mathfrak{B}, \alpha)$ finds the leftmost occurrence on the tape of the symbol α, moves one square to the left, and then moves to *m*-configuration $\mathfrak{C}$. If there is no occurrence of α on the tape, the machine moves to *m*-configuration $\mathfrak{B}$.

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

$\mathfrak{c}(\mathfrak{C}, \mathfrak{B}, \alpha)$**: copy** The *m*-function $\mathfrak{c}(\mathfrak{C}, \mathfrak{B}, \alpha)$ finds the leftmost symbol on the tape marked with the symbol α, copies it to the end of the tape, and then moves to *m*-configuration $\mathfrak{C}$. If there is no symbol marked with α on the tape, the machine moves to *m*-configuration $\mathfrak{B}$.

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{c}(\mathfrak{C}, \mathfrak{B}, \alpha)$			$\mathfrak{f}'(\mathfrak{c}_1(\mathfrak{C}), \mathfrak{B}, \alpha)$
$\mathfrak{c}_1(\mathfrak{C})$	β		$\mathfrak{pe}(\mathfrak{C}, \beta)$

$\mathfrak{ce}(\mathfrak{C}, \mathfrak{B}, \alpha)$**: copy and erase** The *m*-function $\mathfrak{c}(\mathfrak{C}, \mathfrak{B}, \alpha)$ finds the leftmost symbol on the tape marked with the symbol α, copies the symbol to the end of the tape, and then erases the symbol and moves to *m*-configuration $\mathfrak{C}$. If there is no symbol marked with α on the tape, the machine moves to *m*-configuration $\mathfrak{B}$.

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

$\mathfrak{ce}(\mathfrak{B}, \alpha)$**: copy and erase all** The *m*-function $\mathfrak{ce}(\mathfrak{B}, \alpha)$ copies all symbols marked with α to the end of the tape, erases them, and then moves to *m*-configuration $\mathfrak{B}$.

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

The table for the universal machine uses a variant of this *m*-function which will copy and erase symbols marked with five different symbols. This can be defined in terms of $\mathfrak{ce}(\mathfrak{C}, \mathfrak{B}, \alpha)$ as follows:

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{ce}_5(\mathfrak{B}, \alpha, \beta, \gamma, \delta, \varepsilon)$			$\mathfrak{ce}(\mathfrak{ce}(\mathfrak{ce}(\mathfrak{ce}(\mathfrak{ce}(\mathfrak{B}, \varepsilon), \delta), \gamma), \beta), \alpha)$

$\mathfrak{cp}(\mathfrak{C}, \mathfrak{U}, \mathfrak{G}, \alpha, \beta)$**: compare** The *m*-function $\mathfrak{cp}(\mathfrak{C}, \mathfrak{U}, \mathfrak{G}, \alpha, \beta)$ compares the first symbol on the tape marked with α and the first symbol marked with β. If there are no symbols marked with α or β, the machine moves to *m*-configuration $\mathfrak{G}$; if both exist, and the marked symbols are the same, it moves to *m*-configuration $\mathfrak{C}$; otherwise, it moves to *m*-configuration $\mathfrak{U}$.

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{cp}(\mathfrak{C}, \mathfrak{U}, \mathfrak{G}, \alpha, \beta)$			$\mathfrak{f}'(\mathfrak{cp}_1(\mathfrak{C}, \mathfrak{U}, \beta), \mathfrak{f}(\mathfrak{U}, \mathfrak{G}, \beta), \alpha)$
$\mathfrak{cp}_1(\mathfrak{C}, \mathfrak{U}, \beta)$	γ		$\mathfrak{f}'(\mathfrak{cp}_2(\mathfrak{C}, \mathfrak{U}, \gamma), \mathfrak{U}, \beta)$
$\mathfrak{cp}_2(\mathfrak{C}, \mathfrak{U}, \gamma)$	γ		$\mathfrak{C}$
	not γ		$\mathfrak{U}$

$\mathfrak{cpe}(\mathfrak{C}, \mathfrak{U}, \mathfrak{G}, \alpha, \beta)$**: compare and erase** The *m*-function $\mathfrak{cpe}(\mathfrak{C}, \mathfrak{U}, \mathfrak{G}, \alpha, \beta)$ does the same as $\mathfrak{cp}(\mathfrak{C}, \mathfrak{U}, \mathfrak{G}, \alpha, \beta)$, but if symbols marked with α and β exist and are the same, the α and β are erased.

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{cpe}(\mathfrak{C}, \mathfrak{U}, \mathfrak{G}, \alpha, \beta)$			$\mathfrak{cp}(\mathfrak{e}(\mathfrak{e}(\mathfrak{C}, \mathfrak{C}, \beta), \mathfrak{C}, \alpha), \mathfrak{U}, \mathfrak{G}, \alpha, \beta)$

$\mathfrak{cpe}(\mathfrak{U}, \mathfrak{G}, \alpha, \beta)$**: compare and erase all** The *m*-function $\mathfrak{cpe}(\mathfrak{U}, \mathfrak{G}, \alpha, \beta)$ compares the sequences of symbols on the tape marked with α and β. So long as they are the same, the αs and βs are erased. If the sequences are identical, the machine moves to *m*-configur ation $\mathfrak{G}$ when all the marks have been removed; it moves to *m*-configuration $\mathfrak{U}$ when a difference is detected.

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{cpe}(\mathfrak{U}, \mathfrak{G}, \alpha, \beta)$			$\mathfrak{cpe}(\mathfrak{cpe}(\mathfrak{U}, \mathfrak{G}, \alpha, \beta), \mathfrak{U}, \mathfrak{G}, \alpha, \beta)$

$\mathfrak{q}(\mathfrak{C})$**: find end of tape** The *m*-function $\mathfrak{q}(\mathfrak{C})$ moves to the end of the tape, identified by two consecutive blank squares, and then moves to *m*-configuration $\mathfrak{C}$.

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{q}(\mathfrak{C})$	Any	R	$\mathfrak{q}(\mathfrak{C})$
	None	R	$\mathfrak{q}_1(\mathfrak{C})$
$\mathfrak{q}_1(\mathfrak{C})$	Any	R	$\mathfrak{q}(\mathfrak{C})$
	None		$\mathfrak{C}$

$\mathfrak{q}(\mathfrak{C}, \alpha)$**: find last** The *m*-function $\mathfrak{q}(\mathfrak{C}, \alpha)$ finds the last occurrence of α on the tape, and then moves to *m*-configuration $\mathfrak{C}$.

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{q}(\mathfrak{C}, \alpha)$			$\mathfrak{q}(\mathfrak{q}_1(\mathfrak{C}, \alpha))$
$\mathfrak{q}_1(\mathfrak{C}, \alpha)$	α		$\mathfrak{C}$
	not α	L	$\mathfrak{q}_1(\mathfrak{C}, \alpha)$

If there is no occurrence of α on the tape, the machine will endlessly search leftwards, moving beyond the beginning of tape markers.

$\mathfrak{e}(\mathfrak{C})$**: erase marks** The *m*-function $\mathfrak{e}(\mathfrak{C})$ erases all marks from the tape, and then moves to *m*-configuration $\mathfrak{C}$.

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{e}(\mathfrak{C})$	ə	R	$\mathfrak{e}_1(\mathfrak{C})$
	not ə	L	$\mathfrak{e}(\mathfrak{C})$
$\mathfrak{e}_1(\mathfrak{C})$	Any	R, E, R	$\mathfrak{e}_1(\mathfrak{C})$
	None	R	$\mathfrak{C}$

A.2 The Contents of the Tape

It is assumed that an enumeration $q_1, \ldots, q_m$ is given of the *m*-configurations used in the table of $\mathcal{T}$, the machine to be simulated, along with an enumeration $S_0, \ldots, S_n$ of the symbols which can appear on its tape. S_0 is a 'blank' symbol, which is thought of as appearing in empty squares; S_1 is the digit 0 and S_2 the digit 1.

The table of $\mathcal{T}$ is assumed to have been reduced to standard form, consisting of a number of instructions each of which has one of the following forms:

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

Each instruction has five components: a configuration, consisting of the initial *m*-configuration and the symbol in the scanned square, the symbol that will be written in the scanned square (this will be the blank symbol S_0 if the existing symbol is to be erased), the move that the machine will make, and the final *m*-configuration.

Coding Instructions The table of $\mathcal{T}$ is written on the tape of $\mathcal{U}$ as a standard description; this is an encoding of its standard form using a limited set of symbols, namely ;, A, C, D, L, R and N. The m-configuration q_i is coded by a D followed by a sequence of i As, and the symbol S_i by a D followed by a sequence of i Cs. Each instruction is preceded by a semi-colon, and the entire description is written on the F-squares of the tape immediately following the beginning of tape symbols; the end of the description is marked by the special symbol ::, as in the following example.

$$; DADDCRDAA; DAADDRDAAA; DAAADDCCRDAAAA; DAAAADDRDA ::$$

Complete Configurations $\mathcal{U}$ also needs to record the *complete configuration* of $\mathcal{T}$, namely the contents of its tape, the location of the currently scanned square, and its current m-configuration. This will provide all the information necessary to select and perform the next instruction of the simulated machine. Complete configurations are recorded by listing the sequence of symbols on $\mathcal{T}$'s tape. The current m-configuration is inserted into this sequence just before the currently scanned symbol. For example, the complete configuration

$$S_1 S_0 q_2 S_2$$

represents a configuration where $\mathcal{T}$ is in state q_5 and with scanned symbol S_2.

Because complete configurations consist only of symbols and m-configurations, they can be represented on $\mathcal{U}$'s tape by following the same coding conventions as are used to create standard descriptions of machine tables. Using these conventions, the complete configuration shown above would be represented by the following sequence of symbols:

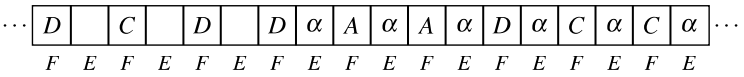
$$DCDDAADCC.$$

At any step in the computation, the behaviour of $\mathcal{T}$ is determined by its current m-configuration and the symbol in the currently scanned square, together known as its *configuration*. In the context of a complete configuration, a configuration can be identified by looking for an A, which identifies an m-configuration: the configuration is then the sequence of symbols starting with the D preceding the A, and including all immediately following As and the D and Cs following that represent the currently scanned symbol.

An auxiliary table $\mathfrak{con}(\mathfrak{C}, \alpha)$ is defined to carry out the common task of marking a configuration with a given symbol α.

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{con}(\mathfrak{C}, \alpha)$	Not A	R, R	$\mathfrak{con}(\mathfrak{C}, \alpha)$
	A	$L, P\alpha, R$	$\mathfrak{con}_1(\mathfrak{C}, \alpha)$
$\mathfrak{con}_1(\mathfrak{C}, \alpha)$	A	$R, P\alpha, R$	$\mathfrak{con}_1(\mathfrak{C}, \alpha)$
	D	$R, P\alpha, R$	$\mathfrak{con}_2(\mathfrak{C}, \alpha)$
$\mathfrak{con}_2(\mathfrak{C}, \alpha)$	C	$R, P\alpha, R$	$\mathfrak{con}_2(\mathfrak{C}, \alpha)$
	Not C	R, R	$\mathfrak{C}$

This table will mark the first configuration to the right of the machine's starting position. For example, if the machine was positioned at the left of the complete configuration shown above, at the conclusion of the above table, the configuration would be marked as follows:



For clarity, in the explanations below the symbols in the F-squares will be written consecutively, above the symbols that mark them. In this notation, the tape fragment above would be shown as follows:

$$DCD\underbrace{DAADCC}_{\alpha}.$$

The Structure of $\mathcal{U}$'s Tape At the beginning of the computation the coded table of instructions for $\mathcal{T}$ is written at the start of $\mathcal{U}$'s tape, followed by the :: symbol. As each instruction is executed, both the contents of $\mathcal{T}$'s tape and its current m-configuration may change. To record these changes, each complete configuration of $\mathcal{T}$ is written at the end of $\mathcal{U}$'s tape, preceded by a colon. The current configuration can therefore always be found at the end of the tape, after the last colon.

The output of the machines that Turing considers consists of the symbols written in the F-squares. It is assumed that machines write a symbol in every F-square, and that these symbols are never changed later in the computation. The output of $\mathcal{T}$, therefore, consists of the symbols 0 or 1 written in a previously blank square. When $\mathcal{U}$ detects that $\mathcal{T}$ has output a symbol, it writes that symbol on its own tape following the current complete configuration, and separated from it by a colon. Thus the output of $\mathcal{T}$ can be retrieved from the tape of $\mathcal{U}$ by looking for these special symbols.

As the computation proceeds, then, the symbols in the F-squares of $\mathcal{U}$ have the structure indicated in the following (unrealistic) example.

$$;\underbrace{DADDCRDAA}_{\textit{instruction}};DAADDRDAAA::DCDAADCC:0:\underbrace{DCDDAADDCC}_{\textit{complete configuration}}$$

Further output symbols and complete configurations will be added to the end of the tape, but once written, a symbol in an F-square is never changed.

A.3 The Main Table

The universal machine is defined by nine skeleton tables, each of which carries out a single step in the overall computation. The relationships between the skeleton tables and the overall flow of the computation are shown in the flowchart in Fig. A.1. Each cycle round the outside loop in the flowchart corresponds to the execution of a single instruction in the description of $\mathcal{T}$. The inner loop represents a search to find the instruction that matches the current configuration of $\mathcal{T}$.

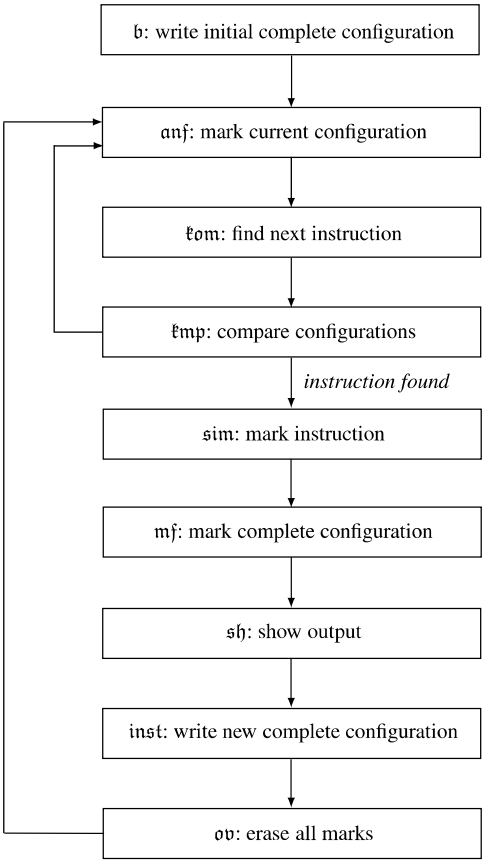


Fig. A.1 The design of the universal machine

b: Write Initial Complete Configuration At the start of a computation, then, the tape of $\mathcal{U}$ contains a pair of ə symbols marking the start of the tape, the standard description of $\mathcal{T}$, written on the succeeding F-squares, and the symbol :: in the next F-square.

The initial complete configuration of $\mathcal{T}$ is $q_1 S_0$, the initial *m*-configuration q_1 followed by the symbol representing a blank square. In standard form, this would be represented by the symbols DAD. The *m*-function b therefore writes the symbols :DAD at the end of the tape of $\mathcal{U}$, and then moves to *m*-function anf.

m-config.	*symbol*	*operations*	*final m-config.*
b			f($\mathfrak{b}_1$, $\mathfrak{b}_1$, ::)
$\mathfrak{b}_1$		R, R, P :, $R, R, PD, R, R, PA, R, R, PD$	anf

anf: Mark Current Configuration The *m*-configuration anf looks for the last colon on the tape, marks the following configuration, part of the current complete configuration of $\mathcal{T}$, with the symbol y, and then moves to *m*-configuration fom.

m-config.	*symbol*	*operations*	*final m-config.*
anf			q(anf$_1$, :)
anf$_1$			con(kom, y)

The effect of this step on the markup of $\mathcal{T}$'s current complete configuration can be shown as follows:

$$S_{i_1} \ldots S_{i_{k-1}} \underbrace{q_i S_{i_k}}_{y} S_{i_{k+1}} \ldots S_{i_n}.$$

kom: Find Next Instruction $\mathcal{U}$ must now look for the instruction applicable to the current configuration of $\mathcal{T}$. The inner cycle in Fig. A.1 searches through the instructions at the beginning of the tape until an instruction is found with the same initial *m*-configuration and scanned symbol as the current configuration.

As each instruction is located, the semi-colon preceding it is marked with a z. The *m*-configuration kom searches for the rightmost unmarked semi-colon, marks it with a z, marks the immediately following configuration with x, and then moves to *m*-configuration kmp.

m-config.	*symbol*	*operations*	*final m-config.*
kom	;	R, Pz, L	con(kmp, x)
	z	L, L	kom
	not z nor ;	L	kom

At the end of this operation, the configuration in the instruction that is currently being examined will be marked up as follows:

$$\ldots \underbrace{;}_{z} \underbrace{q_{i_n} S_{j_n}}_{x} S_{k_n} M_{l_n} q_{m_n} ; \ldots$$

kmp: Compare Configurations Next, the current configuration, marked with y, must be compared with the configuration in the current instruction, marked with x. If they differ, any remaining x and y marks are erased and $\mathcal{U}$ moves back to *m*-configuration anf to inspect the next instruction. If they are the same, the applicable instruction has been located and $\mathcal{U}$ moves to *m*-configuration sim.

m-config.	*symbol*	*operations*	*final m-config.*
kmp			cpe(e(e(anf, x), y), sim, x, y)

sim: Mark Instruction In *m*-configuration sim, the configuration in the current instruction is marked with blanks; this positions the machine after the configuration, and the rest of the instruction is marked up as follows. The z markings are then removed and the machine moves to the *m*-configuration mk.

$$q_{i_n} S_{j_n} \underbrace{S_{k_n} M_{l_n}}_{u} \underbrace{q_{m_n}}_{y}$$

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{sim}$			$\mathfrak{f}'(\mathfrak{sim}_1, \mathfrak{sim}_1, z)$
$\mathfrak{sim}_1$			$\mathfrak{con}(\mathfrak{sim}_2, \ )$
$\mathfrak{sim}_2$	A		$\mathfrak{sim}_3$
	not A	L, Pu, R, R, R	$\mathfrak{sim}_2$
$\mathfrak{sim}_3$	not A	L, Py	$\mathfrak{e}(\mathfrak{mk}, z)$
	A	L, Py, R, R, R	$\mathfrak{sim}_2$

$\mathfrak{mk}$: Mark Complete Configuration In *m*-configuration $\mathfrak{mk}$, the current complete configuration, found after the last colon on the tape, is marked up as follows and terminated with a colon, and the machine moves the *m*-configuration $\mathfrak{sh}$.

$$\underbrace{S_{i_1} \ldots S_{i_{k-2}}}_{v} \underbrace{S_{i_{k-1}}}_{x} q_i S_{i_k} \underbrace{S_{i_{k+1}} \ldots S_{i_n}}_{w} :$$

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{mk}$			$\mathfrak{q}(\mathfrak{mk}_1, :)$
$\mathfrak{mk}_1$	not A	R, R	$\mathfrak{mk}_1$
	A	L, L, L, L	$\mathfrak{mk}_2$
$\mathfrak{mk}_2$	C	R, Px, L, L, L	$\mathfrak{mk}_2$
	:		$\mathfrak{mk}_4$
	D	R, Px, L, L, L	$\mathfrak{mk}_3$
$\mathfrak{mk}_3$	not :	R, Pv, L, L, L	$\mathfrak{mk}_3$
	:		$\mathfrak{mk}_4$
$\mathfrak{mk}_4$			$\mathfrak{con}(\mathfrak{l}(\mathfrak{l}(\mathfrak{mk}_5)), \ )$
$\mathfrak{mk}_5$	Any	R, Pw, R	$\mathfrak{mk}_5$
	None	$P :$	$\mathfrak{sh}$

$\mathfrak{sh}$: Show Output In *m*-configuration $\mathfrak{sh}$, the current instruction is examined to see if $\mathcal{T}$ produces any output. If it does, the appropriate symbol is written to $\mathcal{U}$'s tape before the machine moves to *m*-configuration $\mathfrak{inst}$.

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{sh}$			$\mathfrak{f}(\mathfrak{sh}_1, \mathfrak{inst}, u)$
$\mathfrak{sh}_1$		L, L, L	$\mathfrak{sh}_2$
$\mathfrak{sh}_2$	D	R, R, R, R	$\mathfrak{sh}_3$
	not D		$\mathfrak{inst}$
$\mathfrak{sh}_3$	C	R, R	$\mathfrak{sh}_4$
	not C		$\mathfrak{inst}$
$\mathfrak{sh}_4$	C	R, R	$\mathfrak{sh}_5$
	not C		$\mathfrak{pe}_2(\mathfrak{inst}, 0, :)$
$\mathfrak{sh}_5$	C		$\mathfrak{inst}$
	not C		$\mathfrak{pe}_2(\mathfrak{inst}, 1, :)$

The machine first looks for symbols marked with u; these are the new symbol and the move symbol of the current instruction, marked up in m-configuration $\mathfrak{sim}$. A symbol only counts as output if it is a 0 or 1 written in a previously blank square. In standard form, a blank square is represented by the symbol D, 0 by DC and 1 by DCC, so in these two cases the relevant part of the tape will be marked up as follows:

$$\ldots D \underbrace{DCM}_{u} \ldots \qquad \ldots D \underbrace{DCCM}_{u} \ldots$$

M here represents one of the symbols L, R or N. The machine therefore looks first (in m-configuration $\mathfrak{sh}_1$) at the symbol *before* the first symbol marked with a u, and if it is a D, checks for one of the symbol sequences shown above. It prints 0 : or 1 : on $\mathcal{U}$'s tape as appropriate, and then moves to m-configuration $\mathfrak{inst}$.

$\mathfrak{inst}$: Write New Complete Configuration In m-configuration $\mathfrak{inst}$ the new complete configuration is written at the end of the tape. This is derived by making two changes to the last complete configuration, which was marked up in the m-configuration $\mathfrak{mk}$. Firstly, the current scanned symbol S_{i_k} should be replaced with the symbol S_{k_n}, which was marked with u in the m-configuration $\mathfrak{sim}$. Secondly, the final m-configuration q_{m_n}, marked with y the m-configuration $\mathfrak{sim}$ should be written in one of three possible positions, depending on the move operation in the current instruction.

This can all be achieved by copying the symbols marked v, x, u and w in order at the end of the tape, inserting the final m-configur ation, marked with y, at the appropriate place in this sequence of symbols. The m-configuration $\mathfrak{inst}$ first finds the symbol representing the move operation, removes the u mark from it, and then copies the remaining marked symbols as required.

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{inst}$			$\mathfrak{q}(\mathfrak{l}(\mathfrak{inst}_1), u)$
$\mathfrak{inst}_1$	L	R, E	$\mathfrak{ce}_5(\mathfrak{ov}, v, y, x, u, w)$
$\mathfrak{inst}_1$	N	R, E	$\mathfrak{ce}_5(\mathfrak{ov}, v, x, y, u, w)$
$\mathfrak{inst}_1$	R	R, E	$\mathfrak{ce}_5(\mathfrak{q}(\mathfrak{inst}_2, A), v, x, u, y, w)$
$\mathfrak{inst}_2$		R, R	$\mathfrak{inst}_3$
$\mathfrak{inst}_3$	None	PD	$\mathfrak{ov}$
	D		$\mathfrak{ov}$

The complete configurations written on $\mathcal{U}$'s tape only represent the part of $\mathcal{T}$'s tape that has been used so far. When $\mathcal{T}$ moves right and uses a new square, this has to be added explicitly as a new blank symbol at the end of the new complete configuration. This is accomplished by the last two lines of the table above.

$\mathfrak{ov}$: Erase All Marks Finally, in m-configuration $\mathfrak{ov}$, all marks are removed from the tape and $\mathcal{U}$ returns to state $\mathfrak{anf}$ to find and execute the next instruction of $\mathcal{T}$.

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{ov}$			$\mathfrak{e}(\mathfrak{anf})$

References

Ackermann, W.: Zum Hilbertschen Aufbau der reellen Zahlen. Math. Ann. **93**, 118–133 (1928). Translated as "On Hilbert's construction of the real numbers" in van Heijenoort, pp. 493–507 (1967)

ACM: Proceedings of the ACM Programming Languages and Pragmatics Conference, San Dimas, California, 8–12 August 1965. Commun. ACM **9**(3), 137–232 (1966)

Agar, J.: The Government Machine: A Revolutionary History of the Computer. MIT Press, Cambridge (2003)

Aiken, H.H.: Proposed automatic calculating machine (1937). Reprinted in Cohen and Welch, pp. 9–29 (1999)

Aiken, H.H.: The Automatic Sequence Controlled Calculator. Lecture delivered 16 July 1946. In: Campbell-Kelly, M., Williams, M.R. (eds.) The Moore School Lectures: Theory and Techniques for Design of Electronic Digital Computers. Charles Babbage Institute Reprint Series for the History of Computing, vol. 9, pp. 149–168. MIT Press, Cambridge (1985)

Aiken, H.H., Hopper, G.M.: The Automatic Sequence Controlled Calculator. Electr. Eng. **65**, 384–391, 449–454, 522–528 (1946)

Akera, A.: Voluntarism and the fruits of collaboration: the IBM user group, Share. Technol. Cult. **42**(4), 710–736 (2001)

Alt, F.L.: A Bell Telephone Laboratories computing machine—I. Math. Tables Other Aids Comput. **3**(21), 1–13 (1948a)

Alt, F.L.: A Bell Telephone Laboratories computing machine—II. Math. Tables Other Aids Comput. **3**(22), 69–84 (1948b)

Anonymous: Automatic high-speed computing: a progress report on the EDVAC, 30 September 1945. Quoted in Metropolis and Worlton, p. 55 (1980), and Aspray, p. 38 (1990b)

Archibald, R.C.: Conference on advanced computation techniques. Math. Tables Other Aids Comput. **II**(13), 65–68 (1946)

Arden, B.W. (ed.): What Can be Automated? The Computer Science and Engineering Research Study (COSERS). MIT Press, Cambridge (1980)

Ashworth, W.J.: Memory, efficiency, and symbolic analysis. Isis **87**, 629–653 (1996)

Aspray, W.F.: From mathematical constructivity to computer science: Alan Turing, John von Neumann, and the origins of computer science in mathematical logic. Ph.D. thesis, University of Wisconsin-Madison (1980)

Aspray, W. (ed.): Computing Before Computers. Iowa State University Press, Ames (1990a)

Aspray, W.: John Von Neumann and the Origins of Modern Computing. MIT Press, Cambridge (1990b)

Aspray, W., Burks, A.: Papers of John von Neumann on Computing and Computing Theory. Charles Babbage Institute Reprint Series for the History of Computing, vol. 12. MIT Press, Cambridge (1987)

DOI 10.1007/978-1-84882-555-0, © Springer-Verlag London Limited 2011

Austrian, G.D.: Hermann Hollerith. Columbia University Press, New York (1982)
Babbage, C.: Observations of the application of machinery to the computation of mathematical tables. Mem. Astron. Soc. **1**, 311–314 (1822a). Reprinted in Babbage, vol. 2, pp. 33–37 (1989)
Babbage, C.: On the Application of Machinery to the Purpose of Calculating and Printing Mathematical Tables. Booth and Baldwin, Cradock, and Joy, London (1822b). Reprinted in Babbage, vol. 2, pp. 6–14 (1989)
Babbage, C.: The science of number reduced to mechanism (1822c). Unpublished manuscript, printed in Babbage, vol. 2, pp. 15–32 (1989)
Babbage, C.: On the theoretical principles of the machinery for calculating tables. Edinb. Philos. J. (1823). Reprinted in Babbage, vol. 2, pp. 38–43 (1989)
Babbage, C.: On a method of expressing by signs the action of machinery. Philos. Trans. R. Soc. Lond. A **116**, 250–265 (1826a)
Babbage, C.: On the determination of the general term of a new class of infinite series. Trans. Camb. Philos. Soc. **2**, 217–225 (1826b). Reprinted in Babbage, vol. 2, pp. 61–68 (1989)
Babbage, C.: On the influence of signs in mathematical reasoning. Trans. Camb. Philos. Soc. **2**, 325–377 (1827). Reprinted in Babbage, vol. 1, pp. 371–408 (1989)
Babbage, C.: Notation. In: The Edinburgh Encyclopedia, vol. 15, pp. 394–399. William Blackwood, London (1830). Reprinted in Babbage, vol. 1 (1989)
Babbage, C.: Statement addressed to the Duke of Wellington respecting the calculating engine (1834). Unpublished manuscript, printed in Babbage, vol. 3, pp. 2–8 (1989)
Babbage, C.: A letter to M. Quetelet from Charles Babbage respecting the calculating machine. Acad. R. Sci., Lett. Beaux-Arts Brux. **2**, 123–126 (1835a). Reprinted in Babbage, vol. 3 (1989)
Babbage, C.: On the Economy of Machinery and Manufactures, 4th edn. Charles Knight, London (1835b). Reprinted in Babbage, vol. 8 (1989)
Babbage, C.: The Ninth Bridgewater Treatise: A Fragment. John Murray, London (1837a). 2nd edn. (1838). Reprinted in Babbage, vol. 9 (1989)
Babbage, C.: On the mathematical powers of the calculating engine (1837b). Printed in Babbage, vol. 3, pp. 15–61 (1989)
Babbage, C.: Passages from the Life of a Philosopher. Longman, London (1864)
Babbage, C.: The Works of Charles Babbage. William Pickering, London (1989). Edited in 9 volumes by M. Campbell-Kelly
Babbage, C., Herschel, J.F.W.: Preface. In: Memoirs of the Analytical Society, pp. i–xxii. Cambridge University Press, Cambridge (1813). Reprinted in Babbage, vol. 1 (1989)
Bachman, C.W.: The programmer as navigator. Commun. ACM **16**(11), 653–658 (1973)
Backus, J.W.: The IBM 701 speedcoding system. J. Assoc. Comput. Mach. **1**(1), 4–6 (1954)
Backus, J.W.: Automatic programming: properties and performance of FORTRAN systems I and II (1958). In: Mechanisation of Thought Processes: Proceedings of a Symposium held at the National Physical Laboratory, 24th, 25th, 26th and 27th November 1958, pp. 231–248. National Physical Laboratory, HMSO, London (1959)
Backus, J.W.: The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference (1959). In: Proceedings of the International Conference on Information Processing, Unesco, Paris, 15–20 June 1959, pp. 125–132. Unesco, Oldenbourg and Butterworths, Paris, Munich, London (1960)
Backus, J.W.: Programming in America in the 1950s—some personal impressions (1980). In: Metropolis, N., Howlett, J., Rota, G.-C. (eds.) A History of Computing in the Twentieth Century. Academic Press, San Diego (1980)
Backus, J.W.: ALGOL session: transcript of question and answer session (1981a). In: Wexelblat, R.L. (ed.) History of Programming Languages, p. 162. Academic Press, San Diego (1981)
Backus, J.W.: The history of FORTRAN I, II and III (1981b). In: Wexelblat, R.L. (ed.) History of Programming Languages, pp. 25–45. Academic Press, San Diego (1981)
Backus, J.W., Heising, W.P.: FORTRAN. IEEE Trans. Electron. Comput. **EC-13**(4), 382–385 (1964)
Bacon, F.: Instauratio Magna. London (1620)

Baily, F.: On Mr Babbage's new machine for calculating and printing mathematical and astronomical tables. Astron. Nachr. **46**, 409–422 (1823). Reprinted in Babbage, vol. 2, pp. 44–56 (1989)

Baker, C.L.: The PACT I coding system for the IBM Type 701. J. Assoc. Comput. Mach. **3**(4), 272–278 (1956)

Baker, F.T.: Chief programmer team management of production programming. IBM Syst. J. **11**(1), 56–73 (1972a)

Baker, F.T.: System quality through structured programming. In: AFIPS Proceedings of the 1972 Fall Joint Computer Conference. AFIPS Conference Proceedings, vol. 41, pp. 339–344. AFIPS Press, Montvale (1972b)

Barcan Marcus, R.: Interpreting quantification. Inquiry **5**, 252–9 (1962)

Barron, D.W., Buxton, J.N., Hartley, D.F., Nixon, E., Strachey, C.: The main features of CPL. Comput. J. **6**(2), 134–143 (1963)

Barwise, J., Etchemendy, J.: The Liar: An Essay on Truth and Circularity. Oxford University Press, London (1987)

Basili, V.: The role of experimentation in software engineering: past, present and future. In: ISCE'96: Proceedings of the 18th International Conference on Software Engineering, pp. 442–449. IEEE Comput. Soc., Los Alamitos (1996)

Bauer, F.L.: Appendix 5 to Naur (1981): Notes by F.L. Bauer. In: Wexelblat, R.L. (ed.) History of Programming Languages, pp. 127–130. Academic Press, San Diego (1981)

Bauer, F.L.: The Plankalkül of Konrad Zuse—revisited (2000). In: Rojas, R., Hashagen, U. (eds.) The First Computers—History and Architecture, pp. 277–293. MIT Press, Cambridge (2000)

Bauer, F.L., Wössner, H.: The "Plankalkül" of Konrad Zuse: A forerunner of today's programming languages. Commun. ACM **15**(1), 678–685 (1972)

Bauer, F.L., et al.: Letter from GAMM members to Prof. John Carr, III, President of ACM (1957). Reprinted in Bemer, pp. 160–161 (1969)

Bemer, R.W.: Automatic programming systems. Commun. ACM **2**(5), 16 (1959)

Bemer, R.W.: A politico-social history of Algol. In: Halpern, M.I., Shaw, C.J. (eds.) Annual Review in Automatic Programming, vol. 5, pp. 152–237. Pergamon, Elmsford (1969)

Bemer, R.W.: Computing prior to FORTRAN. Ann. Hist. Comput. **6**(1), 16–18 (1984)

Benington, H.D.: Production of large computer programs. In: Proceeding of the Symposium on Advanced Programming Methods for Digital Computers, pp. 15–28. Office of Naval Research, Arlington (1956)

Bennett, J.M., Prinz, D.G., Woods, M.L.: Interpretative sub-routines (1952). In: Forrester, J.W., Hamming, R.W. (eds.) Proceedings of the 1952 ACM National Meeting, Toronto, pp. 81–87 (1952)

Bergin, T.J., Gibson, R.G. (eds.): History of Programming Languages—II. ACM, New York (1996)

Berkeley, E.C.: Giant Brains, or Machines that Think. Wiley, New York (1949)

Berkeley, E.C.: The relations between symbolic logic and large-scale calculating machines. Science (New Ser.) **112**(2910), 395–399 (1950)

Bloch, R.M.: Mark I calculator (1947). In: Proceedings of a Symposium on Large-Scale Digital Calculating Machinery, 7–10 January 1947. The Annals of the Computation Laboratory of Harvard University, vol. XVI, pp. 23–30. The Navy Department Bureau of Ordnance and Harvard University, Harvard University Press, Cambridge (1947)

Bloch, R.M.: Programming Mark I (1999). In: Cohen, I.B., Welch, G.W. (eds.) Makin' Numbers: Howard Aiken and the Computer. MIT Press, Cambridge (1999)

Bloch, R.M., Campbell, R.V.D., Ellis, M.: The logical design of the Raytheon computer. Math. Tables Other Aids Comput. **III**(24), 286–295 (1948)

Boehm, B.W.: Software engineering. IEEE Trans. Comput. **C-25**(12), 1226–1241 (1976)

Boehm, B.W.: Verifying and validating software requirements and design specifications. IEEE Softw. **1**(1), 75–88 (1984)

Böhm, C., Jacopini, G.: Flow diagrams, Turing machines and languages with only two formation rules. Commun. ACM **9**(5), 366–371 (1966)

Boole, G.: An Investigation of the Laws of Thought. Macmillan, New York (1854)
Booth, A.D.: Relay computers (1949). In: Report of a Conference on High Speed Automatic Calculating Machines, 22–25 June 1949, pp. 27–30. Cambridge University Mathematical Laboratory, Cambridge (1950)
Booth, A.D.: Opening address (1960). In: Goodman, R. (ed.) Annual Review in Automatic Programming, I. Working Conference on Automatic Programming of Digital Computers held at Brighton, 1–3 April 1959, pp. 1–7. Pergamon, Elmsford (1960)
Boring, E.G.: Mind and mechanism. Am. J. Psychol. **LIX**(2), 184 (1946). Quoted in Edwards, p. 188 (1996)
Bosak, R., Clippinger, R.F., Dobbs, C., Goldfinger, R., Jasper, R.B., Keating, W., Kendrick, G., Sammet, J.E.: An information algebra. Commun. ACM **5**(4), 190–204 (1962)
Bowden, B.V. (ed.): Faster than Thought: A Symposium on Digital Computing Machines. Pitman, London (1953)
Briggs, L.J.: Impact of the war on science. Electr. Eng. **65**(1), 8–10 (1946)
Bromley, A.G.: Charles Babbage's Analytical Engine, 1838. Ann. Hist. Comput. **4**(3), 196–217 (1982)
Bromley, A.G.: Babbage's Analytical Engine plans 28 and 28a—the programmer's interface. IEEE Ann. Hist. Comput. **22**(4), 5–19 (2000)
Brooker, R.A., Wheeler, D.J.: Floating operations on the EDSAC. Math. Tables Other Aids Comput. **7**(41), 37–47 (1953)
Brooker, R.A., Gill, S., Wheeler, D.J.: The adventures of a blunder. Math. Tables Other Aids Comput. **6**(38), 112–113 (1952)
Burks, A.W.: Electronic computing circuits of the ENIAC. Proc. IRE **35**(8), 756–767 (1947)
Burks, A.W., Goldstine, H.H., von Neumann, J.: Preliminary discussion of the logical design of an electronic computing instrument. Technical Report, Institute of Advanced Study (1946). 2nd edn. (2 September 1947) reprinted in Aspray and Burks, pp. 97–142 (1987)
Butterfield, H.: The Whig Interpretation of History. Bell, London (1931)
Callon, M.: Society in the making: the study of technology as a tool for sociological analysis. In: Bijker, W.E., Hughes, T.P., Pinch, T.J. (eds.) The Sociological Construction of Technological Systems, pp. 83–103. MIT Press, Cambridge (1987)
Campbell-Kelly, M.: Programming the EDSAC: Early programming activity at the University of Cambridge. Ann. Hist. Comput. **2**(1), 7–36 (1980)
Campbell-Kelly, M.: From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry. MIT Press, Cambridge (2003)
Campbell-Kelly, M., Aspray, W.: Computer: A History of the Information Machine. Basic Books, New York (1996)
Campbell-Kelly, M., Williams, M.R. (eds.): The Moore School Lectures: Theory and Techniques for Design of Electronic Digital Computers. Charles Babbage Institute Reprint Series for the History of Computing, vol. 9. MIT Press, Cambridge (1985)
Carlyle, T.: Signs of the times. Edinb. Rev. (1829)
Carnap, R.: The Logical Syntax of Language. Routledge & Kegan Paul, London (1937)
Carnap, R.: Foundations of Logic and Mathematics. International Encyclopedia of Unified Science, vol. I(3). The University of Chicago Press, Chicago (1939)
Carnap, R.: Introduction to Semantics. Harvard University Press, Cambridge (1942)
Carpenter, B.E., Doran, R.W.: The other Turing machine. Comput. J. **20**(3), 269–279 (1977)
Ceruzzi, P.E.: Reckoners. Greenwood Press, Westport (1983)
Ceruzzi, P.E.: Crossing the divide: architectural issues and the emergence of the stored program computer, 1935–1955. IEEE Ann. Hist. Comput. **19**(1), 5–12 (1997)
Ceruzzi, P.E.: A History of Modern Computing. MIT Press, Cambridge (1998)
Ceruzzi, P.E.: A view from 20 years as a historian of computing. IEEE Ann. Hist. Comput. **23**(4), 49–55 (2001)
Ceruzzi, P.E.: A History of Modern Computing, 2nd edn. MIT Press, Cambridge (2003)
Chaitin, G.J.: Exploring Randomness. Springer, Berlin (2001)
Cherry, E.C.: A history of the theory of information (1950). In: Symposium on Information Theory: Report of Proceedings, pp. 22–43. Ministry of Supply, London (1950)

Church, A.: A set of postulates for the foundation of logic. Ann. Math. **33**(2), 346–366 (1932)
Church, A.: An unsolvable problem of elementary number theory. Am. J. Math. **58**(2), 345–363 (1936)
Cleave, J.P.: Application of formula translation to automatic coding (1960). In: Goodman, R. (ed.) Annual Review in Automatic Programming, I. Working Conference on Automatic Programming of Digital Computers held at Brighton, 1–3 April 1959, pp. 81–92. Pergamon, Elmsford (1960)
CODASYL Data Base Task Group: Data base task group report to the CODASYL programming language committee. Technical Report, General Electric Research and Development Center, October 1969
Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)
Cohen, I.B., Welch, G.W. (eds.): Makin' Numbers: Howard Aiken and the Computer. MIT Press, Cambridge (1999)
Colebrooke, H.T.: Address on presenting the Gold Medal of the Astronomical Society to Charles Babbage. Mem. Astron. Soc., pp. 509–512 (1825). Reprinted in Babbage, vol. 2, pp. 57–60 (1989)
Colilla, R.A., Sams, B.H.: Information structures for processing and retrieving. Commun. ACM **5**(1), 11–16 (1962)
Collier, B.: The little engines that could've: The calculating machines of Charles Babbage. Ph.D. thesis, Harvard University (1970)
Comrie, L.J.: The application of calculating machines to astronomical computing. Pop. Astron. **33**, 243–246 (1925)
Comrie, L.J.: The application of the Hollerith tabulating machine to Brown's tables of the moon. Mon. Not. R. Astron. Soc. **92**, 694–707 (1932a)
Comrie, L.J.: The National Almanac Office Burroughs machine. Mon. Not. R. Astron. Soc. **92**, 523–541 (1932b)
Comrie, L.J.: Inverse interpolation and scientific applications of the national accounting machine. Suppl. J. R. Stat. Soc. **3**(2), 87–114 (1936)
Comrie, L.J.: The application of Hollerith equipment to an agricultural investigation. Suppl. J. R. Stat. Soc. **4**(2), 210–224 (1937)
Copeland, B.J.: Computable numbers: a guide (2004a). In: Copeland, B.J. (ed.) The Essential Turing. Oxford University Press, London (2004b)
Copeland, B.J. (ed.): The Essential Turing. Oxford University Press, London (2004b)
Croarken, M.: Early Scientific Computing in Britain. Oxford University Press, London (1990)
Curry, H.B.: An analysis of logical substitution. Am. J. Math. **51**(3), 363–384 (1929)
Cutland, N.J.: Computability. Cambridge University Press, Cambridge (1980)
Dahl, O.-J., Nygaard, K.: SIMULA—a language for programming and description of discrete event systems. Introduction and user's manual. Technical Report 11, Norwegian Computing Centre (1965). 5th edn. (1967) available at http://www.edelweb.fr/Simula. Accessed 4 May 2008
Dahl, O.-J., Nygaard, K.: SIMULA—an ALGOL-based simulation language. Commun. ACM **9**(9), 671–678 (1966)
Dahl, O.-J., Nygaard, K.: Class and subclass declarations. In: Buxton, J.N. (ed.) Simulation Programming Languages, pp. 158–174. North-Holland, Amsterdam (1968)
Dahl, O.-J., Myhrhaug, B., Nygaard, K.: Common base language. Technical Report S-2, Norwegian Computing Centre (1968)
Dahl, O.-J., Dijkstra, E.W., Hoare, C.A.R.: Structured Programming. Academic Press, San Diego (1972)
Dasgupta, S.: Design Theory and Computer Science. Cambridge University Press, Cambridge (1991)
Daston, L.: Enlightenment calculations. Crit. Inq. **21**(1), 182–202 (1994)
Davies, D.W.: Corrections to Turing's universal machine. Undated manuscript, first published in Copeland (2004)
Davis, M.: Influences of mathematical logic on computer science (1988). In: Herken, R. (ed.) The Universal Turing Machine: A Half-Century Survey. Oxford University Press, London (1988)

Davis, M.: The Universal Computer. Norton, New York (2000)
De Millo, R.A., Lipton, R.J., Perlis, A.J.: Social processes and proofs of theorems and programs. Commun. ACM **22**(5), 271–280 (1979)
de Prony, G.F.C.M.R.: Notice sur les Grandes Tables. Didot, Paris (1824)
Department of Defense: Reference manual for the Ada programming language. United States Department of Defense, ANSI/MIL-STD-1815A-1983 (1983)
Department of Scientific and Industrial Research: A.C.E. the automatic computing machine. Electr. Eng. **18**(12), 372–373 (1946)
Diehm, I.C.: Computer aids to code checking (1952). In: Forrester, J.W., Hamming, R.W. (eds.) Proceedings of the 1952 ACM National Meeting, Toronto, pp. 19–21 (1952)
Dijkstra, E.W.: An attempt to unify the constituent concepts of serial program execution (1962a). In: Symbolic Languages in Data Processing: Proceedings of the Symposium Organized and Edited by the International Computation Centre, Rome, 26–31 March 1962, pp. 237–251. Gordon and Breach, New York (1962)
Dijkstra, E.W.: Some meditations on advanced programming (1962b). In: Popplewell, C.M. (ed.) Information Processing 1962: Proceedings of IFIP Congress, vol. 62, pp. 535–538. North-Holland, Amsterdam (1963)
Dijkstra, E.W.: On the design of machine independent programming languages (1963). In: Goodman, R. (ed.) Annual Review in Automatic Programming, vol. 3, pp. 27–42. Pergamon, Elmsford (1963)
Dijkstra, E.W.: Programming considered as a human activity (1965). In: Kalenich, W.A. (ed.) Information Processing 1965: Proceedings of IFIP Congress, vol. 65, pp. 213–217. Spartan Books, Washington (1965)
Dijkstra, E.W.: A constructive approach to the problem of program correctness. BIT Numer. Math. **8**, 174–186 (1968a)
Dijkstra, E.W.: Go to statement considered harmful. Commun. ACM **11**(3), 147–148 (1968b)
Dijkstra, E.W.: The structure of the "THE"-multiprogramming system. Commun. ACM **11**(5), 341–346 (1968c)
Dijkstra, E.W.: EWD249: notes on structured programming (1969a). Unpublished manuscript, available at http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF (accessed 4 May 2008). Circulated as Dijkstra (1970), and published with additional material as Dijkstra (1972)
Dijkstra, E.W.: Structured programming. In: Buxton, J.N., Randell, B. (eds.) Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969, pp. 84–88. NATO (1969b)
Dijkstra, E.W.: Notes on structured programming, 2nd edn. Technical Report 70-WSK-03, Department of Mathematics, Technical University Eindhoven, The Netherlands (1970)
Dijkstra, E.W.: Notes on structured programming (1972). In: Dahl, O.-J. et al. (eds.) Structured Programming, pp. 1–82. Academic Press, San Diego (1972)
Donaldson, J.R.: Structured programming. Datamation **19**(12), 52–54 (1973)
Dubbey, J.M.: The Mathematical Work of Charles Babbage. Cambridge University Press, Cambridge (1978)
Earley, J.: Toward an understanding of data structure. Commun. ACM **14**(10), 617–627 (1971)
Eckert, W.J.: Punched Card Methods in Scientific Calculation. The Thomas J. Watson Astronomical Computing Bureau, Columbia University, New York (1940)
Eckert, J.P. Jr.: Disclosure of magnetic calculating machine. Typescript dated 29 January 1944. Reprinted in Lukoff, pp. 207–209 (1979)
Eckert, J.P. Jr.: A preview of a digital computing machine (1946). In: Campbell-Kelly, M., Williams, M.R. (eds.) The Moore School Lectures: Theory and Techniques for Design of Electronic Digital Computers. Charles Babbage Institute Reprint Series for the History of Computing, vol. 9, pp. 109–126. MIT Press, Cambridge (1985). Lecture delivered 15 July 1946
Eckert, J.P., Mauchly, J.W., Warren, S.R.: PY Summary Report No. 1, 31 March 1945. Quoted in Aspray, p. 38 (1990b)
Edgerton, D.: The Shock of the Old: Technology and Global History Since 1900. Profile Books, London (2006)

Edwards, P.N.: The Closed World: Computers and the Politics of Discourse in Cold War America. MIT Press, Cambridge (1996)
Elgot, C.C.: On single vs. triple address computing machines. J. Assoc. Comput. Mach. **1**(3), 119–123 (1954)
Elgot, C.C., Robinson, A.: Random-access stored-program machines, an approach to programming languages. J. Assoc. Comput. Mach. **11**(4), 365–399 (1964)
ENIAC: ENIAC progress report, 31 December 1943. Quoted in Stern (1981)
Feldman, J.A.: A formal semantics for computer languages and its application in a compiler-compiler. Commun. ACM **9**(1), 3–9 (1966)
Felton, G.E.: Assembly, interpretive and conversion programs for PEGASUS (1960). In: Goodman, R. (ed.) Annual Review in Automatic Programming, I. Working Conference on Automatic Programming of Digital Computers held at Brighton, 1–3 April 1959, pp. 32–57. Pergamon, Elmsford (1960)
Fetzer, J.H.: Program verification: the very idea. Commun. ACM **31**(9), 1048–1063 (1988)
Floyd, R.W.: The syntax of programming languages—a survey. IEEE Trans. Electron. Comput. **EC-13**(4), 346–353 (1964)
Floyd, R.W.: Assigning meanings to programs. In: Schwartz, J.T. (ed.) Mathematical Aspects of Computer Science. Proceedings of Symposia in Applied Mathematics, vol. XIX, pp. 19–32. American Mathematical Society, Providence (1967)
Floyd, R.W.: Toward interactive design of correct programs (1971). In: Freiman, C.V. (ed.) Information Processing 71: Proceedings of the IFIP Congress, vol. 71, pp. 7–10. North-Holland, Amsterdam (1972)
Frege, G.: Begriffsschrift, Eine der Arithmetischen Nachgebildete Formelsprache des Reinen Denkens. Halle (1879). Translated into English as "Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought" in van Heijenoort, pp. 1–82 (1967)
Fritz, W.B.: ENIAC—a problem solver. IEEE Ann. Hist. Comput. **16**(1), 25–45 (1994)
Gandy, R.: The confluence of ideas in 1936 (1988). In: Herken, R. (ed.) The Universal Turing Machine: A Half-Century Survey, pp. 55–111. Oxford University Press, London (1988)
Garwick, J.V.: The definition of programming languages by their compilers (1964). In: Steel, T.B. (ed.) Formal Language Description Languages for Computer Programming, pp. 139–147. North-Holland, Amsterdam (1966)
Gerhart, S.L., Yelowitz, L.: Observations of fallibility in applications of modern programming methodologies. IEEE Trans. Softw. Eng. **SE-2**(3), 195–207 (1976)
Gill, S.: The diagnosis of mistakes in programmes on the EDSAC. Proc. R. Soc. Lond. Ser. A, Math. Phys. Sci. **206**, 538–554 (1951)
Gill, S.: Getting programmes right. In: Automatic Digital Computation, pp. 289–292. National Physical Laboratory, HMSO, London (1953). Reprinted in Williams and Campbell-Kelly, pp. 209–498 (1989)
Gill, S.: Current theory and practice of automatic programming. Comput. J. **2**(3) (1959)
Gillies, D. (ed.): Revolutions in Mathematics. Oxford University Press, London (1992)
Gillies, D., Zheng, Y.: Dynamic interactions with the philosophy of mathematics. Theoria **16**(3), 437–459 (2001)
Gilmore, P.C.: An abstract computer with a Lisp-like machine language without a label operator. In: Braffort, P., Hirschberg, D. (eds.) Computer Programming and Formal Systems, pp. 71–86. North-Holland, Amsterdam (1963)
Giloi, W.K.: Konrad Zuse's Plankalkül: the first high-level, "non von Neumann" programming language. IEEE Ann. Hist. Comput. **19**(2), 17–24 (1997)
Gödel, K.: Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme I. Monatshefte Math. Phys. **38**, 173–198 (1931). English translation in Gödel, pp. 145–195 (1986). Page references to this translation
Gödel, K.: On undecidable propositions of formal mathematical systems. Mimeographed lecture notes, taken by S.C. Kleene and J. Barkley Rosser (1934). Reprinted in Gödel, pp. 346–369 (1986)
Gödel, K.: Collected Works, Publications 1929–1936, vol. I. Oxford University Press, London (1986). S. Feferman et al. (eds.)

Goldberg, A., Kay, A.: Smalltalk-72 instruction manual. Technical Report SSL 76-6, Xerox Palo Alto Research Center (1976)
Goldstine, H.H.: The Computer from Pascal to Von Neumann. Princeton University Press, Princeton (1972)
Goldstine, H.H., Goldstine, A.: The electronic numerical integrator and computer (ENIAC). Math. Tables Other Aids Comput. **II**(15), 97–110 (1946)
Goldstine, H.H., von Neumann, J.: On the principles of large scale computing machines (1946). Unpublished. Reproduced in Aspray and Burks, pp. 317–348 (1987)
Goldstine, H.H., von Neumann, J.: Planning and coding problems for an electronic computing instrument, Part II, vol. 1. Technical Report, Institute of Advanced Study (1947). Reprinted in Aspray and Burks, pp. 151–222 (1987)
Goldstine, H.H., von Neumann, J.: Planning and coding problems for an electronic computing instrument, Part II, vol. 3. Technical Report, Institute of Advanced Study (1948). Reprinted in Aspray and Burks, pp. 286–306 (1987)
Good, I.J.: Discussion contribution (1951). In: Manchester University Computer, Inaugural Conference, July 1951, p. 192 (1951). Reprinted in Williams and Campbell-Kelly, pp. 165–206 (1989)
Goodenough, J.B., Gerhart, S.L.: Toward a theory of test data selection. IEEE Trans. Softw. Eng. **SE-1**(2), 156–173 (1975)
Gordon, G.: A general purpose systems simulation program. In: Proceedings of the Eastern Joint Computer Conference, pp. 87–104 (1961)
Gordon, G.: GPSS session: transcript of presentation (1981). In: Wexelblat, R.L. (ed.) History of Programming Languages, pp. 426–434. Academic Press, San Diego (1981)
Gorn, S.: Some basic terminology connected with mechanical languages and their processors. Commun. ACM **4**(8), 336–339 (1961)
Gorn, S.: Theory of mechanical languages. Commun. ACM **5**(1), 62 (1962)
Gorn, S.: Summary remarks and general discussion. Commun. ACM **7**(2), 133–136 (1964). ACM Mechanical Languages Workshop, August 1963
Grattan-Guinness, I.: Work for the hairdressers: the production of de Prony's logarithmic and trigonometric tables. Ann. Hist. Comput. **12**(3), 177–185 (1990)
Green, J., Shapiro, R.M., Helt, F.R. Jr., Franciotti, R.G., Theil, E.H.: Remarks on ALGOL and symbol manipulation. Commun. ACM **2**(9), 25–27 (1959)
Grier, D.A.: When Computers Were Human. MIT Press, Cambridge (2005)
Hamblin, C.L.: Computer languages. Aust. J. Sci. **20**(5), 135–139 (1957)
Hamblin, C.L.: GEORGE IA and II: a semi-translation programming scheme for DEUCE. Programming and operation manual. Unpublished report, University of New South Wales (1958)
Hartree, D.R.: Letter to the Times (1946a)
Hartree, D.R.: The ENIAC, an electronic computing machine. Nature **158**(4015), 500–506 (1946b)
Hartree, D.R.: Calculating Instruments and Machines. University of Illinois Press, Champaign (1949)
Harvard: Proceedings of a Symposium on Large-Scale Digital Calculating Machinery, 7–10 January 1947. The Annals of the Computation Laboratory of Harvard University, vol. XVI. The Navy Department Bureau of Ordnance and Harvard University, Harvard University Press, Cambridge (1947)
Heide, L.: Punched-Card Systems and the Early Information Explosion 1880–1945. Johns Hopkins University Press, Baltimore (2009)
Heims, S.J.: John Von Neumann and Norbert Wiener: From Mathematics to the Technologies of Life and Death. MIT Press, Cambridge (1980)
Henderson, P., Snowdon, R.: An experiment in structured programming. BIT Numer. Math. **12**, 38–53 (1972)
Hilbert, D.: Über das Unendliche. Math. Ann. **95**, 161–190 (1926). Translated as "On the infinite" in van Heijenoort, pp. 367–392 (1967)
Hilbert, D., Ackermann, W.: Grundzüge der Theoretischen Logik. Springer, Berlin (1928)
Hoare, C.A.R.: Record handling. In: Genuys, F. (ed.) Programming Languages, pp. 291–347. Academic Press, San Diego (1968)

Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
Hoare, C.A.R.: Proof of a program: FIND. Commun. ACM **14**(1), 39–45 (1971)
Hoare, C.A.R.: A note on the for statement. BIT Numer. Math. **12**, 334–341 (1972a)
Hoare, C.A.R.: Notes on data structuring (1972b). In: Structured Programming, pp. 83–173. Academic Press, San Diego (1972)
Hoare, C.A.R.: Programming is an engineering profession. Technical Report PRG-27, Programming Research Group, Oxford University (1982)
Hoare, C.A.R.: How did software get so reliable without proof? In: Gaudel, M.-C., Woodcock, J. (eds.) FME'96: Industrial Benefit and Advances in Formal Methods. Proceedings of the Third International Symposium of Formal Methods Europe. Lecture Notes in Computer Science, vol. 1051, pp. 1–17. Springer, Berlin (1996)
Hobbes, T.: Leviathan. Andrew Crooke, London (1651)
Hodges, A.: Alan Turing: The Engima. Vintage, New York (1983)
Hopper, G.M.: The education of a computer (1952). In: Proceedings of the 1952 ACM National Meeting (Pittsburgh), pp. 243–249. ACM, New York (1952)
Hopper, G.M.: Automatic programming: present status and future trends (1959). In: Mechanisation of Thought Processes: Proceedings of a Symposium held at the National Physical Laboratory, 24th, 25th, 26th and 27th November 1958, pp. 155–194. National Physical Laboratory, HMSO, London (1959)
Huskey, H.D.: The status of high-speed digital computing systems. Mech. Eng. **70**(12), 975–978 (1948)
Huskey, H.D.: Semiautomatic instruction on the Zephyr (1951). In: Proceedings of a Second Symposium on Large-Scale Digital Calculation Machinery, 13–16 September 1949. The Annals of the Computation Laboratory of Harvard University, vol. XXVI, pp. 83–90. The Navy Department Bureau of Ordnance and Harvard University, Harvard University Press, Cambridge (1951)
IBM: Preliminary Report: Specification for the IBM Mathematical FORmula TRANSlating System, FORTRAN. International Business Machines Corporation, 590 Madison Avenue, New York, 22, New York. Programming Research Group, Applied Science Division (1954)
IBM: Programmer's Reference Manual: The FORTRAN Automatic Coding System for the IBM 704 EDPM. International Business Machines Corporation, 590 Madison Ave., New York 22, NY. 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 (1956)
IBM: Reference Manual: FORTRAN II for the IBM 704 Data Processing System. International Business Machines Corporation, 590 Madison Ave., New York 22, NY (1958)
Ingalls, D.H.H.: The Smalltalk-76 system design and implementation. In: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 9–16 (1978)
International Computation Center: Symbolic Languages in Data Processing: Proceedings of the Symposium Organized and Edited by the International Computation Centre, Rome, 26–31 March 1962. Gordon and Breach, New York (1962)
Irons, E.T.: A syntax directed compiler for ALGOL 60. Commun. ACM **4**(1), 51–55 (1961)
Isaac, E.J.: Machine aids to coding (1952). In: Forrester, J.W., Hamming, R.W. (eds.) Proceedings of the 1952 ACM National Meeting, Toronto, pp. 17–19 (1952)
Jardine, L., Silverthorne, M. (eds.): The New Organon. Cambridge University Press, Cambridge (2000)
Jones, R.F.: Science and language in England of the mid-seventeenth century. J. Engl. Ger. Philol. **31**, 315–331 (1932)
Jones, C.B.: The early search for tractable ways of reasoning about programs. IEEE Ann. Hist. Comput. **25**(2), 26–49 (2003)
Katz, C.: Systems of debugging automatic coding. In: Automatic Coding: Proceedings of the Symposium, 24–25 January 1957. Franklin Institute, Philadelphia (1957). J. Franklin Inst., Monograph No. 3, pp. 17–27

Kay, A.C.: The early history of Smalltalk (1996). In: Bergin, T.J., Gibson, R.G. (eds.) History of Programming Languages—II, pp. 511–579. ACM, New York (1996)
Kay, A., Goldberg, A.: Personal dynamic media. Computer **10**(3), 31–41 (1977)
Kistermann, F.W.: The invention and development of the Hollerith punched card. Ann. Hist. Comput. **13**(3), 245–259 (1991)
Kistermann, F.W.: Hollerith punched card system development (1905–1913). IEEE Ann. Hist. Comput. **27**(1), 56–66 (2005)
Kleene, S.C.: A theory of positive integers in formal logic. Part I. Am. J. Math. **57**(1), 153–173 (1935a)
Kleene, S.C.: A theory of positive integers in formal logic. Part II. Am. J. Math. **57**(2), 219–244 (1935b)
Kleene, S.C.: General recursive functions of natural numbers. Math. Ann. **112**(5), 727–742 (1936a)
Kleene, S.C.: λ-definability and recursiveness. Duke Math. J. **2**(2), 340–353 (1936b)
Knuth, D.E.: Von Neumann's first computer program. ACM Comput. Surv. **2**(4), 247–260 (1970)
Knuth, D.E.: Structured programming with go to statements. ACM Comput. Surv. **6**(4), 260–301 (1974)
Knuth, D.E., McNeley, J.L.: SOL—a symbolic language for general-purpose systems simulation. IEEE Trans. Electron. Comput. **EC-13**(4), 401–408 (1964)
Knuth, D.E., Trabb Pardo, L.: The early development of programming languages (1980). In: Metropolis, N., Howlett, J., Rota, G.-C. (eds.) A History of Computing in the Twentieth Century. Academic Press, San Diego (1980)
Kuhn, T.S.: The Structure of Scientific Revolutions. University of Chicago Press, Chicago (1962)
Lakatos, I. (ed.): Problems in the Philosophy of Mathematics. North-Holland, Amsterdam (1967)
Lakatos, I.: Falsification and the methodology of scientific research programmes. In: Lakatos, I., Musgrave, A. (eds.) Criticism and the Growth of Knowledge, pp. 91–196. Cambridge University Press, Cambridge (1970)
Landin, P.J.: The mechanical evaluation of expressions. Comput. J. **6**(4), 308–320 (1964)
Landin, P.J.: A correspondence between ALGOL 60 and Church's lambda-notation: Part I. Commun. ACM **8**(2), 89–101 (1965a)
Landin, P.J.: A correspondence between ALGOL 60 and Church's lambda-notation: Part II. Commun. ACM **8**(3), 158–165 (1965b)
Landin, P.J.: The next 700 programming languages. Commun. ACM **9**(3), 157–166 (1966)
Laning, J.H. Jr., Zierler, N.: A program for translation of mathematical equations with Whirlwind I. Engineering Memorandum E-364, Instrumentation Laboratory, Massachusetts Institute of Technology (1954)
Lardner, D.: Babbage's calculating engine. Edinb. Rev. **59**, 263–327 (1834). Reprinted in Babbage, vol. 2 (1989)
Larman, C., Basili, V.R.: Iterative and incremental development: a brief history. Computer **36**(6), 47–56 (2003)
Ledgard, H.F.: The case for structured programming. BIT Numer. Math. **14**, 45–57 (1974)
Lefort, M.F.: Description des grandes tables logarithmiques et trigonométriques calculées au Bureau du cadastre sous la direction de Prony et exposition des methodes et procédés mis en usage pour leur construction. Ann. Obs. Imp. Paris **4**, 123–150 (1858)
Lewis, C.I.: A Survey of Symbolic Logic. University of California Press, Berkeley (1918)
Liskov, B.: Report of session on structured programming. ACM SIGPLAN Not. **8**(9), 5–10 (1973)
Liskov, B., Zilles, S.: Programming with abstract data types. ACM SIGPLAN Not. **9**(4), 50–59 (1974)
Liskov, B.H., Zilles, S.N.: Specification techniques for data abstractions. IEEE Trans. Softw. Eng. **SE-1**(1), 7–19 (1975)
Lombardi, L.: Theory of files. In: Proceedings of the Eastern Joint Computer Conference, pp. 137–141 (1960)
Lovelace, A.A.: Sketch of the Analytical Engine Invented by Charles Babbage Esq. By L.F. Menabrea of Tutin, Officer of the Military Engineers. Scientific Memoirs vol. 3, pp. 666–731 (1843). With notes upon the memoir by the translator. Reprinted in Babbage, vol. 3 (1989)

Lucas, P.: Formal definition of programming languages and systems (1972). In: Freiman, C.V. (ed.) Information Processing 71: Proceedings of the IFIP Congress, vol. 71, pp. 291–297. North-Holland, Amsterdam (1972)
Lucas, P., Walk, K.: On the formal description of PL/I. Annu. Rev. Autom. Program. **6**(3), 105–182 (1969)
Mahoney, M.S.: The history of computing in the history of technology. Ann. Hist. Comput. **10**, 113–125 (1988)
Mahoney, M.S.: Cybernetics and information technology. In: Olby, R.C. (ed.) Companion to the History of Modern Science. Chapman and Hall, London (1989). Chap. 34
Mahoney, M.S.: Computer science: the search for a mathematical theory. In: Krige, J., Pestre, D. (eds.) Science in the Twentieth Century, pp. 617–634. Harwood Academic, Reading (1997)
Manna, Z., Waldinger, R.J.: Toward automatic program synthesis. Commun. ACM **14**(3), 151–165 (1971)
Manna, Z., Waldinger, R.J.: The logic of computer programming. IEEE Trans. Softw. Eng. **SE-4**(3), 199–229 (1978)
Marcus, M., Akera, A.: Exploring the architecture of an early machine: the historical relevance of the ENIAC machine architecture. IEEE Ann. Hist. Comput. **18**(1), 17–24 (1996)
Markowitz, H.M., Hauser, B., Kerr, H.W.: SIMSCRIPT—A Simulation Programming Language. Prentice-Hall, New York (1963)
Martin, C.D.: The myth of the awesome thinking machine. Commun. ACM **36**(4), 120–133 (1993)
Masani, N., Randell, B., Ferry, D.K., Saeks, R.: The Wiener memorandum on the mechanical solution of partial differential equations. Ann. Hist. Comput. **9**(2), 183–197 (1987)
Mauchly, J.W.: The use of high speed vacuum tubes for calculating (1942). Unpublished memorandum, quoted in Stern, p. 56 (1981)
Mauchly, J.W.: Code and control II: machine design and instruction codes (1946). In: Campbell-Kelly, M., Williams, M.R. (eds.) The Moore School Lectures: Theory and Techniques for Design of Electronic Digital Computers. Charles Babbage Institute Reprint Series for the History of Computing, vol. 9, pp. 453–461. MIT Press, Cambridge (1985). Lecture delivered 9 August 1946
Mauchly, J.W.: Preparation of problems for EDVAC-type machines (1947). In: Proceedings of a Symposium on Large-Scale Digital Calculating Machinery, 7–10 January 1947. The Annals of the Computation Laboratory of Harvard University, vol. XVI, pp. 203–207. The Navy Department Bureau of Ordnance and Harvard University, Harvard University Press, Cambridge (1947)
Mauchly, J.W.: Suggested Form for "BINAC BRIEF CODE" (1949). Unpublished notes, printed in Schmitt, pp. 17–18 (1988)
McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part I. Commun. ACM **3**(4), 184–195 (1960)
McCarthy, J.: A basis for a mathematical theory of computation, preliminary report. In: Proceedings of the Western Joint Computer Conference, pp. 225–238 (1961)
McCarthy, J.: Towards a mathematical science of computation (1962). In: Popplewell, C.M. (ed.) Information Processing 1962: Proceedings of IFIP Congress, vol. 62, pp. 21–28. North-Holland, Amsterdam (1963)
McCarthy, J.: A basis for a mathematical theory of computation (1963a). In: Braffort, P., Hirschberg, D. (eds.) Computer Programming and Formal Systems, pp. 33–70. North-Holland, Amsterdam (1963). Corrected and extended version of McCarthy (1961)
McCarthy, J.: General discussion (1963b). Quoted in Gorn (1964)
McCarthy, J.: Problems in the theory of computation (1965). In: Kalenich, W.A. (ed.) Information Processing 1965: Proceedings of IFIP Congress, vol. 65, pp. 219–222. Spartan Books, Washington (1965)
McCarthy, J.: A formal description of a subset of ALGOL (1964). In: Steel, T.B. (ed.) Formal Language Description Languages for Computer Programming, pp. 1–12. North-Holland, Amsterdam (1966)
McCarthy, J.: History of LISP (1981). In: Wexelblat, R.L. (ed.) History of Programming Languages, pp. 173–185. Academic Press, San Diego (1981)

McCartney, S.: ENIAC: The Triumphs and Tragedies of the World's First Computer. Berkley Books, New York (1999)
McCracken, D.: Revolution in programming: an overview. Datamation **19**(12), 50–52 (1973). Reprinted in Yourdon (1979)
McCracken, D.D., Jackson, M.A.: Life cycle concept considered harmful. Softw. Eng. Notes **7**(2), 29–32 (1982)
McCulloch, W.S.: Contribution to discussion following von Neumann 1948 (1948). In: Jeffress, L. (ed.) Cerebral Mechanisms in Behavior, pp. 32–41. Wiley, New York (1951)
McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophys. **5**, 115–133 (1943)
Menabrea, L.F.: Notions sur la machine analytique de M. Charles Babbage. Bibl. Univers. Genève **41**, 352–376 (1842). Translated in Lovelace, pp. 667–690 (1843)
Metropolis, N., Worlton, J.: A trilogy of errors in the history of computing. Ann. Hist. Comput. (1980)
Miller, J.C.P.: Remarks on checking (1949). In: Report of a Conference on High Speed Automatic Calculating Machines, 22–25 June 1949, pp. 123–124. University Mathematical Laboratory, Cambridge (1950)
Mills, H.D.: Software development. IEEE Trans. Softw. Eng. **SE-2**(4), 265–273 (1976)
Mills, H.D.: Structured programming: retrospect and prospect. IEEE Softw. **3**(6), 58–66 (1986)
Mooers, C.N.: Code and control IV: example of a three-address code and the use of "stop order tags" (1946). In: Campbell-Kelly, M., Williams, M.R. (eds.) The Moore School Lectures: Theory and Techniques for Design of Electronic Digital Computers. Charles Babbage Institute Reprint Series for the History of Computing, vol. 9, pp. 465–484. MIT Press, Cambridge (1985). Lecture delivered 12 August 1946
Morris, C.W.: Foundations of the Theory of Signs. International Encyclopedia of Unified Science, vol. I(2). University of Chicago Press, Chicago (1938)
Morris, J.H. Jr.: Protection in programming languages. Commun. ACM **16**(1), 15–21 (1973)
Naur, P. (ed.): ALGOL-Bulletin No. 1. Regnecentralen, Copenhagen (1959)
Naur, P. (ed.): ALGOL-Bulletin No. 15. Regnecentralen, Copenhagen (1962)
Naur, P.: Proof of algorithms by general snapshots. BIT Numer. Math. **6**, 310–316 (1966)
Naur, P.: Programming by action clusters. BIT Numer. Math. **9**, 250–258 (1969)
Naur, P.: The European side of the last phase of the development of ALGOL 60 (1981). In: Wexelblat, R.L. (ed.) History of Programming Languages, pp. 92–139. Academic Press, San Diego (1981)
Naur, P., Randell, B. (eds.): Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968. Scientific Affairs Division, NATO, Brussels (1969)
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., Woodger, M.: Report on the algorithmic language ALGOL 60. Commun. ACM **3**(5), 299–314 (1960)
Newell, A., Simon, H.A.: The logic theory machine: a complex information processing system. IRE Trans. Inf. Theory **IT-2**(3), 61–79 (1956)
Newell, A., Tonge, F.M.: An introduction to Information Processing Language V. Commun. ACM **3**(4), 205–211 (1960)
Newman, M.H.A.: General principles of the design of all-purpose computing machines. Proc. R. Soc. Lond. Ser. A, Math. Phys. Sci. **195**, 271–274 (1949). Record of a discussion held on 4 March 1948
Nygaard, K., Dahl, O.-J.: The development of the SIMULA languages (1981). In: Wexelblat, R.L. (ed.) History of Programming Languages, pp. 439–480. Academic Press, San Diego (1981)
Oettinger, A.G.: Programming a digital computer to learn. Philos. Mag., 7th. Ser. **43**, 1243–1263 (1952)
Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12), 1053–1058 (1972)

Patterson, G.W.: Logical syntax and transformation rules (1949). In: Proceedings of a Second Symposium on Large-Scale Digital Calculation Machinery, 13–16 September 1949. The Annals of the Computation Laboratory of Harvard University, vol. XXVI, pp. 125–133. The Navy Department Bureau of Ordnance and Harvard University, Harvard University Press, Cambridge (1951)

Peacock, G.: A Treatise on Algebra. Cambridge University Press, Cambridge (1830)

Peláez, E.: The stored-program computer: Two conceptions. Soc. Stud. Sci. **29**(3), 359–389 (1999)

Perlis, A.J., Samelson, K.: Preliminary report—international algebraic language. Commun. ACM **1**(12), 8–22 (1958)

Pickering, A.: The Mangle of Practice: Time, Agency and Science. University of Chicago Press, Chicago (1995)

Pickering, A.: Cybernetics and the mangle: Ashby, Beer and Pask. Soc. Stud. Sci. **32**(3), 413–437 (2002)

Playfair, J.: On the arithmetic of impossible quantities. Philos. Trans. R. Soc. Lond. **68**, 318–343 (1778)

Popplewell, C.M. (ed.): Information Processing 1962: Proceedings of IFIP Congress, vol. 62. North-Holland, Amsterdam (1963)

Post, E.L.: Finite combinatory processes—formulation 1. J. Symb. Log. **1**(3), 103–105 (1936)

Post, E.L.: Recursive unsolvability of a problem of Thue. J. Symb. Log. **12**, 1–11 (1947)

Pratt, V.: Thinking Machines: The Evolution of Artificial Intelligence. Blackwell Sci., Oxford (1987)

Prinz, D.G.: Robot chess. Res. Sci. Appl. Ind. **5**, 261–266 (1952)

Puyen, J., Vauquois, B.: A propos d'un langage universel (1960). In: Proceedings of the International Conference on Information Processing, Unesco, Paris, 15–20 June 1959, pp. 132–137. Unesco, Oldenbourg and Butterworths, Paris, Munich, London (1960)

Pycior, H.M.: Symbols, Impossible Numbers, and Geometric Entanglements. Cambridge University Press, Cambridge (1997)

Rabinowitz, I.N.: Report on the algorithmic language FORTRAN II. Commun. ACM **5**(6), 327–337 (1962)

Radin, G., Rogoway, H.P.: NPL: highlights of a new programming language. Commun. ACM **8**(1), 9–17 (1965)

Randell, B.: On Alan Turing and the origins of digital computers. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 7, pp. 3–20. Edinburgh University Press, Edinburg (1972)

Randell, B.: Facing up to faults. Comput. J. **43**(2), 95–106 (2000)

Redmond, K.C., Smith, T.M.: Project Whirlwind: The History of a Pioneer Computer. Digital Press, Paris (1980)

Rees, G., Wakely, M. (eds.): The Instauratio Magna Part II: Novum Organum and Associated Texts. The Oxford Francis Bacon, vol. XI. Clarendon Press, Oxford (2004)

Renwick, W.: The E.D.S.A.C. demonstration (1949). In: Report of a Conference on High Speed Automatic Calculating Machines, 22–25 June 1949, pp. 21–26. University Mathematical Laboratory, Cambridge (1950)

Robinson, C.: Automatic programming on DEUCE (1960). In: Goodman, R. (ed.) Annual Review in Automatic Programming, I. Working Conference on Automatic Programming of Digital Computers held at Brighton, 1–3 April 1959, pp. 111–126. Pergamon, Elmsford (1960)

Robinson, H., Hall, P., Hovenden, F., Rachel, J.: Postmodern software development. Comput. J. **41**(6), 363–375 (1998)

Rochester, N., Goldfinger, R.: The special issue on computer languages—editorial. IEEE Trans. Electron. Comput. **EC-13**(4), 343 (1964)

Rojas, R.: The architecture of Konrad Zuse's early computing machines (2000). In: Rojas, R., Hashagen, U. (eds.) The First Computers—History and Architecture, pp. 237–261. MIT Press, Cambridge (2000)

Rope, C.: ENIAC as a stored-program computer: a new look at the old records. IEEE Ann. Hist. Comput. **29**(4), 82–87 (2007)

Rosen, S.: Programming systems and languages: a historical survey. In: Proceedings of the 1964 Spring Joint Computer Conference. AFIPS Conference Proceedings, vol. 25, pp. 1–15. Spartan Books, Inc., Cleaver-Hume Press, Baltimore, London (1964)
Rosen, S.: Programming Systems and Languages. McGraw-Hill, New York (1967)
Rosen, S.: Programming systems and languages 1965–1975. Commun. ACM **15**(7), 591–600 (1972)
Rosenbloom, P.C.: The Elements of Mathematical Logic. Dover, New York (1950)
Rosenblueth, A., Wiener, N., Bigelow, J.: Behaviour, purpose and teleology. Philos. Sci. **10**(1), 18–24 (1943)
Ross, D.T.: A generalized technique for symbol manipulation and numerical calculation. Commun. ACM **4**(3), 147–150 (1961)
Ross, D.T.: Design and production in software engineering: discussion contribution (1968). In: Naur, P., Randell, B. (eds.) Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968, p. 32. Scientific Affairs Division, NATO, Brussels (1969)
Ross, D.T., Rodriguez, J.E.: Theoretical foundations for the computer-aided design system. In: Proceedings of the 1963 Spring Joint Computer Conference. AFIPS Conference Proceedings, vol. 23, pp. 305–322. Spartan Books, Inc., Cleaver-Hume Press, Baltimore, London (1963)
Rosser, J.B.: Highlights of the history of the lambda-calculus. Ann. Hist. Comput. **6**(4), 337–349 (1984)
Ryder, B., Hailpern, B. (eds.): Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages. ACM, New York (2007)
Samelson, K.: Appendix 7 to Naur (1981): Comments by K. Samelson, 1978 December 1 (1981). In: Wexelblat, R.L. (ed.) History of Programming Languages, pp. 131–134. Academic Press, San Diego (1981)
Sammet, J.E.: Basic elements of COBOL 61. Commun. ACM **5**(5), 237–253 (1962)
Sammet, J.E.: Programming Languages: History and Fundamentals. Prentice-Hall, New York (1969)
Schick, G.J., Wolverton, R.W.: An analysis of competing software reliability models. IEEE Trans. Softw. Eng. **SE-4**(2), 104–120 (1978)
Schmitt, W.F.: The UNIVAC SHORT CODE. Ann. Hist. Comput. **10**(1), 7–18 (1988)
Schönfinkel, M.: Über die Bausteine der mathematischen Logik. Math. Ann. (1924). Translated as "On the building blocks of mathematical logic" in van Heijenoort, pp. 357–366 (1967)
Shannon, C.E.: A symbolic analysis of relay and switching circuits. Trans. Am. Inst. Electr. Eng. **57**, 713–723 (1938)
Shannon, C.E.: Programming a computer for playing chess. Philos. Mag., 7th Ser. **41**, 256–275 (1950). Paper first presented at National IRE Convention, 9 March 1949
Shannon, C.E.: Computers and automata. Proc. IRE **41**(10), 1234–1241 (1953)
Shapiro, S.: Splitting the difference: the historical necessity of synthesis in software engineering. IEEE Ann. Hist. Comput. **19**(1), 20–54 (1997)
SHARE: News and notices. Commun. ACM **1**(3), 17 (1958a)
SHARE: News and notices. Commun. ACM **1**(4), 16–17 (1958b)
Shaw, C.J.: Jovial—a programming language for real-time command systems (1963). In: Goodman, R. (ed.) Annual Review in Automatic Programming, vol. 3, pp. 53–119. Pergamon, Elmsford (1963)
Shaw, M.: Prospects for an engineering discipline of software. IEEE Softw. **7**(6), 15–24 (1990)
Sheridan, P.B.: The Fortran automatic coding system. In: Summer Institute for Symbolic Logic, Cornell University, pp. 452–453. American Mathematical Society, Providence (1957)
Sheridan, P.B.: The arithmetic translator-compiler of the IBM FORTRAN automatic coding system. Commun. ACM **2**(2), 9–21 (1959)
Shoch, J.F.: An overview of the programming language Smalltalk-72. ACM SIGPLAN Not. **14**(9), 64–73 (1979)
Skolem, T.: Begründung der elementaren Arithmetik durch die rekurrierende Denkweise ohne Anwendung scheinbarer Veränderlichen mit unendlichem Ausdehnungsbereich. Skr. Utg. Videnskapsselskapet Kristiania, I. **6**, 1–38 (1923). 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, pp. 302–333 (1967)
Soare, R.I.: Computability and recursion. Bull. Symb. Log. **2**(3), 284–321 (1996)
Speiser, A.P.: Konrad Zuse's Z4: architecture, programming and modifications at ETH Zurich (2000). In: Rojas, R., Hashagen, U. (eds.) The First Computers—History and Architecture, pp. 263–276. MIT Press, Cambridge (2000)
Steel, T.B. Jr.: UNCOL: the myth and the fact. In: Goodman, R. (ed.) Annual Review in Automatic Programming, vol. 2, pp. 325–344. Pergamon, Elmsford (1961)
Steel, T.B. (ed.): Formal Language Description Languages for Computer Programming. North-Holland, Amsterdam (1966)
Stern, N.: John von Neumann's influence on electronic digital computing, 1944–1946. Ann. Hist. Comput. **2**(4), 349–362 (1980)
Stern, N.: From ENIAC to UNIVAC: An Appraisal of the Eckert-Mauchly Computers. Digital Press, Paris (1981)
Stevens, W., Myers, G., Constantine, L.: Structured design. IBM Syst. J. **13**(2), 115–139 (1974)
Strachey, C.: Contribution to panel discussion on metasyntactic and metasemantic languages (1962). In: Symbolic Languages in Data Processing: Proceedings of the Symposium Organized and Edited by the International Computation Centre, Rome, 26–31 March 1962, pp. 99–110. Gordon and Breach, New York (1962)
Strachey, C.: Towards a formal semantics (1964). In: Steel, T.B. (ed.) Formal Language Description Languages for Computer Programming, pp. 198–220. North-Holland, Amsterdam (1966)
Strachey, C.: Fundamental concepts in programming languages (1967). Unpublished lecture notes, International Summer School in Computer Programming, Copenhagen. Published as Strachey (2000)
Strachey, C.: Fundamental concepts in programming languages. High.-Order Symb. Comput. **13**, 11–49 (2000)
Strachey, C., Wilkes, M.V.: Some proposals for improving the efficiency of ALGOL 60. Commun. ACM **4**(11), 488–491 (1961)
Stroustrup, B.: The Design and Evolution of C++. Addison-Wesley, Reading (1994)
Tarski, A.: Pojęcie prawdy w językach nauk dedukcyjnych. Warsaw (1933). Translated with additions into German as Tarski (1935), and then into English as "The concept of truth in formalized languages" in Tarski, pp. 152–278 (1983)
Tarski, A.: Der Wahrheitsbegriff in den formalisierten Sprachen. Stud. Philos. **I**, 261–405 (1935)
Tarski, A.: Grundlegung der wissenschaftlichen Semantik. In: Actes du Congrès International de Philosophie Scientifique, pp. 1–8. Hermann, Paris (1936a)
Tarski, A.: O ugruntowaniu naukowej semantyki. Prz. Filoz. **39**, 50–57 (1936b). Translated into German as Tarski (1936a), and then into English as "The establishment of scientific semantics" in Tarski, pp. 401–408 (1983)
Tarski, A.: Logic, Semantics, Metamathematics, 2nd edn. Hackett Publishing, Indianapolis (1983)
Taylor, A.: The FLOW-MATIC and MATH-MATIC automatic programming systems (1960). In: Goodman, R. (ed.) Annual Review in Automatic Programming, I. Working Conference on Automatic Programming of Digital Computers held at Brighton, 1–3 April 1959, pp. 196–206. Pergamon, Elmsford (1960)
Tennent, R.D.: The denotational semantics of programming languages. Commun. ACM **19**(8), 437–543 (1976)
Todd, J.: John von Neumann and the national accounting machine. SIAM Rev. **16**(4), 526–530 (1974)
Truesdell, L.E.: The development of punch card tabulation in the bureau of the Census 1890–1940. US Department of Commerce (1965)
Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. Proc. Lond. Math. Soc. **42**, 230–265 (1936)
Turing, A.M.: Proposal for development in the mathematics department of an automatic computing engine (ACE). Technical Report, National Physical Laboratory, Teddington, UK (1946). Reprinted in Carpenter and Doran, pp. 20–105 (1986)

Turing, A.M.: Intelligent machinery. Technical Report, National Physical Laboratory (1948). Reprinted in Copeland (2004b)
Turing, A.M.: Checking a large routine (1949). In: Report of a Conference on High Speed Automatic Calculating Machines, 22–25 June 1949, pp. 70–72. University Mathematical Laboratory, Cambridge (1950)
Turing, A.M.: Computing machinery and intelligence. Mind **59**, 433–60 (1950a)
Turing, A.M.: Discussion on Dr. E. Slater's paper on "Statistics for the chess computer and the factor of mobility" (1950b). In: Symposium on Information Theory: Report of Proceedings, pp. 198–200. Ministry of Supply, London (1950)
Turing, A.M.: Discussion contribution (1951). In: Manchester University Computer, Inaugural Conference, July 1951, p. 192 (1951). Reprinted in Williams and Campbell-Kelly, pp. 165–206 (1989)
Turing, A.M.: Lecture to the London Mathematical Society on 20 February 1947. In: Carpenter, B.E., Doran, R.W. (eds.) A.M. Turing's ACE Report of 1946 and Other Papers. Charles Babbage Institute Reprint Series for the History of Computing, vol. 10, pp. 106–124. MIT Press, Cambridge (1986)
Ulam, S.M.: Von Neumann: the interaction of mathematics and computing (1980). In: Metropolis, N., Howlett, J., Rota, G.-C. (eds.) A History of Computing in the Twentieth Century, pp. 93–99. Academic Press, San Diego (1980)
van der Poel, W.L.: Some notes on the history of ALGOL. In: Zemanek, H. (ed.) A Quarter Century of IFIP, pp. 373–392. Elsevier, Amsterdam (1986)
van Heijenoort, J.: From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931. Harvard University Press, Cambridge (1967)
van Wijngaarden, A.: Generalized ALGOL (1962). In: Symbolic Languages in Data Processing: Proceedings of the Symposium Organized and Edited by the International Computation Centre, Rome, 26–31 March 1962, pp. 409–419. Gordon and Breach, New York (1962)
von Neumann, J.: First draft of a report on the EDVAC. Technical Report, Moore School of Electrical Engineering, University of Pennsylvania (1945). Reprinted as von Neumann (1993) with corrections by Michael D. Godfrey
von Neumann, J.: The general and logical theory of automata (1948). In: Jeffress, L. (ed.) Cerebral Mechanisms in Behavior, pp. 1–41. Wiley, New York (1951)
von Neumann, J.: First draft of a report on the EDVAC. IEEE Ann. Hist. Comput. **15**(4), 27–75 (1993)
Wang, H.: A variant to Turing's theory of computable numbers. J. Assoc. Comput. Mach. **4**(1), 63–92 (1957)
Wegner, P.: Operational semantics of programming languages. ACM SIGPLAN Not. **7**(1), 128–141 (1972)
Wegstein, J.H.: Algorithms: announcement. Commun. ACM **3**(2), 73 (1960)
Wells, M.B.: Reflections on the evolution of algorithmic language (1980). In: Metropolis, N., Howlett, J., Rota, G.-C. (eds.) A History of Computing in the Twentieth Century. Academic Press, San Diego (1980)
Wexelblat, R.L. (ed.): History of Programming Languages. Academic Press, San Diego (1981). From the ACM SIGPLAN History of Programming Languages Conference, 1–3 June 1978
Wheeler, D.J.: Programme organization and initial orders for the EDSAC. Proc. R. Soc. Lond. Ser. A **202**, 573–589 (1950)
Wheeler, D.J.: The use of sub-routines in programmes (1952). In: Proceedings of the 1952 ACM National Meeting (Pittsburgh), pp. 235–236. ACM, New York (1952)
Whitehead, A.N., Russell, B.: Principia Mathematica vol. I. Cambridge University Press, Cambridge (1910)
Wiener, N.: Memorandum on the mechanical solution of partial differential equations (1940). Printed in Masani et al. (1987)
Wiener, N.: Letter to Arturo Rosenblueth, 24 January 1945. Quoted in Heims, pp. 185–186 (1980)
Wiener, N.: Cybernetics. Technology Press, Wiley, New York (1948)
Wilkes, M.V.: The design of a practical high-speed computing machine. The EDSAC. Proc. R. Soc. Lond. Ser. A **195**, 274–279 (1949). Record of a discussion held on 4 March 1948

Wilkes, M.V.: Automatic calculating machines. J. R. Soc. Arts **C**(4862), 56–90 (1951a)
Wilkes, M.V.: Can machines think? Spectator **6424**, 177–178 (1951b)
Wilkes, M.V.: Pure and applied programming (1952). In: Forrester, J.W., Hamming, R.W. (eds.) Proceedings of the 1952 ACM National Meeting, Toronto, pp. 121–124 (1952)
Wilkes, M.V.: Can machines think? Proc. IRE **41**(10), 1230–1234 (1953a)
Wilkes, M.V.: The use of a 'floating address' system for orders in an automatic digital computer. Proc. Camb. Philos. Soc. **49**(1), 84–89 (1953b)
Wilkes, M.V., Wheeler, D.J., Gill, S.: The Preparation of Programs for an Electronic Digital Computer. Addison-Wesley, Reading (1951)
Wilkins, J.: An Essay Towards a Real Character and a Philosophical Language. Sa: Gellibrand and John Martyn, for the Royal Society, London (1668)
Williams, S.B.: Bell Telephone Laboratories' relay computing system (1947). In: Proceedings of a Symposium on Large-Scale Digital Calculating Machinery, 7–10 January 1947. The Annals of the Computation Laboratory of Harvard University, vol. XVI, pp. 41–68. The Navy Department Bureau of Ordnance and Harvard University, Harvard University Press, Cambridge (1947)
Williams, F.C.: The University of Manchester computing machine (1951). In: Manchester University Computer, Inaugural Conference, July 1951, pp. 171–177 (1951). Reprinted in Williams and Campbell-Kelly, pp. 165–206 (1989)
Williams, M.R., Campbell-Kelly, M. (eds.): The Early British Computer Conferences. Charles Babbage Institute Reprint Series for the History of Computing, vol. 14. MIT Press, Cambridge (1989)
Wirth, N.: Program development by stepwise refinement. Commun. ACM **14**(4), 221–227 (1971a)
Wirth, N.: The programming language Pascal. Acta Inform. **1**, 35–63 (1971b)
Wirth, N.: On the composition of well-structured programs. ACM Comput. Surv. **6**(4), 247–259 (1974)
Wirth, N.: Algorithms + Data Structures = Programs. Prentice-Hall, New York (1976)
Wirth, N., Hoare, C.A.R.: A contribution to the development of ALGOL. Commun. ACM **9**(6), 413–432 (1966)
Wisdom, J.O.: The hypothesis of cybernetics. Br. J. Philos. Sci. **II**(5), 1–24 (1951)
Woodger, M.: An introduction to ALGOL 60. Comput. J. **3**(2), 67–75 (1960)
Woodhouse, R.: On the necessary truth of certain conclusions obtained by means of imaginary quantities. Philos. Trans. R. Soc. Lond. **91**, 89–119 (1801)
Woodhouse, R.: On the independence of the analytical and geometrical methods of investigation; and on the advantages to be derived from their separation. Philos. Trans. R. Soc. Lond. **92**, 85–125 (1802)
Wulf, W.A., Russell, D.B., Haberman, A.N.: BLISS: a language for systems programming. Commun. ACM **14**(12), 780–790 (1971)
Yourdon, E.N. (ed.): Classics in Software Engineering. Yourdon Press, Englewood Cliffs (1979)
Zemanek, H.: Semiotics and programming languages. Commun. ACM **9**(3), 139–143 (1966)
Zilles, S.N.: Procedural abstraction: a linguistic protection technique. ACM SIGPLAN Not. **8**(9), 142–146 (1973)
Zuse, K.: Über den Allgemeinen Plankalkül als Mittel zur Formalierung schematisch-kombinativer Aufgaben. Arch. Math. **1**(6), 441–449 (1948)
Zuse, K.: The Computer—My Life. Springer, Berlin (1993)

Index

M. Priestley, *A Science of Operations*, History of Computing,
DOI 10.1007/978-1-84882-555-0, © Springer-Verlag London Limited 2011