ADVANCES IN

PROGRAMMING AND NON-NUMERICAL COMPUTATION

Edited by

L. FOX

Director, Oxford University Computing Laboratory

SYMPOSIUM PUBLICATIONS DIVISION

PERGAMON PRESS

OXFORD · LONDON · EDINBURGH · NEW YORK

TORONTO · SYDNEY · PARIS · BRÄUNSCHWEIG

Pergamon Press Ltd., Headington Hill Hall, Oxford
4 & 5 Fitzroy Square, London W.1

Pergamon Press (Scotland) Ltd., 2 & 3 Teviot Place, Edinburgh 1

Pergamon Press Inc., Maxwell House, Fairview Park, Elmsford,
New York 10523

Pergamon of Canada Ltd., 207 Queen's Quay West, Toronto 1

Pergamon Press (Aust.) Pty. Ltd., 19a Boundary Street, Rushcutters Bay,
N.S.W. 2011, Australia

Pergamon Press S.A.R.L., 24 rue des Écoles, Paris 5e

Vieweg & Sohn GmbH, Burgplatz 1, Braunschweig

First edition 1966
Reprinted 1969

Library of Congress Catalog Card No. 65-18420

Printed in Great Britain by A. Wheaton & Co., Exeter
08 011356 7

PREFACE

THIS is the third volume of the Proceedings of Summer Schools organised by the Oxford University Computing Laboratory and the Delegacy for Extra-Mural Studies, and published by Pergamon Press. The first two (*Numerical Solution of Ordinary and Partial Differential Equations*, edited by L. Fox, Pergamon, 1962, and *Computing Methods in Crystallography*, edited by J. S. Rollett, Pergamon, 1965) reflected the main interests of the staff of the Laboratory. The 1963 Summer School, whose 27 lectures are summarised in this volume, was of quite a different nature, since none of the Laboratory staff (including the editor) had any particular knowledge and certainly no research experience in the fields of the theory of computer programming and of non-numerical uses of digital machines. It arose through some controversial correspondence, between the editor and two of the contributors, published in the *Computer Journal* in 1962.

In that correspondence I offered a platform for Stanley Gill and Christopher Strachey, and I was interested in the possibility of collecting together, in a small volume, some of the ideas, results and ambitions of workers in the non-numerical field. This seemed to me particularly desirable as a stimulant to possible university research workers, who have virtually no undergraduate experience of these fields and who should not be expected to have to search for information in recent issues of learned journals. In this respect, at least in 1963, non-numerical analysis was poorly served in contrast to numerical analysis, in which books of all kinds, and at all levels, have proliferated in the last decade.

Gill and Strachey cooperated immediately, and both the topics and the contributors were their suggestions. I have acted mainly as secretary, catalyst, host, editor and student. The result, I hope, is a book of modest size which provides a suitable introduction for a final year student seeking interesting research possibilities not too closely connected with his undergraduate work. It should also give to the intelligent layman, who is prepared to do some non-trivial reading, ideas about just what a machine can do, how it does it, and some of the methods, and the problems, of making further advances.

It seemed reasonable to separate the book into two parts, keeping the theories of programming separate from the uses of programmes. In the first part an introduction gives a succinct historical account of the development of programming since the invention of the digital computer, and the other four chapters discuss the theory and the developing practice of methods of communicating with the computer, particularly for non-numerical purposes. The second part also has an introduction, giving a summary of possible non-numerical work, and more detail of three particular applications, in theorem-proving, game-playing and learning, and information retrieval. It is not a text-book, but I hope it might help to stimulate the writing of books, suitable in the first place for post-graduate courses, and ultimately, when the solid foundations have been laid, for undergraduates.

The contributors were

S. Gill, Professor of Computer Science, Imperial College (3 lectures),
P. M. Woodward, Royal Radar Establishment, Malvern (3 lectures),
D. W. Barron, Cambridge University Mathematical Laboratory (4 lectures),
C. Strachey, consultant (5 lectures),
J. Foster, Royal Radar Establishment, Malvern (3 lectures),
P. J. Landin, consultant (3 lectures),
D. C. Cooper, London University Computer Unit (2 lectures),
D. Michie, Experimental Programming Unit, Edinburgh University (2 lectures),
R. M. Needham, Cambridge University Mathematical Laboratory (2 lectures).

Mr. J. Iliffe (Ferranti Ltd.) gave two lectures on Evaluation Processes, which were concerned more with hardware than with software and are not included in the book.

I am very grateful to them all for their lectures, of consistently high quality, and for their written contributions which left little for the editor to worry about. But I must thank particularly Strachey and Barron, whose lectures on programming amounted to one very significant third of the programme. Strachey used a stimulating "thinking-aloud" method which is easier to write on the board, with frequent erasures, than to put in print. Barron wrote Chapter 3, and in so doing put both himself and Strachey quite brilliantly into concise print And, of course, I must thank the audience, about seventy strong drawn from the industrial and academic worlds, who contributed uninhibitedly to the discussions and to what, I believe, was a useful and successful experiment.

22 September, 1965 L. Fox

CHAPTER 1

INTRODUCTION TO AUTOMATIC COMPUTING

S. GILL

1. INTRODUCTION

This chapter lays the groundwork for the first part of this book by describing in general and elementary terms the concepts that form the basis of the normal everyday use of computers. It is not intended as a practical manual for computer users, but those who are not familiar with computers already should find this chapter a sufficient guide to enable them to follow the material of Chapters 2–5. Experienced computer users, unless they need a refresher course, can proceed to Chapter 2.

This book is about the use of *digital* computers, as opposed to the *analogue* variety. The distinction is a fundamental one, concerned with the method of representing real numbers, with which the majority of calculations are concerned. In an analogue machine we use some physical quantity which can be varied continuously, like electrical current or voltage, or mechanical displacement or rotation, and we make this proportional to the number we wish to represent. In an analogue computer based on mechanical rotation, for example, we can connect shafts through gearwheels and other contrivances so that the rotation of a shaft may depend on the rotations of others, thus causing it to represent the result of an addition, subtraction, or multiplication.

In a digital machine, on the other hand, we represent a number as a set of digits. The fundamental feature of a digit is that it can assume only one of a finite set of values (ten, in the case of a decimal digit). Thus with a given number of digits we can represent only a finite number of possibilities. We can equate these with the integers within a certain range, or by adopting a suitable scaling we can represent non-integral values to a particular degree of precision.

Both digital and analogue computers can be used for a great variety of numerical computations, with various relative advantages. But these need not concern us here. We are more concerned with the fact that digits can be used to represent other things than numbers. For example a letter from the English alphabet, that is one of 26 possibilities, can be represented by a pair

of decimal digits (which actually cater for 100 possibilities), or by 5 binary digits (each having two possible values, making a total of $2^5 = 32$ possible combinations). Using such a representation we can actually carry out alphabetical sorting in a digital computer.

Digits can also represent switch positions, and so be used to direct signals along alternative paths. This example is a significant one; it means that the operation of the machine itself can be controlled by information of the same kind as that on which it operates. This fact underlies the great advances that have been made in digital computers in recent years, and this book is very much concerned with ways of representing information by digits, especially information about what is to be done with other information.

2. THE STORED-PROGRAM COMPUTER

Digital computers have had a long history, in which the quest throughout has been for greater speed. After centuries of slow progress with mechanical and electromechanical designs, during which the speed was improved by a factor of perhaps 20 or 30, we have in the last twenty years seen a revolution wrought by the introduction of electronics, which have almost overnight increased the speed of computation by a factor of a million. This is about the same factor as that by which the output of a modern printing press (in words per hour) exceeds that of a quill pen. The full consequences of this revolution will be enormous, and are only just beginning to take effect. We shall get some glimpses of them in this book, particularly those that have to do with mathematics.

The first problem raised by electronics was how to design a machine that would really utilize the potential speed of electronics; clearly, it would be of little value merely to speed up, say, a desk calculator a million times. Nowadays, with computer manufacturing an established industry, there are great problems in exploring the many possible types of design and relating them to practical requirements. The original task of simply finding one possible design was not too difficult, and most of the ideas had been foreshadowed by builders of pre-electronic machines, notably Babbage in the nineteenth century.

The essential requirements (beyond the ability to do arithmetic at high speed on numbers represented in the form of electrical pulses) are a means of controlling the sequence of operations performed and selecting the appropriate operands for them, and some method of holding the results for later use. The latter requirement calls for some "memory" device, or *store*, which can receive and emit groups of digits at electronic speed. Clearly it would be an intolerable limitation if every result of an arithmetical operation had to be passed out of the machine in printed form, or on punched tape, which

would be a relatively very slow process. Instead we exploit the fact that in any reasonably long calculation most of the results formed are only intermediate results, to be used later in the calculation but not required to be seen. An electronic store can be used for these intermediate results, thus saving a great deal of time. Various kinds of store have been invented, holding thousands or even millions of binary digits.

Control of the machine is achieved by making it capable of performing any of a certain repertoire of elementary steps, one at a time. Each possible step can be specified by a group of symbols called an *instruction*. Thus a whole calculation can be specified by a collection of instructions, called a *program*. While the calculation is being performed a part of the computer called the *control unit* has the task of taking the instructions of the program one by one, interpreting them, and issuing the necessary control signals to the various parts of the machine.

This form of control was anticipated by Babbage, and also by designers of more recent machines which read their instructions from a punched tape or belt. This, however, was not good enough for electronic computers which need their instructions at a much greater rate, and so the alternative was adopted of making the control unit take its instructions from the store, which has a much higher speed than punched tape. Thus the same store is now used to hold both the information on which the calculation is being performed and the program that directs its execution. This step was a crucial one in the development of the subject because it paved the way for a rapid development of computer programming techniques, leading up to the kind of ideas that are described in this book. It meant that the program itself was accessible to be operated on by the machine in the same way as the data, encouraging users to experiment with various forms of automatic processing of programs. It also meant, since all the information in the store was (generally speaking) equally accessible at any moment, that the sequence in which the instructions were executed could be very flexible, perhaps depending quite frequently on the results being obtained; at a moment's notice the control unit could begin obeying a remote part of the program. Computers in which the program is held in the store are called *stored-program* computers; they include all existing electronic digital computers except the very smallest.

The rapid progress that has been made in computer programming since the first stored-program computers appeared has brought us to a stage at which we must begin to re-examine the principles on which the subject is based, and this is one of the purposes of this book. At first, attention was focussed on the arithmetical operations, and machines were judged by the range of operations provided; the store was comparatively incidental. Today our attention has shifted towards the store. We tend to take for granted the details of the arithmetic, but we devote considerable thought to the meaning of groups of

digits in the store under various circumstances, and to constructing representations of an ever widening class of concepts in terms of digits in the store.

3. PROGRAMMING

To get a mental picture of the essential features of programming, let us imagine a computer in which information is handled in the form of decimal digits (this in itself is rather atypical, since most computers use binary digits, but decimal digits are more familiar and the difference is quite unimportant). Let us suppose also that these decimal digits are handled in groups of 12; thus one group can represent, for example, an integer in the range from 0 to $10^{12} - 1$. Alternatively, by putting a decimal point at the beginning, it can represent a number between zero and unity to a precision of twelve decimal places. There are also various conventions, which we need not go into here, for representing numbers which may be either positive or negative. For our purposes it will be sufficient if we merely think of the numerical interpretation of the digits as being a non-negative integer less than 10^{12}.

Now an instruction, if it is to define an elementary operation that can be performed by this computer, must specify all the details of the operation. This means that any aspect of the operation which might be varied from one occasion to another must somehow be specified in the instruction, so that its meaning shall be unambiguous. In particular the instruction must give the identity of every operand involved in the operation.

Almost every elementary operation will involve operands taken from the store, so there must be some way of identifying these. Let us suppose that there is a label associated with every group of 12 digits that is put in the store, and that the label consists of just 3 decimal digits. This label is usually called the *address* of the 12-digit group; it must be quoted whenever a group is stored, and again when it is retrieved, so that the right group can be found. Obviously, with 3 digits we can only form 1000 distinct labels, so that no more than 1000 items can be put in the store if they are all to be retrieved again; this is therefore a limit to the size of the store.

As a simple example of an instruction let us suppose that one type of operation is that of addition, and that such an operation involves extracting two 12-digit numbers from the store, adding them together, and putting the sum into the store. An instruction for this operation must first of all specify that this is an addition, as distinct from all the other types which the machine may be capable of performing. Let us assume that a 3-digit code is used for this, and that the code for addition is 001.

The complete instruction must also specify three addresses, two to identify the operands which are to be added together, and the third to be associated with the result when it is stored away. Suppose, for example, that we want the

machine to add together the numbers labelled 200 and 201, and to store the result with the label 500. The instruction for this would be

001 200 201 500. (3.1)

Before considering other types of instruction, there are some remarks to be made about the way the store works. Suppose that, when the instruction (3.1) is to be be obeyed, there is already an item labelled 500 in the store; what happens? Does the machine succeed in obeying the instruction, and if so what will be recovered when an operand labelled 500 is next called for? Also, what happens to the operands, labelled 200 and 201, used by this instruction? Are they still in the store after the instruction has been obeyed?

These questions are not trivial, and some interesting kinds of computing procedures have been developed which are best described in terms of stores that have non-standard answers to these questions. However, all everyday stores in practice adopt the same standard rules, which are as follows. Any previous item in the store having the same label is lost when a new item is stored, so that it is only the last item having that label that can ever be recovered. Secondly, using an item that is in the store (e.g. by using it as an operand in an arithmetical operation) does not destroy it; it is merely "read", and remains in the store for further use; in fact it can be used an unlimited number of times.

So far we have made no reference to the notion of "physical position" in the store. Now it happens that the behaviour just described would be displayed equally by a store that could be described as divided permanently into 1000 sections, each capable of holding any 12 digits. The sections would be identified by addresses running from 000 to 999. This in fact corresponds more closely to the actual physical form of most stores, and is the kind of picture that is presented in most introductions to programming. Starting from this concept, the idea of "labels" to identify items is then introduced as a way of breaking away from the rigidity of this simple addressing scheme. It should be noted, however, that what matters is not whether one conceives the store as having any particular physical layout, but simply what kinds of label are permitted. If labels are limited to 3 digits each, this is just as restrictive as building a store with 1000 places each permanently associated with an address in the range 000 to 999.

Returning now to the matter of instructions, there must of course be many other types of basic operation besides addition. The code for addition is 001; let us suppose also that the code for subtraction is 002, that for multiplication is 020, and so on. For example, to cause the computer to store, with address 500, the difference of the numbers having addresses 200 and 201, we need the instruction

002 200 201 500.

Thinking of the store as divided into 1000 slots, we may describe this operation as "storing *in* 500 the difference of the numbers *in* 200 and 201". The first 3 digits of this instruction are called the *function part*, and the other three groups of 3 digits are called the *address parts* of the instruction.

Programming the computer in terms of these elementary instructions consists largely of turning a static definition of the task into a dynamic series of simple steps that will achieve the object. A favourite elementary example in programming is the evaluation of the quadratic expression

$$y = ax^2 + bx + c,$$

given the values of a, b, c and x. This example immediately illustrates that the order of the operations deserves some study, for if we simply tell the machine to evaluate each term separately and to add them together it must perform 3 multiplications and 2 additions. If however we put the expression in the form

$$y = (ax + b)x + c$$

we can eliminate one multiplication.

In practice an example like this would form part of a much larger program, in which the operands required here would already have been obtained in the store. Suppose that a, b and c have addresses 200, 201 and 202 respectively, and that x has address 300 (colloquially, "x is in 300"). We propose to put y in 500, i.e. to store the value of y associated with the address 500. The instructions required are given by

Instruction	Effect	
020 200 300 500	puts ax in 500	
001 500 201 500	puts $ax + b$ in 500	(3.2)
020 500 300 500	puts $(ax + b)x$ in 500	
001 500 202 500	puts y in 500	

Note that the values of a, b, c and x are still available for further use, if required. Note also that the intermediate results ax, $ax + b$, etc., that are formed during this calculation must be held somewhere temporarily (i.e. be stored in association with some address so that they can be identified when they are required later). In this example the address 500 has been used for all the intermediate results. One address is sufficient because each result is used once only and can be destroyed when the next is formed. Furthermore, by using the address 500 we avoid unnecessary interference with stored information; we know that any previous item with this address is destined for destruction anyway, because the address has been earmarked to receive the value of y. However, in general, such economy of addresses is not possible, and we often have to adopt other addresses to use as "working space" during a calculation.

4. INSTRUCTION CODES

As mentioned earlier, programs of instructions are themselves held in the store during execution. It will not have escaped the reader's attention that each instruction in (3.2) consists of 12 decimal digits, and is precisely the same kind of information as an operand. The store will therefore accept it as an item like any other, to be associated with any given address. Thus we can imagine the four instructions in (3.2) as being stored in association with four distinct addresses. In choosing these, however, we must know how the control unit determines the sequence in which to execute instructions. There must in fact be a normal rule of progression; we shall assume the usual and obvious one, which is that instructions are normally executed in consecutive order of their addresses, considered as integers. It is worth noting, incidentally, that this is the only way in which any ordering of the addresses is "built into" the machine; for all other purposes the addresses are merely a set of independent labels. The instructions (3.2), then, must have consecutive addresses, say from 600 to 603, and we have

Address	Store contents
600	020 200 300 500
601	001 500 201 500
602	020 500 300 500
603	001 500 202 500

Having somehow persuaded the control unit to start obeying the first of these it will automatically proceed through them all in order (and will then take the item with the address 604 as its next instruction).

In view of the variety of kinds of item that can be associated with an address, commonly including numbers and instructions and often other types of information as well, it is useful to have a single term to denote the set of digits that is stored, without implying that it has any particular meaning. The term that is used is "word". In our hypothetical machine a group of 12 decimal digits would be called a *word*. In practice various machines have different *word lengths*. Twelve decimals, or the equivalent number of binary digits, is near the average.

The code by which an instruction represents an elementary operation is called the "instruction code" of the machine. The instruction code illustrated in (3.2) is a *three address* code, so called because each instruction has room for three addresses in it. A more common kind of instruction code is the *one address* kind. With such a code it is necessary to have at least one special storage position, called an "accumulator", whose use is implicit in most of the basic operations. Addition, for example, means taking a number from the store and adding it to the number in the accumulator, leaving the

result in the accumulator. There must also be a "load" operation, which copies an item from the store into the accumulator, and there must also be a "store" operation, which puts into the store the item from the accumulator. Thus, in a one address code, the instructions (3.2) for forming

$$y = (ax + b)x + c$$

could be written in the form

Instruction	Effect	
000 200	Load a into accumulator	
020 300	Form ax in accumulator	
001 201	Form $ax + b$ in accumulator	(4.1)
020 300	Form $(ax + b)x$ in accumulator	
001 202	Form y in accumulator	
100 500	Store y with address 500	

This is clearly neater than the equivalent program (3.2) in three address code. In practice there is not a significant difference between the two types of code; this example happens to be favourable to the one address type because every operation except the first makes use of the result obtained by its predecessor. The one address instructions are of course much shorter, which means that it is usually wasteful to devote a whole word to a single instruction. They might be combined in pairs, so that (4.1) would be stored in the form

Address	Store contents
600	000 200 020 300
601	001 201 020 300
602	001 202 100 500

However, to make the examples easier to follow, we will assume that each instruction has a distinct address, and will print them in a single column. Also we will henceforth take for granted the numerical coding of the function part of each instruction, so that (4.1) would be represented in the mnemonic form

```
Load 200
Mult 300
Add  201
Mult 300
Add  202
Store 500
```

To complete the picture we must mention that computers have reading devices, for reading information off punched cards or tape, and that normally

all information, including programs as well as data, is taken into the machine in this way. In the normal course of events the program is read in first and placed in the store; then its execution begins, as a result of which any data required are also taken in, and results are printed or punched out. Operation of the input and output equipment is controlled, like the arithmetic, by instructions in the program. These instructions are clearly rather specialized, and although they call for a lot of attention from practical computer designers and programmers there is no need to dwell on them here.

5. CONTROL JUMPS AND LOOPS

Since every instruction is read into the machine through a mechanical device, the reader may wonder why we bother to put them into the store before they are used. In fact if a program were merely a single string of instructions, each to be obeyed once only, they could perfectly well be obeyed as they came into the machine. In practice, however, the situation is quite different. Any reasonably long calculation involves a great deal of repetition, with different sets of data undergoing the same operations. We therefore arrange that the program may ask the control unit to break its rule of taking its next instruction from the succeeding address, and instead to "jump" to some other specified address for its next instruction. These jumps are brought about by special instructions; for example

$$\text{Jump } 850 \tag{5.1}$$

would cause the next instructions to be taken from positions 850, 851, . . . , etc., regardless of where the jump instruction itself may be. By making the machine jump a few steps backward it can be made to repeat some operations a large number of times. Thus a comparatively short program can specify quite a long repetitive calculation. It then becomes quite sensible to hold the program in the store while it is being obeyed.

A jump instruction such as (5.1), which invariably causes a jump whenever it is encountered, is called an "unconditional" jump instruction. But these alone are not of great value. More significant are various forms of *conditional* jump instruction, or "test" instruction, which cause a jump only when some criterion is satisfied. A one address instruction code, for example, would contain a selection of test instructions with criteria related to the contents of the accumulator, depending perhaps on the sign of the number in the accumulator, or on whether it is zero.

To illustrate the use of a test instruction, suppose that there are two numbers x and y stored with the addresses 300 and 301, and that we want the computer to copy the larger number into position 400. Assuming that we can have an instruction that will test the sign of the accumulator, then we

should first make the machine form the difference $x - y$ in the accumulator. At this point, if the test instruction is used, it will lead to one of two alternative paths in the program, depending on whether the difference $x - y$ is positive or negative, i.e. on which of x and y is the greater. When the appropriate one has been copied into 400, it is a simple matter to arrange that the paths reunite for the next part of the program. This is achieved with the instructions

Address	Instruction		Effect	
600	Load	300	takes x	
601	Subtract	301	forms $x - y$	
602	Jump if pos	606	jumps if $x > y$	
603	Load	301	takes y (the greater)	(5.2)
604	Store	400	stores y	
605	Jump to	608	goes to next part of program	
606	Load	300	takes x (the greater)	
607	Store	400	stores x	

It is actually possible to shorten this bit of program by one or two instructions, but there is no need to discuss such fine points here. The essential thing to notice is the path taken by control: it always starts by obeying instructions 600, 601 and 602. After that it obeys *either* 603, 604 and 605, *or* 606 and 607, depending on circumstances. However, it will always go to 608 for the following instruction. (Also, this example assumes that the convention by which words represent numbers allows for both positive and negative numbers to be represented. Which of the various possible conventions is used does not matter here.)

The example (5.2) does not involve repetition. As a trivial and admittedly rather useless example of a repetitive task, suppose that we want to subtract y from x repeatedly until the result becomes negative, and then to store the result in 450. It will be assumed that x and y are known to be positive. The instructions required for this are given by:

Address	Instruction		Effect
700	Load	300	takes x
701	Subtract	301	subtracts y
702	Jump if pos	701	repeats if result still positive
703	Store in	450	stores result when negative

Notice that, although this "loop" of two instructions in 701 and 702 may be obeyed various numbers of times, depending on the data, it is bound to be obeyed at least once. Sometimes this is undesirable; for example, suppose that x may itself be negative, and that it is desired that no subtraction is to

occur when this is so. The program would have to be modified to the form

Address	Instruction		Effect	
700	Load	300	takes x	
701	Jump if neg	704	omits loop if x negative	
702	Subtract	301	loop as previously	(5.3)
703	Jump if pos	702		
704	Store in	450	stores result	

A small extra price has to be paid for the ability to omit the loop altogether.

Loops of many kinds occur in programs, and in a text on practical programming there would be many examples of increasing complexity. The simplest are those, like (5.3), in which the criterion for repetition is provided by the data being operated upon. Typical examples come from the field of numerical analysis, in which iterative processes are frequently employed to derive successively better approximations to a required result. Here the process is repeated until some error estimate, obtained as part of the calculation, is less than the margin of tolerance.

Many loops are required to be performed a specific number of times, the number either being fixed when the program is written or obtained in some way from the data before the loop itself is obeyed. Such loops involve *counting*, which is a simple auxiliary task that must be performed by the computer as it repeats the loop, and for which instructions must be included in the loop. For example we can make the machine keep somewhere a number i, which is the number of repetitions yet to be performed. Included in the loop are instructions that reduce i by 1 each time the loop is performed. Before the loop is entered i is set equal to the total number of repetitions required. Hence when i reaches zero the loop has been obeyed for the last time, so that an instruction that tests whether i is zero can be used to complete the loop.

Counting is such a common operation that most computers now have special registers, called index registers, designed for holding counts, and instruction codes make special provision for them.

Loops, with or without counting, abound in programs. There are nearly always loops within loops, often within other loops which may themselves be within loops. Consequently the inner loops may be obeyed a vast number of times; indeed, the computer may spend most of its time obeying a few small parts of the program, and it is these inner loops that must be given close attention if the computing time is to be minimized.

6. ADDRESS MODIFICATION

We come now to the most crucial technique of computer programming, that of causing the program itself to be modified as the calculation proceeds.

This technique does not call for any new facility in the machine; it is immediately possible by virtue of the fact that the program is held in the same store as the data, so that any operations that can be performed on the contents of the store can be performed on the program. Indeed alteration of the program can quite easily occur accidentally, through an error in the program or a fault in the machine; the result can be quite puzzling. Used properly, however, the technique adds enormously to the variety of tasks that can be programmed. It is in fact used so frequently in practice that special provisions for this purpose are often made in the design of computers.

The things that are altered are almost always the address parts of instructions, and so the significant technique is *address modification*. There are three main kinds of reason for wanting to use this technique in programming.

The first is that it may be required to select an item from the store according to some information appearing in the data. For example in a commercial calculation we may be processing a series of invoices for various types of product, each type identified by a catalogue number, n, which runs from 1 to 100. The processing involves reference to a price list which is held in the store, with addresses 200 to 299, in order of catalogue number. The raw data for each invoice will include the catalogue number describing that particular purchase, and we must program the computer so that this number is used in constructing the address part of an instruction which can then be obeyed in order to extract the appropriate price from the list. In this example the address required is $199 + n$, which is obtained by a single addition.

The second situation calling for address modification is the processing of strings of similar items in the store, when the same set of operations has to be performed in turn on each member of the string. The items may, for example, be elements of a vector, and it may be required to multiply each one by the same factor. A program of this kind could simply consist of a separate set of instructions for dealing with each element, but this would make a very long program. It is more economical of storage space to use the loop technique, making the same set of instructions operate on each element in turn. This requires, however, that wherever the address of an element is referred to in an instruction, this address is changed for each repetition of the loop so that successive elements are dealt with.

The third reason for using address modification is simply organizational. In putting together the various parts of a large program and arranging for them to work together properly, it may be inconvenient or impossible to fix all the addresses. The same piece of program may operate first on one batch of data and later on another; the same batch of data may be held first in one part of the store and then in another; a jump instruction may have to lead to various places at various stages of the computation. As

programs become more ambitious and complicated, these organizational matters assume greater importance.

There is perhaps a fourth reason for using address modification, which might be distinguished from the above three although it has connections with them. It is not of such common occurrence at present, but it is particularly relevant to the subject matter of this book. It arises when we are dealing with data having a variety of possible structures. There may for example be an indefinitely sized group of items, such as the parts of an algebraic formula, related to each other in a particular way. Some scheme must be found for representing it in the store and processing its parts in some appropriate sequence. Schemes of this kind often entail a form of coding which includes, mixed up with the data, a number of addresses identifying other parts of the data. The program must therefore frequently pick an address out of the data and use it as the address part of a subsequent instruction to process another part of the data.

We have mentioned that most computers now have special provisions for address modification. These are usually associated with the index registers, which also serve for counting, and the usual arrangement is that a special part of each instruction may indicate that the address part is not to be used as it stands but is to have the contents of one of the index registers added to it for the purposes of interpreting the instruction. For example suppose that index register no. 2 contains the number 26. An ordinary (unmodified) multiplication operation might be defined by the instruction

Instruction	Effect
Mult (no mod) 010	Multiplies number in accumulator by the number whose address is 10.

With modification this might become

Instruction	Effect
Mult (mod by 2) 010	Multiplies number in accumulator by number whose address is 36 (= 10 + 26).

The effect of this instruction, and indeed of any other instructions that use index register 2 as a modifier, can be altered by changing the contents of register 2.

Another facility sometimes provided is that of "indirect addressing" which may be illustrated by the instruction

Instruction	Effect
Mult (indirect) 010	Multiplies number in accumulator by number whose address is number whose address is 10.

Another possibility, that is actually catered for in some instruction codes, is to use the address part of the instruction to quote the actual operand instead of its address, illustrated by

Instruction	Effect
Multiply by 010	Multiplies the number in the accumulator by 10.

Many different "addressing rules" can be made up by using various combinations of these ideas, and in recent years they have proliferated considerably in actual computer designs. This proliferation reflects our increasing preoccupation with the problems of arranging information in the store, and the subject is now in a rather untidy state which needs some measure of rationalization.

7. LOGICAL OPERATIONS

In our examples so far, the actual processing of the data has been confined to simple arithmetic. Indeed our purpose has been to illustrate the structure of programs rather than the work they carry out. We shall continue with this theme throughout the rest of this chapter, but we should at this point mention briefly some other types of operation that are included in the instruction codes of computers.

Besides the arithmetical operations we have referred briefly to the input and output of information, to the jump instructions, and to the index registers which are often the subject of special instructions for counting and address modification. In addition to these, there are instructions for use when parts of a word are required to be dealt with separately. All instruction codes contain a means of picking out specified digits in a word, i.e. replacing by zero all the digits other than those required. This is best illustrated in terms of binary digits; the digits to be picked out are specified by giving a word, often called a "mask", having 1's in the required positions and 0's elsewhere. Thus, to pick out the middle 10 digits of a 30-digit word, we would use a computer operation that combined it with a mask illustrated by

$$\begin{array}{ll} \text{Data word:} & 001101000111011000100011011110 \\ \text{Mask word:} & 000000000011111111110000000000 \\ \text{Result:} & 000000000011011000100000000000 \end{array} \qquad (7.1)$$

The rule is simple: the resulting digit is a 1 if, and only if, there is a 1 in the corresponding position in both the data word *and* the mask word.

We can perhaps think of the binary digits as being the values of logical propositions: 1 represents truth, 0 represents falsity. Then the operation

(7.1) is equivalent to the logical function "and", applied to each digit position. By this analogy operations of this kind are often called "logical" operations, though the terminology is admittedly rather forced. Another "logical" operation often provided is the "or" operation, typified by

Word *A*:	00110100011101100010011011110
Word *B*:	00000000001111111110000000000
"*A or B*":	00110100011111111110011011110

and there are various others.

Shifting operations are also provided to shift the digits of a word to the right or left. In the case of numbers in the radix notation this corresponds to multiplication or division by some power of the radix, provided that special arrangements are made to deal with negative numbers, and that rounding off is performed when necessary. In view of these minor points it is common to provide "arithmetical shifts" in which these points are suitably dealt with, and also "logical shifts" which simply shift the given digits.

Arithmetical overflow, i.e. forming a number that is outside the range that can be represented by a computer word, is of great concern to the practical programmer. Various means of dealing with it are provided in computers, but these are irrelevant in the present context.

8. ERROR LOCATION AND CORRECTION

Using such techniques as those described in §§ 3–7 it has been possible to build up programs containing many thousands of instructions. In doing so, however, a number of practical difficulties have been encountered. Writing a big program takes a long time and can be very tedious work, and, moreover, it is very easy to make mistakes which are difficult to find. Programming has now been going on for the last fifteen years, in rapidly increasing volume, and in that time a great many developments have taken place.

One line of development has naturally been in techniques for locating and correcting mistakes in programs. At first sight this is a difficult problem, in view of the fact that a single small error can throw a program completely off course so that the machine may execute many instructions wrongly before this becomes apparent. One of the commonest kinds of mistake is simply to overlook the fact that information is being held at a particular address, and to allow the machine to destroy this information by storing something else with the same address. In practice, of course, a single address is often used in succession for many different purposes within one program, so that it may not be obvious when information is being overwritten incorrectly. This problem is not removed entirely by simply increasing the size of the

store. Overwriting of earlier results by later ones is an essential technique; if every result were placed in a new address, not only would enormous storage capacities be needed, but programming would be less straightforward.

However, error diagnosis is an essentially practical problem, and we are not primarily concerned with it here. Of greater interest, because they paved the way for deeper studies in the principles of programming, were developments aimed at avoiding much of the tedious clerical work which gave rise to most of the mistakes in programming.

9. SUB-ROUTINES

One of the first of these developments was the use of "sub-routines", which were simply self-contained pieces of program that could be used as components in making up programs. A sub-routine is a set of instructions

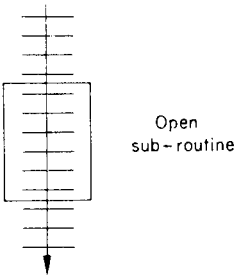


FIG. 1

telling the machine how to execute some easily-described task such as computing a well-known mathematical function, or reading or printing a series of numbers. Sub-routines are usually stored on punched tape or cards in a way that allows them to be incorporated into any program as required; sometimes common sub-routines are kept permanently inside the computer and arrangements are made to use them on demand.

A distinction is made between "open" and "closed" sub-routines. An open sub-routine is the simplest form, consisting merely of a string of instructions designed to be inserted directly into a program. During execution the control of the machine is led to the first instruction of the sub-routine, and after executing the sub-routine passes to whatever instruction may follow it, as shown in Fig. 1. The sub-routine can be thought of by the programmer as a single instruction, although it occupies, of course, several words in the store.

If the same sub-routine is used at several places within a program it is wasteful of space to include a complete copy of it at each place. Hence the

idea developed of a closed sub-routine, which is not itself inserted into the program, but is held elsewhere in the store (i.e. associated with a different group of addresses). Such a sub-routine can be used at many different places in a program by causing control to jump to it when required, and arranging that when each execution of the sub-routine is finished control returns to the appropriate place in the program. Thus only one copy of the sub-routine is needed, as shown in Fig. 2.

To do this one must arrange somehow to keep a record of the "link" address to which control must be returned after executing the sub-routine. Various ways have been invented, and the choice between them depends on

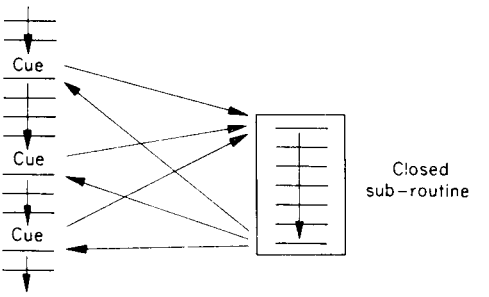


FIG. 2

the details of the particular computer. The common essential is that entry into the sub-routine must be caused by a jump instruction or pair of instructions which provide for the storage of the link in a suitable place. These instructions for entering the sub-routine are often called its "cue". Some instruction codes have special types of instruction for this purpose.

One looks upon the cue as a new kind of instruction for the machine. Once the sub-routine is in place a cue to it can be used exactly like any other machine instruction, having the effect of causing the execution of the task defined by the sub-routine.

Sub-routines do, however, raise some questions that hardly arise in the case of individual instructions, concerned with the use of storage registers. The operation carried out by a sub-routine is usually a good deal more elaborate than that of a single machine instruction, and therefore is likely to require the use of several storage registers to hold intermediate results. In describing the function of the sub-routine it is therefore necessary to say sufficient about its use of "working" registers so that the user does not allow it to destroy information which he wants to retain. A common convention is to let sub-routines use a cluster of adjacent registers (i.e. words with consecutive addresses), perhaps near the beginning of the store, and to specify in each case which of these registers are actually used.

So long as sub-routines remain distinct from one another there is no reason why they should not all use the same positions as working space. If, however, a sub-routine uses another sub-routine within itself, then the first must be so designed that the values of its working variables are not destroyed by the action of the second.

Similarly a sub-routine might well require more arguments than a single machine instruction would, and some conventions must be adopted about the placing of these. In a one address machine with an accumulator, a sub-routine which has only one argument will obviously be designed to take its argument from the accumulator, and if there is only one result to leave this in the accumulator too. If, however, there are several arguments, then it is necessary to state in the specification of the sub-routine something about their location. Again it is common to accept these arguments from a cluster of adjacent registers, often at the beginning of the store.

There is scope for many variations in the manner of giving arguments to a sub-routine; in particular one can give arguments either directly or indirectly. This means, for example, that one can specify that the first three registers of the store must be loaded with the values of three arguments required by the sub-routine, or alternatively one can specify that these three arguments may be anywhere in the store and that their respective addresses must be placed in the first three registers. Another possibility is that the three arguments may be placed in any three adjacent registers and that the address of the first of these must be quoted in the accumulator.

Thus there are many aspects of the use of sub-routines, related to the referencing of relevant items in the store, that call for attention, and a great deal of thought must be given to devising convenient and self-consistent systems. These questions have been touched upon here in the simple context of programming in terms of actual machine instructions and actual store addresses, but many of the problems are really quite fundamental and come up again and again in the more sophisticated programming schemes that have been devised. The programming language ALGOL, for example, allows the use of sub-routines, and a distinction is made between two methods of setting parameters for sub-routines, "by name" and "by value". This corresponds closely to the indirect and direct ways of giving arguments referred to in the last paragraph, although ALGOL makes no mention of store addresses and identifies all quantities by symbolic names.

The distinction made in programming between the address of a register and the item stored in it is very similar to the distinction in mathematics between the name of a variable and its value. It is true that in everyday semantics the distinction between a thing and its name is not very well defined, and there can be several different shades of meaning applied to it. In the case of programming, however, it is usually possible to give a clear

and useful definition of the distinction, and this plays a fundamental role in the development of computing systems.

10. INTERPRETIVE ROUTINES

We have described how a sub-routine can be regarded as extending the instruction code of the computer so that at any point in the program one can write either an instruction in the machine's own code or a cue for the sub-routine.

There is, however, another possibility, which is to arrange to use a program, or at least a large section of program, consisting entirely of cues for sub-routines, without including any ordinary instructions at all. In such a case it can be arranged that the control of the machine remains entirely within the group of sub-routines and never passes to the main program. This means that a private code can be used for the sub-routine cues, which are never obeyed as machine instructions but which are taken as parameters defining the operations to be executed by the sub-routines, and effectively steer the calculation in this way. A single organizing routine, called an *interpretive routine*, takes these parameters one by one and interprets them so as to call into action the appropriate sub-routines. In fact the interpretive routine carries out, explicitly, the same kind of process on the string of parameters as the machine's control unit carries out on the string of instructions in an ordinary program. Thus the machine with the interpretive routine and its sub-routines can be regarded as comprising or at least imitating a new type of computer, operating by "instructions" that are in an entirely new code that need bear no resemblance to the code of the original machine.

This idea of making one machine imitate another is a very useful and a very fundamental one. It was used by Turing in 1936 in a theoretical study of the mathematical property of computability, in which he used the principle that any machine of a certain large class could be imitated by a particular "universal" machine, and therefore that any result computable by any member of the class must be computable by the universal machine. Turing was also the first to show how this idea could be put to practical use in a computer when, in 1946, he proposed a method of programming the ACE at the National Physical Laboratory which we would now describe as an interpretive technique; he referred to the interpreted parameters as "abbreviated code instructions".

The commonest use for interpretive routines in practice is for programming calculations on operands which are not of a type catered for by the machine's own instruction repertoire. For example in a machine equipped to operate on real numbers it may be desired to carry out an extended calculation on

complex numbers, each represented as a pair of real numbers. An interpretive routine could be written with sub-routines for executing the various operations of complex arithmetic, and a suitable code could be chosen for the parameters that would enable each of these operations to be specified; in fact in this case the code might look very similar to the instruction code of the machine itself.

It is true that the parameters cannot be executed at the same rate as that at which the machine itself obeys its instructions, because the interpretive routine probably requires some 20 to 50 machine operations in order to interpret and obey each parameter. In the early days this fact deterred us from making much use of interpretive routines, but speeds of computers have so improved that it is quite practicable to use such methods. In fact the situation is now essentially reversed: a task that is worthy of the speed of a modern computer must involve such a large number of basic machine steps that we are faced with the problem of how to specify such tasks concisely. The use of an interpretive routine is one way of doing this. Without such techniques we could not make effective use of a very high speed computer.

11. COMPILERS

We have described an interpretive routine as one that tells the machine how to imitate another machine. Another way of looking at it is to say that it is doing a language conversion on the program that is supplied to it. By following the interpretive routine the computer is led to a series of basic instructions which convey the same meaning as the parameter that is being interpreted, but which are in the instruction code of the machine instead of the language in which the parameters are written. This conversion is performed continuously as the computation proceeds; each parameter is converted afresh as it is encountered.

But there is, of course, a more obvious way of doing this conversion, which is simply to take the program as originally written and to translate it into the machine's own language, once and for all, before it is executed. Programs which do this kind of conversion have come to be known as *compilers*.

Compilers take as their raw material not a string of words already held in the computer's store but a string of characters representing the written program. This gives the system designer more flexibility in choosing the format of the initial language, which can bear a very close resemblance to the conventional notations of mathematics and/or English. Moreover, since each part of the program only needs to be translated once (whereas an interpretive routine must translate a loop of instructions afresh for each

repetition) one can without extravagance use longer and more powerful translation processes. Nowadays therefore most of the work of translating programs into machine language is done by a compiler, and interpretive routines are reserved for use in special situations. Even where an interpretive routine is used its parameters will probably have been prepared by a compiler from some quite different original language.

The more powerful present-day compilers, which accept various English phrases and sentences, and also algebraic expressions of virtually unlimited length to be evaluated, are the result of several steps in the development of written programming languages. The first moves were to make the machine instructions easier to prepare, without departing from the need to conceive the program as a series of machine instructions. For example function parts of instructions were allowed to be written in a mnemonic form composed of letters, instead of the numerical code used within the machine.

Then it was permitted to give symbolic names to the address parts of instructions, thus allowing them to be written before the numerical values of many of the addresses had been determined. Thus the positions of items of data, or of various parts of the program itself, could be denoted by letters and referred to as such when writing instructions; only when the entire program had been assembled and the positions of items in the store had been fixed was it necessary to substitute actual values for all the addresses. To illustrate this, we repeat the simple example of (5.3), and show how it might appear in the symbolic form, assuming that the data positions are called x and y and the position of the result z. The program might have the appearance

Address	Instruction	Symbolic form
700	Load 300	LOAD x
701	┌─Jump if neg 704	JNEG q
702	│ ┌→Subtract 301	p) SUBT y
703	│ └─Jump if pos 702	JPOS p
704	└→Store in 450	q) STOR z

This example illustrates one convenient way of indicating to which item a symbol refers, by simply writing the symbol as a label against the item. It will be seen that the symbolic form of the program, unlike the previous form, is independent of the addresses of the data and also of the position of the program itself in the store.

The substitution of actual addresses, which must of course be done before the program can be executed, is a straightforward clerical task that can be carried out by a simple form of compiler, usually called an "assembly routine", because it can take several pieces of program written independently of one another, assemble them side by side in the store, ascertain the positions

occupied by them and those available for the data, and make the appropriate substitutions throughout the program. A small extension of assembly routines led to the automatic incorporation of standard sub-routines into the program as required, by copying them from a file of some kind accessible to the computer (e.g. on magnetic tape).

12. DEVELOPMENT OF PROGRAMMING LANGUAGES

After this there was a fairly definite break away from the kind of written language that implied a conception of the program in terms of individual machine instructions. Merely to allow the inclusion of occasional standard sub-routines in a program did not effectively depart from this conception because the idea of a sub-routine is in itself too rigid and limited in scope. A single sub-routine merely defines one type of operation, and the use of parameters only caters for variations that can be described in a simple way within the general definition of the sub-routine.

The next major step came with the introduction of algebraic expressions, which have the characteristic that there is no fixed structure into which parameters can be inserted, but a whole class of structures that can be built up by following certain rules, in much the same way as a molecule can be built up by adding further groups of atoms. For example

$$x + ay$$

is a simple formula, and this can be enlarged by replacing one of the elements by a sub-formula in the form

$$x + a(p - q),$$

and so on indefinitely, leading to such formulae as

$$z^2/p + a(p - z/[r + s]). \tag{12.1}$$

We have here an expression with a quite complicated internal structure, which conveys in a direct and economical way a great deal of information. If this had to be expressed by a series of simple instructions, or even of sub-routine cues and parameters, then a great many of the addresses or parameters would be used effectively to express the structure of the formula, and would do so in a much less obvious and economical way than (12.1).

The inclusion of algebraic expressions led to rapid developments in programming languages for scientific calculations, which frequently call for the evaluation of quantities expressed by such lengthy formulae. Impressed by the convenience of these languages, but unable to grasp or use them effectively, the business world soon felt that there should be scope for other types of programming language suitable for business purposes. Several of these were therefore developed which, in an attempt to make them more

comprehensible to the businessman, made considerable use of English words and phrases. Each instruction in these languages was usually a complete English sentence, in the imperative mood. The form of the sentence lay somewhere between the fixed format of the simple machine type of language and the very free format permitted in algebraic expressions.

The permitted form of an algebraic expression is defined recursively, i.e. at any stage in building up such an expression we can replace any element of it by a new expression, and we can do this to an indefinite extent. In building up an English sentence we do in theory enjoy this freedom, but we can only employ it to a limited extent in practice because the interpretation process is not so well defined or easy to apply as it is in the case of mathematics. In adapting English for use as a programming language it is necessary to be much more precise about permissible structures and how they are to be interpreted. The method adopted is to lay down a set of permissible "skeleton" sentences, and to allow in each case certain optional words or phrases that can be inserted if appropriate.

There are now dozens of highly elaborate programming languages in use, each accompanied by a thick programming manual, and compilers running on one or more types of computer. Several of these languages have now appeared in different versions, and large numbers of programmers are employed on writing, revising, and adapting compilers. Compiler writing, in fact, has grown from an art into an industry.

Several people have studied ways of systematizing the writing of compilers. Two broadly different approaches have been tried, sometimes in combination. The first is simply to write the compiler in a powerful programming language, which may be an already established general purpose one, or one specially devised for the purpose, reflecting the particular kinds of operation that are most frequently needed in compiling. It may perhaps be the language that the compiler is itself designed to translate, in which case a crude and inefficient machine-language version of it, prepared quickly by hand, can be used to compile a more efficient version of itself.

The second approach is to separate out, as far as possible, the distinct parts of the problem; in particular, to prepare as a separate document the definition of the language that is to be translated. An appropriate form of coding, or "metalanguage", must be used for this; various forms have been devised and shown to be adequate. Then the writing of the compiler itself becomes a matter of writing a program for which the language definition is a part of the input data. With care the compiler can be so arranged that the metalanguage is also read according to definitions supplied, so that the whole process can be built up from a very few basic language definitions that are implicit in the compiler program itself. Such compilers are called "syntax directed" compilers.

This development illustrates a general trend in programming, namely a trend towards the separation of the various decisions that a programmer has to make into distinct parts of the written program. In a machine language program a single simple decision, like deciding the location of a particular variable in the store, can have ramifications throughout the program—wherever that variable is referred to, in fact. In symbolic coding the address of the variable can be written throughout the program as a symbol, and only in one separate statement need the value of the address be given. Thus the decision is embodied in this one statement, and is kept out of the rest of the program. "Declarations" of this kind, conveying information affecting the interpretation of the accompanying program, are common features of the higher programming languages. Business programs, in particular, involve a great deal of information about the format of the data, and one language (COBOL) puts all the data declarations into a special section of the program called the "data division".

Another discernible trend is away from the strictly time-sequential operations, as defined by a machine language program, towards a more static definition of the task to be performed. This is happening in three ways. First, the ability to write formulae and to use comprehensive sub-routines means that fairly long series of computer operations can be expressed in a single statement, which has no internal time-structure. Second, some programming schemes now permit the programmer to specify two or more streams of operations which the computer can perform independently, at least for a short stretch of the calculation, so that although the correct sequencing must be observed within each stream the computer has some freedom from the single unalterable sequence of operations laid down by a conventional program. Third, some programming schemes have now been devised (and will be referred to later in this book) in which groups of definitions are presented to the computer, which finds out for itself the sequence in which to evaluate them. Indeed, the "declarations" referred to in the last paragraph are examples of definitions that are applied by the computer as and when required.

A feature that is becoming more and more common in programming is the use of recursion, i.e. permitting a definition to be expressed in terms of itself. In defining a language, for example, an algebraic expression may be defined in terms of the definition of an algebraic expression. In numerical computing, the factorial function $n!$ may be defined in terms of itself in the form

$$n! = 1 \quad \text{if } n = 0, \quad \text{otherwise}$$
$$n \times (n-1)!$$

The parallel in the case of machine language programming is the sub-routine which uses itself as a sub-routine, i.e. which contains a cue to itself. Special

provisions have to be made for such a sub-routine because it needs a whole hierarchy of links and working spaces for itself (it may have been entered several times in succession before it is finished), but if these provisions exist there is no reason why recursive sub-routines should not be used. They may be less economical of computer time than other equivalent forms of programming, but they often provide a more direct way of expressing the computation.

A programmer is nearly always working in an environment provided by other programmers. He takes a system of a particular complexity and, knowing its capabilities, he instructs it how to play the role of a system of higher complexity; and he provides a specification of this higher system so that it can, in turn, be used by someone else. Thus programmers are continually climbing on each other's shoulders. At the bottom are a few who actually write programs in the machine's own instruction code, and at the top are a great many who write working programs to be used by regular machine operators to do a job of work. In the middle are the "system programmers" who provide the compilers and the enormous number of incidental routines that are needed to make a modern computer effective.

Sometimes it seems as if a programmer is trying to climb on his own shoulders. And sometimes he seems to succeed.

CHAPTER 2

LIST PROGRAMMING

P. M. WOODWARD

1. INTRODUCTION

Techniques for manipulating numbers are as old as mathematics, and even those for mechanical computation long pre-date the invention of the electronic computer. Arithmetic work on computers thus began with a solid foundation of Numerical Analysis as a starting point. By contrast, non-numerical analysis has had to begin almost from scratch, and provides programmers with an even greater challenge. A new discipline is being born; our task is to widen it, codify it and to bring about its acceptance as a reputable art. The first step is to set up notations which will enable us to deal with non-numerical information in precise mathematical terms, for non-numerical computer problems, as conceived at present, are essentially mathematical in structure. In programming research, notation sometimes seems to become an end in itself, but no programming language can be constructed without our first having some idea of the kinds of task we want to perform.

Computing is the business of operating on data. In numerical work the items of data are numbers, individually having a large selective information content but also standing in some structural relation to one another. For example the elements of an array have not merely values but positions in the array. The quantity of information contained in the structure is usually quite small; if we consider the three components of an ordinary vector, each to six decimals, we have 10^{18} or so possibilities for the values, but only 6 possible assignments representing structural information. In non-numerical work it seems likely that structure will play a relatively larger part, and give rise to novel programming techniques. For example the *list structure*, which has not figured prominently in numerical computing, is likely to become a basic tool in non-numerical work. Indeed this has already been proved in the studies of Newell and Tonge (1960) and others in the fields of problem solving and theorem proving.

List structures of various types have been occasionally employed by programmers for many years, but no mathematically appealing formalization appeared until the work of McCarthy (1960) was first published. LISP, as

McCarthy's system is called, was a milestone in the programming art, because it used list processing as a medium for exploiting the technique of recursion, and succeeded in doing so without tiresome references to the code or language of a particular computer. By itself the published account of LISP does not constitute a programming language in the general sense, but there can be no better introduction to non-numerical techniques than a study of it. The present chapter is biased so heavily in this direction that it amounts to a short course of instruction in the use of LISP; sections 2 to 10 contain nothing else, minor changes of notation being explained by the need for consistency with other contributors to the present volume.

General programming languages ought to allow for the processing of lists. Schemes have, in fact, been implemented within the framework of FORTRAN (Gelernter *et al.* 1960, and Weizenbaum 1963) and it will not be long before programmers feel the need for extensions to ALGOL to embrace list processing. A brief idea of the methods which might then be available is given in section 11 of the present chapter, which assumes some knowledge of ALGOL on the part of the reader. The language CPL (Barron *et al.* 1963) will combine all the useful facilities of LISP with those of ALGOL and more besides. Finally, a brief account is given in section 12 of the problem of tracing through a list, since this illustrates how cumbersome the ordinary methods of looping can become when we are faced with lists of lists. Recursion is, in fact, the keynote of the subject, as a glance at section 8 may show. How widely it will be used in practical problems will depend very largely on how efficiently such programs can be compiled and run, a question which falls outside the scope of this chapter.

2. ATOMS AND LISTS

Consider a list x given by

$$x = (\mathrm{A}, \mathrm{B}, \mathrm{C}) \tag{2.1}$$

This is a list of three *atoms* which are most conveniently written in the form of capital letters. Atoms could be chess pieces, algebraic signs, railway trucks or numbers, depending on the application, but any "meaning" we may associate with them is beyond the scope of the formal mathematics. We shall not, therefore, discuss meaning any further, nor shall we attempt to split an atom into smaller units. Atoms, like numbers in arithmetic, stand only for themselves. The x in (2.1), on the other hand, is symbolic in the sense that it is being made to stand for the list (A, B, C). It can be used as a variable, and we can say that the *value* of x is (A, B, C). Variables can stand either for isolated atoms or for lists. Thus

$$y = \mathrm{A}$$

would mean that the value of y is the atom A. To avoid confusion of notation we shall use lower-case letters for variables and upper-case letters for atoms. Round brackets will be used to indicate a list, and commas to separate the items listed.

A distinction should be made between A and (A). The former is an atom, whilst (A) is a list containing as its sole item the atom A.

The items in a list need not be atoms, but may be themselves lists. For example

(A, (B, C), D)

is a list of three items, an atom, a list and an atom. It follows that brackets may be nested to any depth, and that lists such as

(((A)))

are allowed. This is a list of one item which is itself a list of one item which is a list of one atom.

3. HEADS AND TAILS

In numerical mathematics a function of a numerical variable has as its value a number. In LISP a function of a list variable has as its value an atom or list. The two simplest of these functions are the head and tail (McCarthy's "*car*" and "*cdr*"), most readily defined by the three examples

x = (A, B, C, D)

(i) $Hd[x]$ = A

$Tl[x]$ = (B, C, D)

x = A

(ii) $Hd[x]$ undefined

$Tl[x]$ undefined, because an atom cannot be split

x = (A)

(iii) $Hd[x]$ = A

$Tl[x]$ = NIL

The head of a list is its first item, the tail is what remains when the head is left out. When there is only one item in the list the tail might be thought to be a list with no items in it. Instead of allowing such a concept we define the tail of a one-item list to be NIL, and we treat NIL as an atom.

The reader may exercise himself in the use of head and tail functions by verifying the following examples.

Exercises in heads and tails

(i)	x = (A, B, C)	$Hd[Tl[Tl[x]]] = C$ $Tl[Tl[Tl[x]]] = NIL$
(ii)	x = (((A)))	$Hd[x] = ((A))$ $Hd[Hd[Hd[x]]] = A$ $Tl[x] = NIL$
(iii)	x = ((A, B), (C, D))	$Hd[x] = (A, B)$ $Tl[x] = ((C, D))$
(iv)	x = (A, (B, (C)))	$Hd[Tl[Hd[Tl[x]]]] = (C)$
(v)	x = (((A), B), C)	$Hd[Hd[Hd[x]]] = A$ $Hd[Tl[x]] = C$

It will be clear that the functions head and tail make the first item of a list more easily accessible than the last. This feature of LISP arises from the particular manner in which lists are stored in the computer. The progressive difficulty in gaining access to later items of a list exactly mirrors the amount of work which has to be done by the machine. To pursue this matter further, however, would take us into questions of machine representation, which are discussed separately in section 9.

4. CONSTRUCTIONS

The functions head and tail enable us to dissect given lists, and the complementary operation is to assemble a list from given pieces. The function "*Cons*" forms, from two arguments, the list with the properties

$$Hd[Cons[x, y]] = x$$
$$Tl[Cons[x, y]] = y$$

Thus if y is a list, $Cons[x, y]$ inserts x as a new first item in front of the other items. If y is the atom NIL, the effect is to put a pair of round brackets around x. No other forms of y are allowed; if y were an atom other than NIL, $Cons[x, y]$ would be undefined, but there are no restrictions on x.

Exercises in the use of Cons

(i)	x = A y = NIL	$Cons[x, y] = (A)$
(ii)	x = A y = (B, C)	$Cons[x, y] = (A, B, C)$

(iii) $x = (A, B)$ $\quad Cons[x, y] = ((A, B), C, D)$

$y = (C, D)$

(iv) $x = (A, B)$

$y = (C, D, E)$

$Cons[Hd[x], Cons[Hd[Tl[x]], y]] = (A, B, C, D, E)$

5. ATOM, EQ AND NULL

Three more simple functions complete the basic repertoire in LISP. These are

Atom[*x*] whose value is **true** if x is an atom, otherwise it is **false**. For convenience, **true** and **false** will be treated as atoms and written T and F.

Eq[*x*, *y*] whose value is T if x and y are the same atom, F if x and y are different atoms, and is undefined if x or y or both are not atoms.

Null[*x*] whose value is T if x is NIL, otherwise it is F. *Null* is not a primitive in LISP, as it can be expressed in terms of *Atom* and *Eq* (see section 6).

The function *Eq* is often inconvenient because of the restriction on the parameters. More useful and no harder to implement is the function

Equ[*x*, *y*] defined for any parameter x and an atom y. Its value is T if x is the atom y and F otherwise.

If this is not taken as a basic function it can readily be expressed in terms of *Atom* and *Eq* (See section 6). *Null*[*x*] is equivalent to *Equ*[*x*, NIL].

Exercises in Atom, Eq and Null

(i) $x = (NIL)$

$Atom[x] = F$

$Null[x] = F$

$Null[Hd[x]] = T$

$Null[Tl[x]] = T$

(ii) $x = ((A, B), C, D)$

$Eq[Hd[x], A]$ undefined

$Equ[Hd[x], A] = F$

6. EXPRESSIONS

Functions are particular cases of expressions, but in order to allow conditional programming, expressions must be defined more generally than this. Let us employ the symbol E to denote a general expression, and the symbol B to denote a Boolean expression. (The class E includes the class B.) The functions *Atom*, *Eq*, and *Null*, complete with their arguments, would be examples of Boolean expressions. Then we may define an expression in general as having the form

$$B_1 \rightarrow E_1, B_2 \rightarrow E_2, \ldots, B_n \rightarrow E_n, E_{n+1} \tag{6.1}$$

The value of this expression is obtained by evaluating the B's from left to right until a value T is obtained, and then taking the corresponding E. If no T is obtained the value is that of E_{n+1}. Any number of arrow clauses can be used, from nought upwards. In ALGOL the form (6.1) is equivalent to

if B_1 **then** E_1 **else**
if B_2 **then** E_2 **else**
. . . . .
if B_n **then** E_n **else** E_{n+1}

A simple illustration is provided by an expression for *Null*[x], given by

$$Atom[x] \rightarrow Eq[x, \text{NIL}], \text{F}$$

As further exercises we might define some elementary Boolean functions in terms of conditional expressions, for example

$$\begin{aligned} And[a, b] &= a \rightarrow b, \text{ F} \\ Not[a] &= a \rightarrow \text{F}, \text{T} \\ Or[a, b] &= a \rightarrow \text{T}, b \end{aligned}$$

It should be pointed out that the B's and E's in our description could themselves be conditional in form. If ambiguities of interpretation result, complete expressions can be enclosed in brackets. In LISP square brackets are used for this purpose, but where the concepts of LISP are embodied in a general programming language it is probable that some other convention will be called for.

7. RECURSION

In exercise (iv) of section 4 we gave for the concatenation of two particular lists the expression

$$Cons[Hd[x], Cons[Hd[Tl[x]], y]]$$

This forms (A, B, C, D, . . .) from

$$x = (A, B)$$
$$y = (C, D, \ldots)$$

If the list x had contained three items the expression would have become

$$Cons[Hd[x], Cons[Hd[Tl[x]], Cons[Hd[Tl[Tl[x]]], y]]]$$

Expressions like this are analogous to written-out loops in machine-code programming, where instructions are repeatedly written down as many times as required. Apart from the tedium this involves, we know that such methods can be applied only when the number of repetitions is fixed in advance. To join two lists together when we do not know how many items there may be in the first one calls for a more general approach. We achieve the required answer with the recursive function definition

$$Append[x, y] = Null[x] \rightarrow y,$$
$$Cons[Hd[x], Append[Tl[x], y]]$$

The attractive feature of a recursive definition is that it enables us to express an iterative process in a single expression. Experience shows that in certain types of problem, which abound in list-processing, recursion is both more natural and more convenient as a way of iterating than the writing of loops of commands. Any complete programming language should permit both, but in basic LISP we concentrate entirely on functional methods and hence on recursion.

8. EXAMPLES OF RECURSIVE FUNCTIONS

Append is the classical example used to illustrate recursion in LISP. Further examples are given below, and the reader will acquire some facility by verifying them, preferably after attempting them himself. No deep understanding of recursive technique can be gained without practical exercise. The solutions offered here are by no means unique, but they illustrate some standard tricks and are included with acknowledgement to J. McCarthy, J. M. Foster, D. P. Jenkins and C. Strachey.

(i) Define $Ff[x]$ to give the first atom in x, ignoring all brackets, e.g.

$$x = (((A, B), C), D, E)$$
$$Ff[x] = A$$

Solution (JMcC),

$$Ff[x] = Atom[x] \rightarrow x,$$
$$Ff[Hd[x]]$$

(ii) Define *Rev*[*x*] to reverse the order of the items in *x*, e.g.

x = (A, (B, C), D)

Rev[*x*] = (D, (B, C), A)

Solution,

$$Rev[x] = Rev1[x, \text{NIL}]$$

where $Rev1[a, b] = Null[a] \rightarrow b,$

$$Rev1[Tl[a], Cons[Hd[a], b]]$$

This example illustrates how the idea of a "working space" carries over into functional programming. The successive values of *a* and *b* in the example are

a	*b*
(A, (B, C), D)	NIL
((B, C), D)	(A)
(D)	((B, C), A)
NIL	(D, (B, C), A)

(iii) Define *Ap*[*x*, *y*] to place *y* after the final item of *x* as the new final item, e.g.

x = (A, B, C)

y = (D, E)

Ap[*x*, *y*] = (A, B, C, (D, E))

Solution (*CS*),

$$Ap[x, y] = Null[x] \rightarrow Cons[y, \text{NIL}],$$
$$Cons[Hd[x], Ap[Tl[x], y]]$$

(iv) Using *Ap*, define *Revs*[*x*] to reverse the order of the items in *x* and all its sublists, e.g.

x = (A, (B, C), D)

Revs[*x*] = (D, (C, B), A)

Solution (*JMF*),

$$Revs[x] = Atom[x] \rightarrow x,$$
$$Ap[Revs[Tl[x]], Revs[Hd[x]]]$$

(v) Define *Mem*[*a*, *x*] to have value T if the atom *a* occurs in the list *x*, otherwise value F.

Solution (*DPJ*),

$$Mem[a, x] = Atom[x] \rightarrow Eq[a, x],$$
$$Mem[a, Hd[x]] \rightarrow \mathrm{T},$$
$$Mem[a, Tl[x]]$$

(vi) Define *Flatten*[x] to remove all inner brackets from a list x, e.g.

x = (A, (B, (C, D)), E)

Flatten[x] = (A, B, C, D, E)

Solution (*JMF, DPJ*),

$$Flatten[x] = Flat[x, \mathrm{NIL}]$$

where $Flat[a, b] = Null[a] \rightarrow b,$
$$Atom[a] \rightarrow Cons[a, b],$$
$$Flat[Hd[a], Flat[Tl[a], b]]$$

(vii) Let x be a list of atoms only, including atoms O and C which may be taken to mean open and close. Define *Tree*[x] as the list corresponding to x but whose list brackets occur in place of O and C, e.g.

x = (O, O, A, C, O, B, D, O, E, C, C, F, C)

Tree[x] = ((A), (B, D, (E)), F)

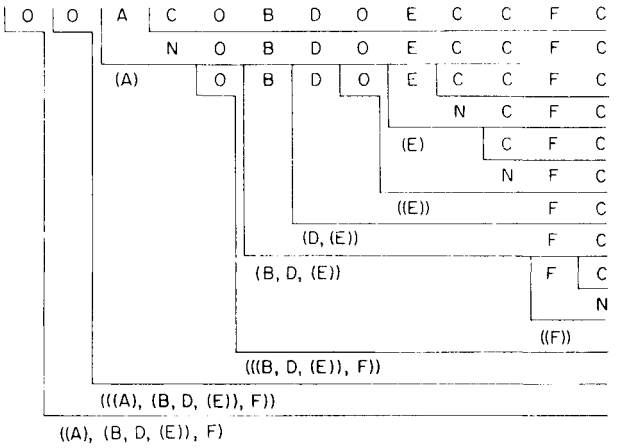
Note that O and C are assumed properly paired.

Solution (*DPJ*),

$$Tree[x] = Equ[Hd[x], \mathrm{C}] \rightarrow Cons[\mathrm{NIL}, Tl[x]],$$
$$Null[Tl[x]] \rightarrow Hd[x],$$
$$Equ[Hd[x], \mathrm{O}] \rightarrow Tree[Tree[Tl[x]]],$$
$$Cons[Cons[Hd[x], Hd[y]], Tl[y]]$$

where $y = Tree[Tl[x]]$

The action of this last program, which surpasses the one originally given in the Summer School lecture, can be followed by referring to a diagram of the following type. For reasons of space, NIL is written N. Other conventions are more easily understood by inspection than by explanation. Briefly, successive rows represent successive stages of evaluation. A vertical line represents the need to remember to take *Tree* of what follows it. The verticals on the left are longest because these *Tree* operations cannot be performed until their arguments have been evaluated, and the evaluations necessitate further *Tree* operations first. For ease of drawing, list brackets for incompletely processed rows are omitted.



9. MACHINE REPRESENTATION OF LISTS

The type of programming described in the foregoing sections calls for storage arrangements of the utmost flexibility. It would be impracticable to store list data in the sequential manner usually adopted for vectors in numerical work, because list structures are not simple strings, nor can they

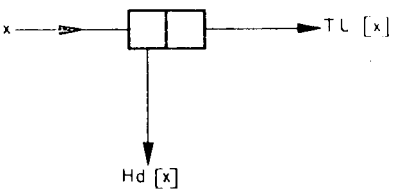


FIG. 1. The p-word representing a list x.

easily be mapped on to rectangular arrays. Furthermore they are continually growing, in ways which are not predictable until the program is running. A branching structure is called for, and where the computer word is long enough this is most easily achieved by packing a pair of addresses into one word. All the words representing the structure of a list are partitioned into halves with an address in each half. For convenience of description a pair of addresses in one word will be termed a p-word.

Every list has, corresponding to it, a p-word containing addresses which point to its head and tail (Fig. 1). When a symbol such as x is used to stand for a list, the store word allocated to x contains the address of the p-word. When x stands for an atom, its store word contains the address of the atom.

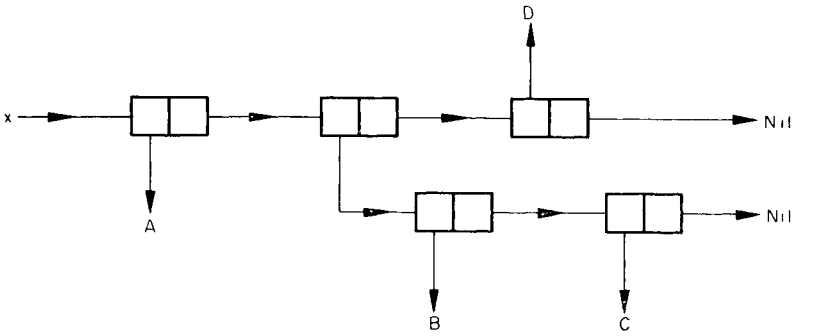


FIG. 2. System of p-words for $x = (A, (B, C), D)$.

As the storage arrangements for atoms vary from one system of implementation to another, they are not of general interest.

It follows that every list implies a chain of p-words, as many as there are items in the list, and the same goes for all the sub-lists, as shown in Fig. 2. In non-numerical problems it would be normal practice to store each atom in one place only, so that where two arrows point to the same atom the addresses giving rise to these arrows would be identical. As diagrams like Fig. 2 are somewhat clumsy they are usually abbreviated as shown in Fig. 3.

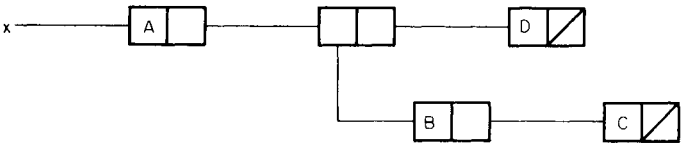


FIG. 3. Abbreviated form of Fig. 2.

The key to an understanding of LISP implementation is probably to study the action of the function *Cons*, since this is the only operation which affects the stored list data. Every time the function *Cons* is called into play one new p-word is added to the store of lists, and linked on to existing atoms or lists (Fig. 4). There is no mechanism in LISP for altering lists once they exist; they can only be extended (backwards), one p-word at a time, by the

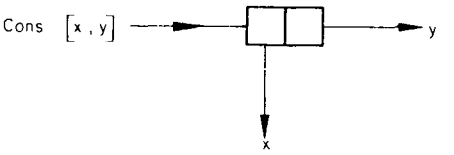


FIG. 4. *Cons*[*x*, *y*] generates one new p-word.

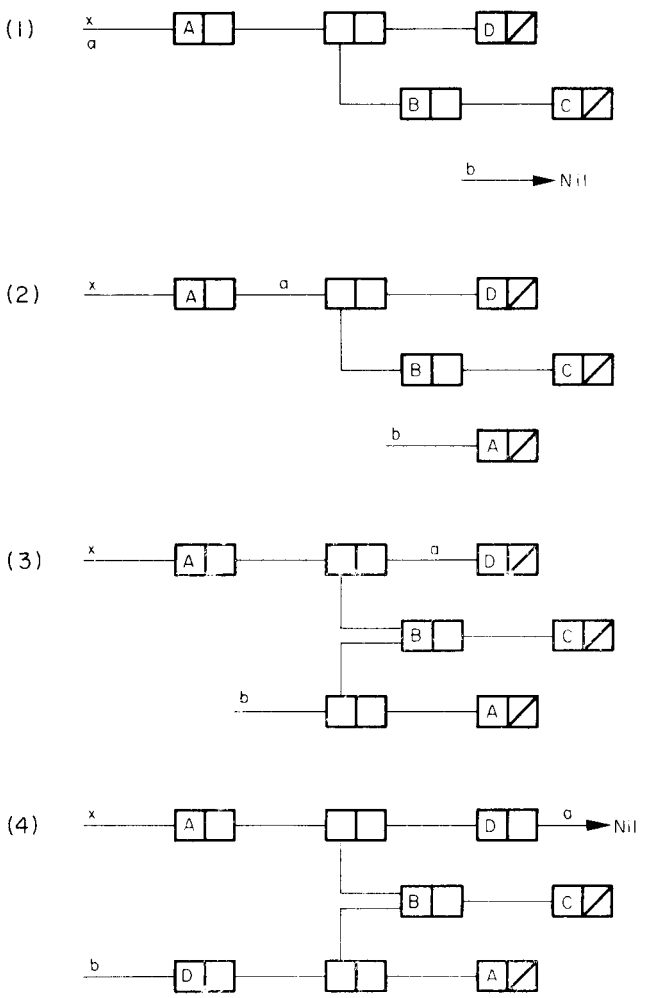


FIG. 5. Four stages in the evaluation of *Rev*[(A, (B, C), D)].

use of *Cons*. This can be followed in a simple example, such as the program for *Rev*[*x*] given in section 8 (ii). The four stages of evaluation given there for *Rev*[(A, (B, C), D)] are reproduced in terms of p-words in Fig. 5. Each recursive entry to *Rev*1[*a*, *b*]—which is given by

$$Null[a] \rightarrow b,$$

$$Rev1[Tl[a], Cons[Hd[a], b]]$$

calls *Cons* into play once, so one new p-word appears at each stage.

It will be noticed in Fig. 5 that a list, like an atom, may be pointed at from more than one place. For example the p-words of *a* are shared by *x*, and the list (B, C) is likewise shared between *x* and *b*. The lack of any means of altering the contents of an existing p-word is now seen to be less of a drawback than might at first have been supposed, since changes to one list would affect another.

10. FUNCTIONAL PROGRAMMING

The most interesting feature of LISP is the emphasis placed on functions, expressions and recursion. All these are included in ALGOL and are no longer novel, but they still remain interesting. The ALGOL programmer is not forced to stretch the ideas of functional programming to their fullest extent, and indeed is sometimes discouraged from doing so because the compiled programs may be inefficient. LISP, on the other hand, forces the use of recursion and in doing so administers a useful discipline on the programmer accustomed to thinking in terms of commands. It is often said that recursion is merely a clever way of performing repetitions which could more easily be expressed as a loop in a command language. This may often be the case, but in list processing, especially, we encounter problems which are handled more easily and naturally by recursion, as will be shown in section 12 with an example worked both ways.

Recursion apart, list processing seems to extend the range of application of functions. In numerical mathematics a function is a single number, whose value may depend on one or more other numbers. Its output is, so to speak, less general than its input. Once we permit a list as the value of a function we are equipped with the means of obtaining, in effect, several outputs at once. It is even possible to regard the entire stream of output characters from a computer as a single grand function of its initial state and inputs (see, for example, Chapter 5). In any realistic problem the function could not conveniently be defined by a single equation, but when auxiliary functions are defined in "where clauses" it becomes feasible.

It should not be supposed that we are advocating LISP as a complete programming language; a useful language for non-numerical work should include all the features of LISP and allow sequences of commands as well. ALGOL almost achieves this, but unfortunately makes no provision for lists. It is not a difficult matter to extend ALGOL so as to allow list work, and at the time of writing at least one ALGOL compiler (for the machine RREAC at the Royal Radar Establishment) is being provided with the necessary extensions. The language proposed for Cambridge and London Universities (CPL) will include list processing facilities from its inception.

11. SEQUENTIAL LIST PROGRAMMING

[*Note.* ALGOL conventions will be used in this section as a medium for discussing some of the points which arise in sequential list programming. To avoid clashing with established ALGOL notation, round brackets will henceforth be used to enclose the parameters of functions.]

The idea underlying command-type programming is that of assigning values to variables so that these values are stored up for subsequent recall and use. The ALGOL assignment statement

$$s := x$$

means "copy the value of x into s" and enables us to go and alter one of them, at the same time keeping the original value. This command could equally apply to variables of list type, though it would have to be explained more carefully. The assignment would not cause a separate copy to be made of the atom or list x, but would merely cause s to refer to the same atom or list as x was currently referring to. Having assigned x to s in this sense we might wish to write a further command, such as

$$x := Tl(x)$$

and there would now be two simultaneous pointers, s and x, in use at different places on the same list. The ability to keep a finger at the place—or indeed as many fingers as desired—does much to provide an often false sense of security.

In LISP no existing lists can ever be altered, though they can grow additional heads. There is no intrinsic reason why, in a general list-processing language, one should be barred from making alterations. It is sometimes the easiest way to bring about a desired result. Consider, for example, the function *Append* given in section 7. The old definition, in ALGOL notation with obvious extensions, would be written

```
list procedure Append(x, y); list x, y;                    (11.1)
Append:= if Null(x) then y else
         Cons(Hd(x), Append(Tl(x), y))
```

This program operates by taking the existing list y and attaching new p-words on to its front, one by one, until the list x has been reconstructed. Although the sublists of x are used in their original form, each item of x gives rise to a new p-word in store, as shown in Fig. 6, where the lists

((A, B), C, (D))

(E, F)

are appended to give

((A, B), C, (D), E, F)

If for any reason we wish to preserve the original list x the process is satisfactory, but if we no longer require x it would be simpler to replace its final NIL by a pointer to y. In LISP this is not possible. In an extended version of ALGOL, however, we might write

$$d: \ \textbf{if } \mathit{Null}(\mathit{Tl}(x)) \ \textbf{then } \mathit{Set\ Tail}(x, y) \qquad (11.2)$$
$$\textbf{else begin } x := \mathit{Tl}(x); \ \textbf{go to } d \ \textbf{end}$$

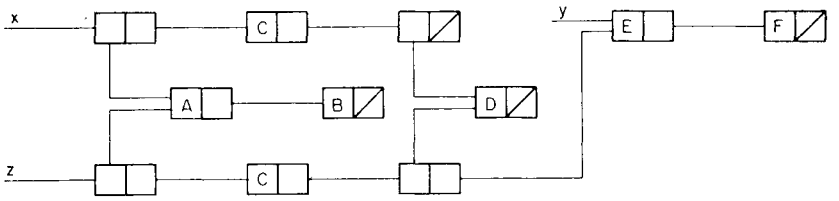


FIG. 6. Result of $z = \mathit{Append}(x, y)$.

This assumes the existence of a procedure *Set Tail*. Two such basic procedures are proposed,

$\mathit{Set\ Head}(x, y)$ Alters $\mathit{Hd}(x)$ to become y,

$\mathit{Set\ Tail}(x, y)$ Alters $\mathit{Tl}(x)$ to become y.

"To become y" means to point to the same atom or list as y is pointing to. The effect is as if we could write $\mathit{Hd}(x) := y$ and $\mathit{Tl}(x) := y$, but assignments of this form might be thought to extend ALGOL too severely, since they call for evaluations on the left hand side.

The ALGOL statement (11.2) (labelled d) merely attaches y to the end of x, but in doing so it leaves us with x pointing to the last item in the original list x, and with no means of moving it back again. We ought, therefore, to use a purely temporary pointer for such a purpose. This could be done by starting with the assignment

$$p := x$$

and then writing the statement d with p instead of x. This is precisely what ought to happen if we defined a procedure $\mathit{Join}(x, y)$ by preceding the statement d with the heading

$$\textbf{procedure } \mathit{Join}\ (x, y); \ \ \textbf{reference } x; \ \ \textbf{list } x, y;$$

The crucial point is that we should say **reference** x, since this would effectively assign x to a temporary variable which would then replace x in the procedure body (here the statement d). This assignment would not cause an entirely new copy of the list x to be made: it would merely use the temporary variable as an additional pointer to the same list. Only the address of the list would be copied.

In numerical work it is common to stipulate that an actual parameter of a function or procedure should be evaluated and given to the program defining the procedure in the form of a number. This stipulation is made in the definition of the procedure by including "**value** x" (say) in the declaration heading. Once the number has been obtained and handed over, all connexion between the procedure definition and the variable or expression which formed the actual parameter is severed. In such a "call by value" the programmer may rest assured that the action of the procedure will not upset the variable he may have written for the actual parameter. In list work a "call by value" could be arranged to afford just the same protection, but it would entail the making of a complete copy of the list given by the actual parameter. The "call by reference" described in the previous paragraph is less elaborate, but it affords no protection against mutilation of actual parameters. (Indeed our program defining $Join(x, y)$ has the mutilation of x as its sole purpose.) It protects the pointer without protecting the list.

Protection is by no means the sole object of parameter substitution by value or by reference, since the efficiency of compiled programs may depend markedly on the type of substitution used. It is not part of the present purpose to enter into a discussion of these more advanced questions, except to remark that the inclusion of **reference** x, y in the heading of the declaration for *Append* in (11.1) would have conformed to good practice.

12. THE TRACE PROBLEM

As already pointed out, recursion is a technique which is well suited to list processing. Nothing illustrates this fact better than the apparently simple problem of making a copy of a complete list-structure, in the genuine sense of constructing a second chain of p-words for a list and all its sublists. One short recursive definition suffices, given by

$$Copy[x] = Atom[x] \rightarrow x,$$
$$Cons[Copy[Hd[x]], Copy[Tl[x]]]$$

or, in ALGOL notation,

list procedure $Copy(x)$; **reference** x; **list** x; (12.1)
$Copy :=$ **if** $Atom(x)$ **then** x **else**
$Cons(Copy(Hd(x)), Copy(Tl(x)))$

It is instructive to attempt the same thing without resorting to the use of recursion, for it gives a good appreciation of the mechanics of the task.

The essence of the problem is to trace through all the items of a list and its sublists. Every p-word in the structure must be examined before it can be copied, which is simple enough if there are no sublists, as the following

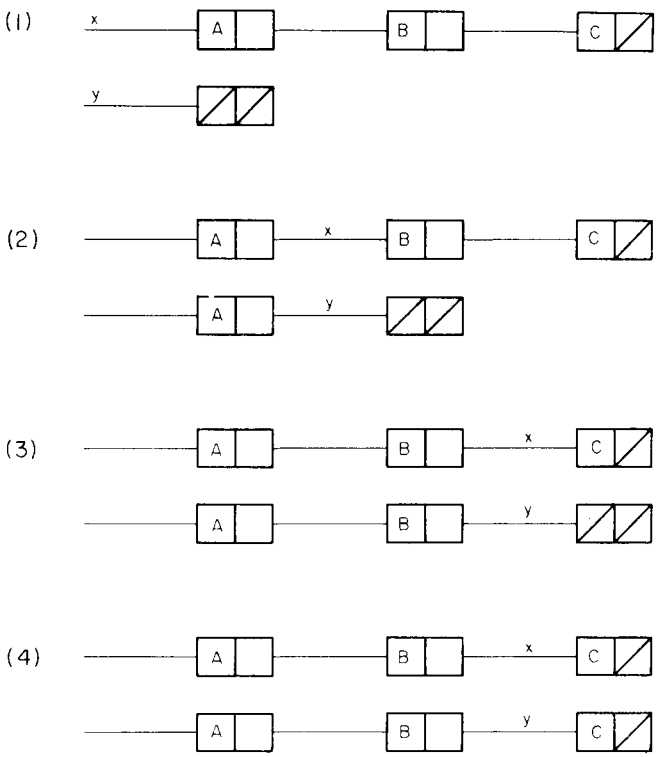


FIG. 7. Stages in copying a list of atoms.

method shows. It is assumed that x is a list of atoms, and "*Word*" is an abbreviation for *Cons*(NIL, NIL).

```
        y := Word;
s:      Set Head (y, Hd(x));
        if Null (Tl(x)) then go to finish else
        begin   Set Tail (y, Word);
                x := Tl(x);
                y := Tl(y);
                go to s
        end
```

The state of play each time the label s is passed is shown in Fig. 7 for x = (A, B, C). It will be seen that x and y are used as markers and progressively stepped through the lists, x inspecting and y copying. But this is the easy problem. When there are sublists the marker will encounter forks, from

which both branches must be traversed in turn. As the pointers linking p-words are unidirectional, completion of one branch leaves the marker with no means of return to the fork. Thus a special list of forks must be constructed for rescue purposes. The process may be followed with the help of a

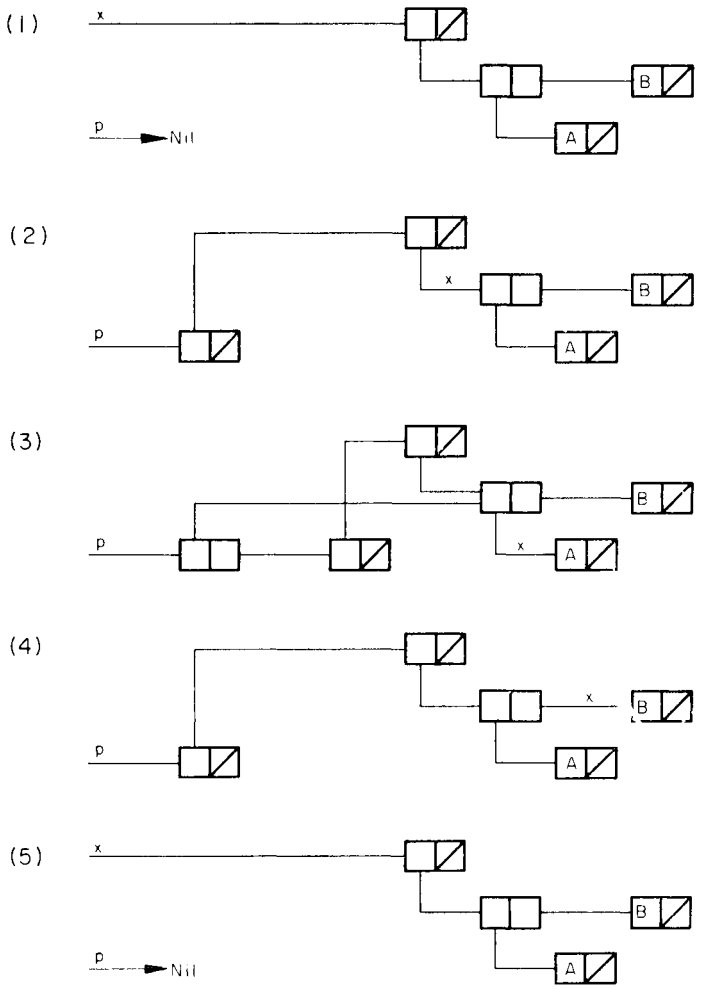


FIG. 8. Five stages in tracing (((A), B)).

skeleton trace program which moves a marker to every p-word of a list-structure. The following program does this and nothing else. Its action may be tried out on the example shown in Fig. 8, where the state of the lists and variables is shown each time the label "*start*" is passed.

The list being traced is x, the rescue list is p.

```
        p := NIL;
start:  if not Atom (Hd(x)) then begin p := Cons (x, p);
                                       x := Hd(x);
                                       go to start
                                  end;
r:      if Tl(x) ≠ NIL then begin x := Tl(x);
                                   go to start
                             end
                        else if p ≠ NIL then begin x := Hd(p);
                                                   p := Tl(p);
                                                   go to r
                                              end
```

When it is realized that this program does no more than trace one list it would seem absurd to prefer such a method to a recursion, and pointless to give the complete program for copying in these terms instead of (12.1).

The difficulty of tracing is due partly to the conventions about machine representation, and the question of alternative representation is much discussed. The simplest modification which has been proposed is that due to Perlis and Thornton (1960), who adopt a different convention for terminating a list. In addition to a special marker, equivalent to McCarthy's NIL, they store at the end of each sublist a pointer back to the originating fork. The resulting structure is known as a Threaded List, because a line can be drawn along the arrows so as to pass through every p-word without taking the pen off the paper. This scheme has as its main drawback the inability to share sublists. A FORTRAN scheme, described by Weizenbaum (1963), uses forward and backward pointers between every item, but the resulting structure occupies more space. McCarthy's scheme has the attraction of simplicity, and provided recursion can be implemented efficiently it is not difficult to use.

13. GARBAGE

In LISP, and to a lesser degree in schemes which allow list mutilation, new p-words have constantly to be found for constructing new lists. In a practical implementation an area of the computer store must be set aside for p-words, and a means has to be found for distinguishing those which are in use from those which are free. This is a dynamic problem, for the list area may quickly become filled. Work would then come to a standstill unless

discarded p-words could be reclaimed. When no free word can be found a round of "garbage collection" must be set in train before work can continue. Garbage may be defined as all p-words which cannot be reached by following the chains of pointers from active list variables.

14. ACKNOWLEDGEMENTS

The writer thanks his colleagues Dr. D. P. Jenkins and Dr. J. M. Foster for the part they have played in convincing him that recursion is easy, for answering innumerable questions and for supplying some elegant examples, especially *Flat* and *Tree*. The principal debt is to J. McCarthy.

15. REFERENCES

BARRON, D. W., BUXTON, J. N., HARTLEY, D. F., NIXON, E. and STRACHEY, C. (1963) The main features of CPL, *Computer Journal* **6**, 134–143.

GELERNTER, H., HANSEN, J. R. and GERBERICH, C. L. (1960) A Fortran-Compiled List-processing Language, *Jour. A.C.M.* **7**, 87–101.

MCCARTHY, J. (1960) Recursive Functions of Symbolic Expressions and their Computation by Machine, *Comm. A.C.M.* **3**, 184–195.

NEWELL, A. and TONGE, F. M. (1960) An Introduction to Information Processing Language V, *Comm. A.C.M.* **3**, 205–211. Gives references to work on problem-solving etc.

PERLIS, A. J. and THORNTON, C. (1960) Symbol Manipulation by Threaded Lists, *Comm. A.C.M.* **3**, 195–204.

WEIZENBAUM, J. (1963) Symmetric List Processor, *Comm. A.C.M.* **6**, 524–544.

CHAPTER 3

PROGRAMMING

D. W. BARRON AND C. STRACHEY

1. INTRODUCTION

This chapter is not a programming course in the strict sense, but it is intended to give some idea of the methods and techniques used in programming non-numerical problems. Since the purpose is to show how one sets about programming such problems rather than teaching the use of a particular computer, it is possible to be somewhat less formal than would otherwise be the case. The first part of this chapter covers the necessary groundwork, and in the second part we provide a number of worked examples of programs for contrived, though non-trivial, problems. Finally we consider in outline how a computer can be organized to deal with functional programs. Attention is directed almost exclusively to the most convenient way of expressing what is to be done, without regard to machine "efficiency". Thus, whether a program requires 20,000 words of store when it could be done using 10,000 words, or whether it takes 20 minutes when it might only take 1, is regarded as irrelevant. (One should guard, however, against taking such arguments to extremes—there is a very real difference between a process which requires a finite, though large, number of steps, and one which requires an infinite number.)

The first question to be decided is the programming language to be used. Basic machine code can be dismissed immediately: it would be over-stating the case to describe machine code as convenient for anything, but since the designer probably visualized his machine as an arithmetic device rather than a symbol-manipulating device it is likely that the machine code is less inconvenient for numerical work than for non-numerical work. Therefore we use a "high-level" language, and program the machine to translate this language into machine code. LISP, described in Chapter 2, is an example of a "problem-oriented" language which is designed for list-processing.

What are the requirements for a language for non-numerical programming? List processing capabilities are obviously important, but they are not sufficient alone. However much one may look down on people who manipulate numbers, in almost all non-numerical programs it is necessary from time to

time to be able to count. (It is interesting to see the lengths to which some list-processing systems are prepared to go in order to avoid the need for counting.) There are some non-numerical problems which are more conveniently treated by conventional techniques than by list processing. Similarly, although functional programming is a powerful technique, there are situations where an "old-fashioned" command-structure language is more convenient. It is therefore desirable to be able to use command or functional forms at will. A further argument against an entirely functional language is that all operators must take the form of functions, and must therefore be prefixed. Thus instead of writing

$$a \times b + c \times d$$

it is necessary to write

$$Plus\,[Times\,[a, b], Times\,[c, d]]$$

Obviously the language should allow the use of the common infixed operators.

A language which meets most of these requirements is CPL. In this chapter we introduce a language which is essentially the relevant parts of CPL with some simplifications. It is therefore by no means a complete or accurate account of CPL, but it introduces all the concepts which are necessary to non-numerical programming.

2. A SIMPLIFIED DESCRIPTION OF CPL

Many of the facilities will be introduced by example rather than explicit statement. Programs will be written using the normal conventions of typography, rather than in the sometimes inelegant form dictated by the restricted character-set of the Flexowriter which produces programs in punched-tape form.

2.1. *Items, Names and Types*

A program operates on *items* which are characterized by a *name* and a *type*. The name identifies the item, and is a property of the item, not a position in the store. (This is in contrast with machine-code programs, where the name pertains to a particular storage cell, not an item of information.) Names can consist of one or more letters; if the name has more than one letter the first letter must be a capital. Examples of names are

$$a \quad b \quad Cons \quad XYZ \quad List \quad LIST$$

Names can also have primes attached, e.g.

$$a' \quad b'''$$

and names which start with a capital can include numbers, e.g.

$$List1 \quad List2$$

The type of a data item tells us what sort of object it is. It may be numerical or non-numerical: some common types are

real	an ordinary number
integer	an integer
Boolean	a truth value
logical	a group of binary digits.

Another sort of item that can occur in a program is a **function**. (Note the use of bold type for what are called basic symbols—that is words which are part of the programming language, not names for data items.)

2.2. *Expressions, Commands and Definitions*

Data items and functions can be formed into *expressions*, which can be numerical or non-numerical depending on the component items and operators. For example, if a, b, c, d are **real** variables and p, q, r are **Boolean** variables,

$(a + b)/(c + d)$ is a numerical expression

$p \wedge (q \vee r)$ is a Boolean expression.

A program is made up of *commands* and *definitions*. The commands are instructions to the computing system to carry out certain operations, and the definitions provide an environment in which the commands are interpreted; both are made up of expressions. Definitions may specify the names of variables being used, with an initial value or type, or may define functions. They are introduced by the word **let**. Examples:

let $a = 3$

let x **be real**

let $Second[z] = Hd[Tl[z]]$

(z is a dummy variable: when the function is used any variable or expression can be written as an argument.)

Examples of simple commands (*assignment commands*) are

$$x := x + 1$$

$$z := Tl[z]$$

$$Tl[y] := \dot{x}$$

Note the use of the symbol := (read as "becomes"), also the use of a function on the left-hand side of an assignment.

2.3. *A Simple Program*

Consider the following example of a simple numerical program. This is a program to calculate e^x using the well-known result

$$e^x = 1 + x + x^2/2! + x^3/3! + \cdots$$

```
§ let t, s, n = 1, 0, 1
  let x be real
  Read [x]
    t, s, n := tx/n, s + t, n + 1
    repeat until t ≪ 1
  Write [s] §
```

This program has many interesting features: we first give a line-by-line commentary.

Line 1: defines three variables t, s, n and assigns initial values 1, 0, 1 respectively.

Line 2: defines a **real** variable x, without initial value.

Line 3: a number is read from the input device and assigned as the value of x.

Line 4:

$$\left.\begin{array}{l} t := tx/n \\ s := s + t \\ n := n + 1 \end{array}\right|$$

These assignments take place *simultaneously*. The right-hand sides are worked out using the "old" values of t, s, n, then the "new" values are assigned.

Line 5: the **repeat** qualifies the preceding command. The symbol $\ll$ is a special relation meaning "an order of magnitude less than".

Line 6: the current value of s is sent to the output device.

The multiple assignment is a particularly powerful construction: even in a command-structure program it removes a great deal of the artificial sequencing that has to be imposed in less sophisticated programming languages. To verify the working of the program, consider the values of t, s and n as the assignment of Line 4 is repeated:

t	s	n
1	0	1
x	1	2
$x^2/2$	$1 + x$	3
$x^3/3.2$	$1 + x + x^2/2$	4

The symbols § and $ are called *section brackets* and the combination

§ definitions
commands $

is called a block: the importance of a block will become apparent later.

2.4. *Repetition Mechanisms*

In the example of the previous section a **repeat** clause was used. Repetition is a very common device in programming, and there are several mechanisms in CPL for this purpose.

Let C be a command, and b a condition which is **true** or **false**. Then the possible forms are:

(a) **while** b **do** C

(b) **until** b **do** C

(c) C **repeat while** b

(d) C **repeat until** b

e.g. **while** $x > 0$ **do** $x := x - a$

until $Null[Tl[z]]$ **do** $z := Tl[z]$

$x, y := x + 1, y - 1$ **repeat until** $(x = 10) \vee (y = 0)$

The difference between forms (a) and (c) is that for (a), if the condition is **false** the command is not executed, whereas for (c), if the condition is **false** the command is nevertheless obeyed once. The condition b can be a simple relation or a compound relation, or a predicate function. A compound relation is built up out of simple relations and the operators

$\wedge$ (and)

$\vee$ (or)

$\sim$ (not)

$a \wedge b$ is **true** only if both a and b are **true**

$a \vee b$ is **true** if either or both of a and b is **true**

$\sim a$ is **true** if a is **false**

A predicate function is one which has the value **true** or **false** according as the argument does or does not satisfy some condition, as for example,

$Null[x]$ is **true** if $x =$ NIL, and **false** otherwise

$Atom[x]$ is **true** if x is an atom, and **false** otherwise

An alternative repetition mechanism, less useful for non-numerical work, is the **for** command, which causes a command (the "controlled command")

to be executed repeatedly with a particular variable (the "controlled variable") taking a different value at each repetition, e.g.

for $i = 1$ **to** 10 **do** C

for $x = 0, 0{\cdot}2, \ldots, 1{\cdot}6$ **do** C

The first of these would cause the controlled command to be executed ten times, i taking the values 1, 2, 3 . . . 10. The second would cause the controlled command to be repeated with x taking the values 0, 0·2, 0·4, . . . , 1·6.

2.5. *Compound Commands*

It is often required to repeat more than one command. A sequence of commands can always be grouped by *section brackets* § and $ to form a *compound command* which is treated as a single command for purposes of repetition. For example,

until *Null*[*Tl*[*z*]] **do** § *Append* [*Hd*[*z*], *y*]; $z := Tl[z]$ $

Section brackets may have identifying letters or numbers attached, e.g. §1.2, §A1.

2.6. *Conditional Elements*

The general form of a *conditional expression* is:

$$(b_1 \rightarrow E_1, b_2 \rightarrow E_2, \ldots, b_n \rightarrow E_n, E_{n+1}).$$

The b's are conditions, and the E's are expressions (which may themselves be conditional). The value of the expression is the E paired with the first b which is found to have the value **true**, starting with b_1. If none of the b's has the value **true** the value of the expression is E_{n+1}. For example,

$$b := a < 0 \rightarrow -1, a = 0 \rightarrow 0, 1$$

$$x := Null[Tl[z]] \rightarrow Hd[y], Hd[Tl[y]]$$

An alternative form of the conditional element is the *conditional command.* If b is a condition and C, $C1$, $C2$ are commands (or compound commands) then the forms are

if b **then do** C

test b **then do** $C1$ **or do** $C2$

The first of these provides an "all-or-nothing" choice: if the condition b is **false** the command C is omitted. The second gives an "either-or" choice:

if *b* is **true** the command *C*1 is obeyed and if *b* is **false** the command *C*2 is obeyed. Either of these may be conditional, for example

test *b*1 **then do** *C*1
or test *b*2 **then do** *C*2
or test *b*3 **then do** *C*3
or do *C*4

This is very reminiscent of a conditional expression, since the effect is to test *b*1, *b*2 . . . in turn till one is found which is **true**, obeying the corresponding command; if all are **false** the final command is obeyed. Observe the difference between this sequence and the sequence

if *b*1 **then do** *C*1
if *b*2 **then do** *C*2
if *b*3 **then do** *C*3
. . .

2.7. *Arrays*

An array is an assemblage of data items in a fixed structure, a particular item being identified by one or more suffices (written after the array name in square brackets). Thus *A*[*i*] denotes an element of a one-dimensional array (or vector), *B*[*i*, *j*] an element of a two-dimensional array, and so on. Like any other data item in a program, an array can be given any name.

The definition of an array includes a specification of the number and range of the suffices, e.g.

let A = *Array* [(1, 10)]
let B = *Array* [(−5, 5), (−10, 10)]

2.8. *Labels and Jumps*

Normally, commands are obeyed in the sequence they appear in the program. Any command can be *labelled*, and the normal sequence can be broken by a *jump* to a labelled command, e.g.

*L*1: *x*, *y* := 0, 1
. . .
. . .
go to *L*1

Any name can be used as a label, provided it is not being used for some other purpose.

A conditional jump can be achieved using a conditional command, e.g.

$$\textbf{if } x > 0 \textbf{ go to END}$$

or by using a conditional expression, e.g.

$$\textbf{go to } x < 0 \rightarrow L1,\ x = 0 \rightarrow L2,\ L3$$

2.9. *Functions*

Functions are one of the most important parts of CPL for non-numerical work. A function is a rule for obtaining a value: it has a name, and is defined by an expression involving its arguments, and possibly other functions, e.g.

$$\textbf{let } Second[x] = Hd[Tl[x]]$$

It is essentially a complicated form of expression, and a *function call* i.e. the function name with actual parameter(s) to be used as argument(s), can appear as an expression or part of an expression, e.g.

$$y := Second[z]$$

In strict CPL the type of the arguments must be given in the definition, but we shall dispense with this requirement.

Many of the functions we shall use will be defined *recursively*: again, in strict CPL this must be specified, but we shall assume that any function can be defined recursively without saying so. A simple example of a recursive function is the factorial function

$$\textbf{let } Fact[z] = (z = 0) \rightarrow 1,\ zFact[z - 1]$$

As a simple example of a list-processing function, take a function which tests whether two lists are identical:

$$\begin{aligned}\textbf{let } Equal[x, y] = {} & Atom[x] \rightarrow Atom[y] \wedge Eq[x, y],\\ & Atom[y] \rightarrow \textbf{false},\\ & Equal[Hd[x], Hd[y]] \wedge Equal[Tl[x], Tl[y]]\end{aligned}$$

(Notice the way in which we make use of the fact that the predicates in a conditional expression are examined sequentially. $Atom[y]$ is tested only if it is already known that x is not an atom.)

Although the definition of a function in terms of a conditional expression is very powerful, it is sometimes required to define a function as the result of obeying certain commands. The way of doing this is best illustrated by an example such as

$$\begin{aligned}&\textbf{let } Factorial[z] = \textbf{result of}\\ &\quad \S\ \textbf{let } f, z' = 1, z\\ &\quad\ \textbf{until } z' = 0 \textbf{ do } f, z' := z'f, z' - 1\\ &\quad\ \textbf{result} := f\ \ \S\end{aligned}$$

The operations to be carried out are grouped in section brackets, and include an assignment to a special "variable" **result**. Here f and z' are *local variables* which exist only whilst the function is being evaluated; they are independent of any variables of the same name which may be used elsewhere in the program, and serve as "working variables". Notice that it is necessary to use a working variable z' which is initially set equal to z; a function can only produce a result, and cannot itself change its argument(s).

As another example of the definition of a function in terms of commands, consider a function to reverse a list given by

let *Rev*[x] = **result of**
  § **let** x', P = x, NIL
  until *Null*[x'] **do**
    x', P := *Tl*[x'], *Cons*[*Hd*[x'], P]
  result := P §

2.10. *Routines*

A function is essentially a complicated expression: a *routine* is a complicated command. It has a name, some *formal parameters*, and a *body*, which consists of a sequence of commands manipulating the formal parameters. When it is *called* in a program, the name is written with some *actual parameters*: this causes the commands of the routine to be obeyed with the actual parameters substituted for the formal parameters. To illustrate the difference between functions and routines, consider the operation of adding an item to the end of a list; for example, given the lists

x = (A, B, C) and
y = (D, E)

to form the list (A, B, C, (D, E)). This can be done by the function (see section 7 of Chapter 2) defined by

let *Ap*1[x, y] = *Null*[x] → *Cons*[y, NIL],
  Cons[*Hd*[x], *Ap*1[*Tl*[x], y]]

The characteristic feature of a function is that it produces a result without changing its arguments, so that *Ap*1[x, y] produces a new list, leaving x and y unaltered. However, if we do not mind altering the original list x, the operation of adding a new element is simply done by a routine, which goes down the list x until it finds the end, then adds y, as for example

define routine *Ap*2[x, y]
  § **let** x' = x
  until *Null*[*Tl*[x']] **do** x' := *Tl*[x']
    Tl[x'] := y §

Note that if y is changed after this routine has been used, x will also be changed. In general, routines are not widely used for list processing, since in a complex structure with common sub-lists changing the elements of a list may produce unexpected effects. By contrast, copying a list and changing the copy (which is what a list-processing function does) is always safe.

In the last example the formal parameters do not appear explicitly on the left-hand side of assignment commands. It is therefore sufficient, when the routine is called, to substitute the actual parameters by *value*. Thus if the call

$$Ap2[p, q]$$

appeared in a program the values of p and q would be handed over to the routine $Ap2$. If formal parameters appear explicitly on the left-hand side of assignments then it is inappropriate to hand over the value of the corresponding actual parameter: instead it is necessary to hand over information which can be used by the routine in changing the value of the actual parameter. A parameter called in this way is said to be called *by reference*, and such parameters are indicated by the call **ref**, e.g.

define routine $X[a, b, c]$
ref a, b
. . .
. . .

The importance of the distinction between call by reference and call by value will be brought out in subsequent examples.

2.11. *Blocks and Definitions*

The concept of a block, consisting of some definitions followed by some commands, the whole grouped in section brackets, has already been introduced. The body of a **result of** clause or a routine definition is another example of a block. The importance of a block arises from the fact that blocks can be nested, and that the definitions at the head of a block only apply within that block and any it encloses. Moreover, each time a block is entered and its definitions obeyed, the defined objects are created anew, so that, for example, if a function or routine is used recursively, at each incarnation a new set of local variables is created.

However, definitions need not always appear at the head of a block. It is often convenient to qualify a command or definition by a local definition, introduced by the symbol **where**; for example,

$$z := ax + by + Fn[ax - by]$$
$$\textbf{where } Fn[z] = (z = 0) \rightarrow 1, Log[Sin[(z + 1)/2]]$$

The effect of this **where** clause is to define a function *Fn* which is valid only within the preceding command. This technique is very useful when a function is defined in terms of an auxiliary function. For example, to reverse a list we can define

$$\begin{aligned}\textbf{let } Rev[x] = {} & Rev1[x, \text{NIL}] \\ & \textbf{where } Rev1[x, y] = Null[x] \rightarrow y, \\ & \qquad\qquad Rev1[Tl[x], Cons[Hd[x], y]]\end{aligned}$$

Another use of the **where** clause is to simplify a complicated condition. For example, suppose it is required to define a function $Tp[x]$ whose value is $Tl[x]$ if the second item on the list x is the symbol $<$ or the symbol $>$, and is otherwise $Tp[Tl[x]]$. A possible definition would be

$$\begin{aligned}\textbf{let } Tp[x] = {} & (Hd[Tl[x]] = \text{`}<\text{'}) \vee (Hd[Tl[x]] = \text{`}>\text{'}) \rightarrow Tl[x], \\ & Tp[Tl[x]]\end{aligned}$$

(Note the use of quotes to indicate a symbol standing for itself, not as a name for something else.) The definition can be simplified to produce

$$\begin{aligned}\textbf{let } Tp[x] = {} & (p = \text{`}<\text{'}) \vee (p = \text{`}>\text{'}) \rightarrow Tl[x], Tp[Tl[x]] \\ & \textbf{where } p = Hd[Tl[x]]\end{aligned}$$

This has the incidental advantage of avoiding unnecessary evaluation of $Hd[Tl[x]]$.

3. ILLUSTRATIVE EXAMPLES

3.1. *Function definitions*

(i)
$$\begin{aligned}\textbf{let } HCF[n, m] = {} & m > n \rightarrow HCF[m, n] \\ & m = 0 \rightarrow n \\ & HCF[m, Rem[n, m]]\end{aligned}$$

This evaluates the *HCF* of two integers n and m using the Euclidean algorithm. $Rem[n, m]$ is a function whose value is the remainder when n is divided by m.

(ii)
$$\begin{aligned}&\textbf{let } Sqrt[x] = S1[x, (1 + x)/2] \\ &\quad \textbf{where } S1[x, y] = y^2 - x \ll 1 \rightarrow y, \\ &\qquad\qquad S1[x, (y + x/y)/2] \\ &\textbf{let } Sqrt[x] = S2[(1 + x)/2] \\ &\quad \textbf{where } S2[y] = z \ll y \rightarrow y, S2[z + y] \\ &\qquad\qquad \textbf{where } z = (x/y - y)/2\end{aligned}$$

These are two functions for evaluating a square root by the well known iteration

$$y_{n+1} = \tfrac{1}{2}(y_n + x/y_n).$$

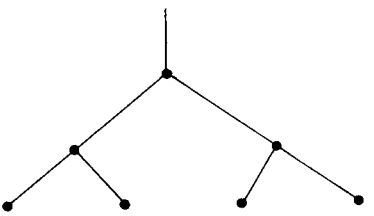
The second version includes the "tricks" usually incorporated in a machine-code program: it shows that functional programs are not necessarily inefficient.

(iii)
```
let Copy1[x] = Null[x] → NIL,
               Atom[x] → x,
               Cons[Copy1[Hd[x]], Copy1[Tl[x]]]
let Copy2[x] = result of
      § let y = Cons[NIL NIL]
        Hd[y] := Atom[Hd[x]] → Hd[x], Copy2[Hd[x]]
        Tl[y] := Null[Tl[x]] → NIL, Copy2[Tl[x]]
        result := y §
```

The first of these is an "orthodox" recursive function to form a copy of a list. The second function is an interesting mixture since although it is in command form, each command causes the function to be used recursively.

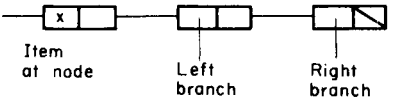
3.2. *Routines—The Tree Sort*

Suppose it is required to sort a set of atoms, which have some natural numerical ordering, into ascending numerical order. This can be done by arranging them in a "tree". A tree is a list structure in which each unit is pointed to by one unit and itself points to two other units, the *left branch* and *right branch* as in the simple picture



We shall call the branch points *nodes*. The objects to be sorted are placed at the nodes of the tree, in such a way that by following the left branch from a node only items less than the item at the node are found, whilst by following the right branch only items greater than or equal to the item at the node are found. The sorting procedure therefore goes in two phases: during the first the tree is constructed, and during the second the items from the tree are printed in correct order.

A node is represented by a list of three items, in the form



A non-existent left or right branch is indicated by a NIL in the appropriate position.

An item x is added to a tree z by the routine

```
routine Add[x, z]
ref z
§1 §2 test Null[z] then do §3 z := Node[x]; Return §3
                or test x < First[z] then do Add[x, Second[z]]
                                    or do Add[x, Third[z]] §2
   where First[z] = Hd[z]
   and Second[z] = Hd[Tl[z]]
   and Third[z] = Hd[Tl[Tl[z]]]
   and Node[z] = Cons[z, Cons[NIL, Cons[NIL, NIL]]] §1
```

The routine is straightforward. It examines the tree, taking the left or right branch at each node as appropriate until it meets an empty branch, and inserts the new item there with both its left and right branches empty. Note that z is called *by reference*: this is vital since ultimately an assignment is made which changes the tree. (Note the use of section brackets §2 and §2 to indicate what is to be qualified by the definitions introduced by **where**.)

If a standard output routine which prints an atom is assumed to exist, and z is the start of the tree, then the tree once constructed is printed by the routine

```
routine PT[z]
§ if Null[z] then Return
  PT[Hd[Tl[z]]]
  Print[Hd[z]]
  PT[Hd[Tl[Tl[z]]]] §
```

The reader is advised to follow through the working of the routine $PT[z]$ on a simple example. (Return indicates the dynamic end of the program comprising the routine: all routines are understood to finish with "Return.")

3.3. *Some Useful Functions and Routines*

(a) *Substitution*

$$\begin{aligned}\textbf{let } Subst[x, y, L] = {} & Null[L] \rightarrow \text{NIL},\\ & Atom[L] \rightarrow (Eq[L, y] \rightarrow x, L),\\ & Cons[Subst[x, y, Hd[L]],\\ & \qquad Subst[x, y, Tl[L]]]\end{aligned}$$

This is a function which replaces y by x wherever it occurs in list L. Alternatively we could have

$$\begin{aligned}&\textbf{let } Subst[x, y, L] = f[L]\\ &\quad \textbf{where } f[L] = Null[L] \rightarrow \text{NIL},\\ &\qquad\qquad Atom[L] \rightarrow (Eq[L, y] \rightarrow x, L),\\ &\qquad\qquad Cons[f[Hd[L]], f[Tl[L]]]\end{aligned}$$

Here $f[L]$ is an "x-for-y substituter": it is an example of a function using 'global' variables.

(b) *Map*

$$\begin{aligned}\textbf{let } Map[f, L] = {} & Null[L] \rightarrow \text{NIL},\\ & Cons[f[Hd[L]], Map[f, Tl[L]]]\end{aligned}$$

This function applies the function f to all elements of a list L so that if

$$L = a, b, c, \ldots, k$$

then
$$Map[f, L] = f[a], f[b], f[c], \ldots, f[k]$$

For example if N is the list $1, 2, 3, \ldots, n$, then

$$Map[Sq, N] \textbf{ where } Sq[x] = xx$$

produces as its value the list $1, 4, 9, \ldots, n^2$. Notice that in the definitions of *Subst* and *Map* it is necessary to take account of the special case where the list is a null list.

(c) *List iteration*

The list iteration function can be used to describe a whole class of iterative procedures. Its definition is given by

$$\begin{aligned}\textbf{let } Lit[F, z, L] = {} & Null[L] \rightarrow z,\\ & F[Hd[L], Lit[F, z, Tl[L]]]\end{aligned}$$

For example suppose N is the list of the integers

$$(1, 2, 3, \ldots, n)$$

then

$$Lit[F1, 1, N] \textbf{ where } F1[x, y] = x \times y$$

gives $n!$ The detailed action is

$$\begin{aligned} Lit[F1, 1, (1, 2, \ldots, n)] &= 1 \times Lit[F1, 1, (2, 3 \ldots n)] \\ &= 1 \times 2 \times Lit[F1, 1, (3, 4 \ldots n)] \\ &= \ldots \qquad\qquad \ldots \\ &= 1 \times 2 \times 3 \ldots \times n \times 1 = n! \end{aligned}$$

Similarly

$$Lit[F2, 0, N] \textbf{ where } F2[x, y] = x + y \quad \text{is } \sum_1^n r$$

$$Lit[F3, 0, N] \textbf{ where } F3[x, y] = 1/x^3 + y \quad \text{is } \sum_1^n 1/r^3$$

The reader is advised to verify these results for himself.

Other examples of list iteration are

$$Lit[Cons, List1[x], L] = Ap[L, x]$$

$$\S\ Lit[F, \text{NIL}, L] \textbf{ where } F[x, y] = Ap[y, x] \ \$ = Rev[L]$$

$$\S\ Lit[F, \text{NIL}, L] \textbf{ where } F[x, y] = Cons[g[x], y] \ \$ = Map[g, L]$$

3.4. *The "Efficiency" of Functional Programs*

In section (3.1) there were given two different functional programs for evaluating a square root, one of which incorporated all the "tricks" that are usually included to speed up a square-root routine, thus showing that functional programs are not necessarily inefficient. As a further illustration of efficient and inefficient programs consider two functions for reversing a list given by

(i) **let** $Rev1[x] = RevA[x, \text{NIL}]$

where $RevA[x, y] = Null[x] \rightarrow y,$

$RevA[Tl[x], Cons[Hd[x], y]]$

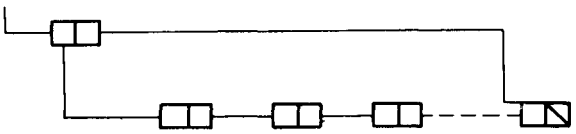
(ii) **let** $Rev2[x] = Null[x] \rightarrow \text{NIL},$

$Ap[Rev2[Tl[x]], Hd[x]]$

where $Ap[L, x]$ puts x as an extra item at the end of list L.

*Rev*2 is more straightforward to follow, but in a LISP-like representation it is expensive, since each application of *Ap* involves copying the entire list with an extra item at the end. *Rev*1, though less immediately obvious, is faster since it makes use of *Cons*, which adds an extra item on the front of a list. Note, however, that this is entirely a matter of representation of the lists. There are schemes in which adding items at either end can be done with

equal facility, as shown in the figure, and in such a scheme there would be nothing to choose between *Rev*1 and *Rev*2 as regards efficiency.



4. COMPLETE PROGRAMS

4.1. *The Railway Time-table Problem*

Given the departure and arrival times of all trains between Cambridge and London, the problem is to compile a timetable of *useful* trains, that is to say trains which take less than two hours and are not overtaken by another train on the journey. The initial data consists of pairs of numbers, being departure and arrival times expressed in minutes after midnight. All trains which go to King's Cross come first in the input data, followed by the trains to Liverpool Street, so some rearrangement is necessary. The first item on the data tape is n, the total number of trains.

The program falls into three sections. In the first, two arrays D and A are set up, holding the departure and arrival times. In the second, an array T is set up containing the departure times of acceptable trains, and in the third section the elements of array T are sorted into order and printed. The heart of the problem is in the second phase, which is discussed in some detail.

An acceptable train must satisfy two conditions; it must be fast, and it must not be overtaken. Therefore we define two predicate functions representing these conditions. The first is quite simple:

$$\textbf{let } Fast[i] = A[i] - D[i] < 120$$

We must take account of the fact that a train may leave before midnight and arrive after midnight, therefore this definition is amended to

$$\begin{aligned}\textbf{let } Fast[i] &= A'[i] - D[i] < 120\\ \textbf{where } A'[i] &= A[i] > D[i] \rightarrow A[i],\\ &\qquad A[i] + 1440\end{aligned}$$

Notice that $A'[i]$ is a function: this illustrates the close analogy between functions and array-element selecting mechanisms.

To establish the second condition we need an auxiliary function $OT[i, j]$ which has the value **true** if train i is overtaken by train j:

$$\begin{aligned}\textbf{let } OT[i, j] &= (D[j] > D[i]) \wedge (A'[j] < A'[i])\\ \textbf{where } A'[i] &= A[i] > D[i] \rightarrow A[i],\\ &\qquad A[i] + 1440\end{aligned}$$

Using this we define the function *Fleet*[*i*] which has the value **true** if no other train overtakes train *i*: this is given by

```
let Fleet[i] = result of
        § let b = false
          for j = 1 to n do
              if OT[i, j] go to END
          b := true
    END: result := b §
```

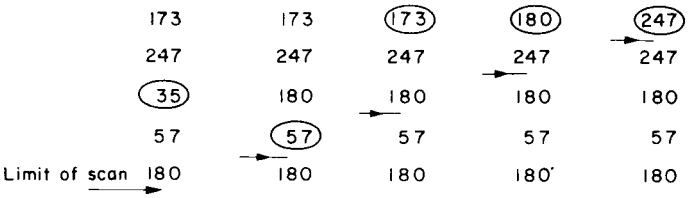
Train *i* is therefore acceptable if *Fast*[*i*] ∧ *Fleet*[*i*] = **true**.

The complete program looks like

```
§1 let n be index
   Read[n]
   §2 let A = Array[(1, n)]
      let D = Array[(1, n)]
      let T = Array[(1, n)]
   for i = 1 to n do
      §2.1 Read[D[i]]
           Read[A[i]] §2.1
   §3 let Fast[i] = A'[i] − D[i] < 120
         where A'[i] = A[i] > D[i] → A[i], A[i] + 1440
      let Fleet[i] = result of
      §3.1    let b = false
              for j = 1 to n do
                  if OT[i, j] go to END
              b := true
      END: result := b   §3.1
              where OT[i, j] = (D[j] > D[i]) ∧ (A'[j] < A'[i])
                    where A'[i] = A[i] > D[i] → A[i],
                                  A[i] + 1440
      let j = 1
      for i = 1 to n do
          if Fast[i] ∧ Fleet[i] do T[j], j := D[i], j + 1
   §4 let a, k = 1441, 0
      §4.1 for i = 1 to j − 1 do
              if T[i] < a do a, k := T[i], i
           Print[a]
           T[k], k, a, j := T[j − 1], 0, 1441, j − 1   §4.1
           repeat until j = 1 §1
```

Note that the **repeat until** $j = 1$ qualifies the whole of the preceding compound command starting at §4.1. Note also that the closing section bracket §1 implies the closing of the other blocks, i.e. it is as if §4 §3 §2 §1 were written.

The first section is straightforward, and assembles the data in arrays A and D. The second section constructs a time-table of useful trains in array T as already explained. The third section, which prints the trains in correct order, may require some explanation. It makes use of a very crude algorithm. It scans the array for the smallest element and prints that: it then takes the element from the end of the array, places it in the position previously occupied by the smallest, and repeats the process scanning one element less in the array. An example may help, and in the following the item ringed is the one printed at each stage.



The program fails if a train leaving before midnight is overtaken by one leaving after midnight: it is left as an exercise for the reader to modify the program to deal with this case.

4.2. *Algebraic Differentiation*

The formal differentiation of algebraic expressions is a good example of symbol manipulation, and is used as a demonstration exercise by many list-processing systems. It has the advantage for this purpose of being obviously non-numerical, and a task which the beginner might consider to be difficult. In fact, as we shall show, it is remarkably easy.

For present purposes, an algebraic expression is considered to consist of the letters A to Z combined with the infixed operators $+$, $-$, $\times$ and $/$, and possibly bracketed; examples of expressions are

$$A + B + C$$

$$(A - B) \times (X + Y/X)$$

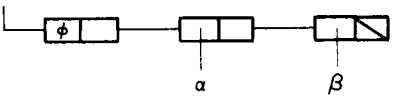
These are to be differentiated with respect to one variable, say X. It is first necessary to decide the representation of the expression in the computer, since with a suitable representation the task of differentiation becomes almost trivial. Any expression of the class introduced can be made up out of sub-expressions of the kind

$$(\alpha\phi\beta)$$

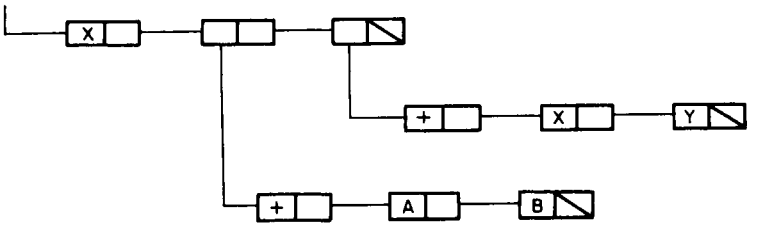
where ϕ is an operator $+$, $-$, $\times$, or $/$, and α, β are either letters or similar bracketed sub-expressions. Thus

$$A + B + C \quad \equiv \quad ((A + B) + C)$$
$$(A - B) \times (X + Y/X) \equiv ((A - B) \times (X + (Y/X)))$$

Such a sub-expression can be represented by a list of 3 items, in the form



If α is an atom it will appear as the head of the second item, otherwise this will point to another 3-list. For example $(A + B) \times (X + Y)$ would be represented by



The virtue of this representation is that it is made up of a set of sub-elements identical in form, and is therefore well adapted to processing by a recursive technique.

We leave aside, for the moment, the question of how to convert an algebraic expression into this form, and consider the program for differentiation.

The function *Diff*[*z*] defined below has as its value the differential coefficient with respect to X of the expression represented by the 3-list z.

```
let Diff[z] =
  Atom[z] → (z = 'X' → '1', '0'),
  result of
    § let p, q, r = Hd[z], Hd[Tl[z]], Hd[Tl[Tl[z]]]
    result := (p = '+') ∨ (p = '−') → List3[p, Diff[q], Diff[r]],
      p = '×' → List3['+', List3 ['×', q, Diff[r]],
                           List3 ['×', r, Diff[q]]],
      List3['−', List3['/', Diff[q], r],
                 List3['/', List3['×', q, Diff[r]],
                            List3['×', r, r]]] §
```

This definition is just a complicated conditional expression. The basic form is

$$Atom[z] \rightarrow E1, E2$$

where $E1$ is the conditional expression

$$(z = \text{'}X\text{'} \rightarrow \text{'1'}, \text{'0'})$$

and $E2$ is a **result of** expression in which **result** is set equal to a conditional expression. Thus if z is the atom X the value is 1, if z is any other atom the answer is zero. Otherwise z must be a list representing a sub-expression, and this is differentiated by the normal rules, using the function *Diff* recursively to differentiate the components of the sub-expression. The extension of the program to deal with exponentiation (denoted by $\uparrow$) should now be obvious.

The list-structure produced by the above function will contain a good deal of redundant information in most cases. For example, given X/A to differentiate it will produce $1/A - X \times 0/A \times A$, given $X \times Y/X$ it will produce $1 + 0/X - Y \times 1/X \times X$, and given $(((A \times (X \times X)) + B \times X) + C)$ it will produce $A \times (X \times 1 + 1 \times X) + 0 \times (X \times X) + B \times 1 + 0 \times X + 0$.

The output can be tidied up by another function *Edit*[*z*], which applies the rules

$$\begin{aligned} x + 0 &= x \\ x - 0 &= x \\ x + x &= 2 \times x \\ x - x &= 0 \\ x \times 0 &= 0 \\ x \times 1 &= x \\ 0/x &= 0 \\ x/1 &= x \end{aligned}$$

```
let Edit[z] =
  Atom[z] → z,
  result of
    § let p, q, r = Hd[z], Hd[T1[z]], Hd[T1[T1[z]]]
      q, r := Edit[q], Edit[r]
      result := p = '×' → (q = '0' → '0',
                           r = '0' → '0',
                           q = '1' → r,
                           r = '1' → q,
                           List3[p, q, r]),
```

$$
\begin{aligned}
&p = \text{'+'} \rightarrow (q = \text{'0'} \rightarrow r,\\
&\qquad\qquad r = \text{'0'} \rightarrow q,\\
&\qquad\qquad q = r \rightarrow \mathit{List3}[\text{'}\times\text{'}, \text{'2'}, q],\\
&\qquad\qquad \mathit{List3}[p, q, r]),\\
&p = \text{'}-\text{'} \rightarrow (r = \text{'0'} \rightarrow q,\\
&\qquad\qquad q = r \rightarrow \text{'0'},\\
&\qquad\qquad \mathit{List3}[p, q, r]),\\
&(q = \text{'0'} \rightarrow \text{'0'},\\
&\ r = \text{'1'} \rightarrow q,\\
&\ \mathit{List3}[p, q, r])\ \ \$
\end{aligned}
$$

Once again the function definition consists of a complicated conditional expression. Note the recursive use of the function at the beginning of the **result of** body, and the structure of the conditional expression in which, corresponding to each condition, there is another conditional expression. The function *Edit* removes most of the obvious redundancies from the output of *Diff*; for example it reduces the output from differentiating $(((A \times (X + X)) + B \times X) + C)$ to $(B + ((2 \times A) \times X))$. These functions can of course be used repeatedly. *Diff*[*Diff*[*z*]] gives the second derivative; to avoid unnecessary work it would be better to use *Diff*[*Edit*[*Diff*[*z*]]], and to get a tidy answer one would use *Edit*[*Diff*[*Edit*[*Diff*[*z*]]]]. Still better, one might define a new function

let *Diff*2[*z*] = *Diff*1[*Diff*1[*z*]]
where *Diff*1[*z*] = *Edit*[*Diff*[*z*]]

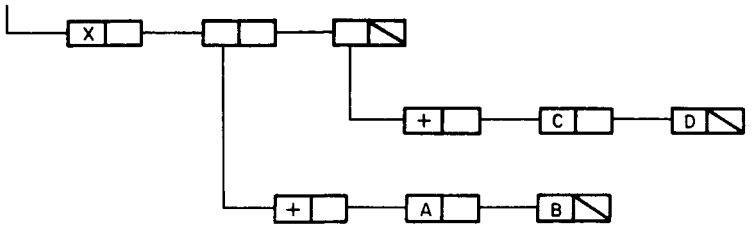
It is now necessary to consider the transformation between the list-structure representation and conventional representations of an expression. We first consider the transformation from a list-structure, which is simpler, and define a routine for the purpose:

define routine *PRL*[*z*]
§1 **if** *Null*[*z*] **then** Return
test *Atom*[*z*] **then** *Print*[*z*] **or**
§2 *Print*['(']
PRL[*Hd*[*Tl*[*z*]]]
Print[*Hd*[*z*]]
PRL[*Hd*[*Tl*[*Tl*[*z*]]]]
Print[')'] §1

This is a simple example of a recursive routine: when called on to output something which is not an atom it outputs an opening bracket, re-enters itself recursively to output the first sub-expression, prints the operator,

re-enters itself to print the second sub-expression, and finally outputs a closing bracket. The reader should follow the action of the routine in printing the following tree structure, verifying that it produces the output

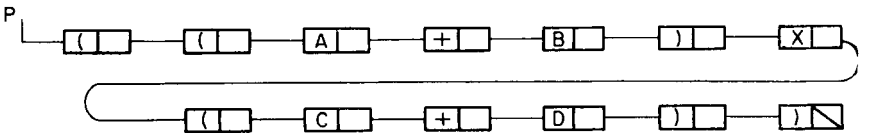
$$((A + B) \times (C + D))$$



Finally we come to the problem of converting a linear symbol string to a list-structure. We assume that the expression to be differentiated is initially in fully bracketed form, and that there exists a routine *RE* which reads an expression from the input device and forms a linear list *P* in which the symbols of the expression are stored in head parts of successive cells. Thus consider the expression

$$((A + B) \times (C + D))$$

The routine *RE* will produce the list



We now define a routine *RNE*[*x*] to convert this into a list structure and assign the name *x* to it.

```
define routine RNE[x]
ref x
§1 let y, z all be logical
   x, P := Hd[P], Tl[P]
   if x = '(' do
   §2 RNE[x]
      y, P := Hd[P], Tl[P]
      RNE[z]
      x, P := List3[y, x, z], Tl[P] §1
```

In following the working of this routine it is important to remember that *x* is called *by reference* and that *y*, and *z* are local variables. Each time the routine is called recursively new local variables are created: we shall distinguish these as *y*1 *z*1, *y*2 *z*2, etc. Since *x* is called by reference, if the routine is called with another parameter, *w* say, it is as if *w* were written in place of

x. Thus for a call $RNE[x]$ that part of the body starting at §2 is given by

```
§2 RNE[x]
   y1, P := Hd[P], Tl[P]
   RNE[z1]
   x, P := List3[y1, x, z1], Tl[P] §2
```

If the call $RNE[z1]$ is effective, then in the recursive entry to the routine this section behaves as if written

```
§2 RNE[z1]
   y2, P := Hd[P], Tl[P]
   RNE[z2]
   z1, P := List3[y2, z1, z2], Tl[P] §2
```

In the analysis of the routine which follows, Roman numerals are used to indicate the depth of recursion, and (a) and (b) identify the recursive calls $RNE[x]$ (line 6) and $RNE[z]$ (line 8) respectively. The routine is entered with $P = ((A + B) \times (C + D))$.

LEVEL	*ACTION*	P
I	$x = ($	$(A + B) \times (C + D))$
	Recursive entry (a) with $y2$, $z2$	
II	$x = ($	$A + B) \times (C + D))$
	Recursive entry (a) with $y3$, $z3$	
III	$x = A$	$+ B) \times (C + D))$
II	$y2 = +$	$B) \times (C + D))$
	Recursive entry (b) with $y3$, $z3$	
III	$z2 = B$	$) \times (C + D))$
II	$x =$ List3$[y2, x, z2]$	
	$= (+, A, B)$	$\times (C + D))$
I	$y1 = \times$	$(C + D))$
	Recursive entry (b) with $y2$, $z2$	
II	$z1 = ($	$C + D))$
	Recursive entry (a) with $y3$, $z3$	
III	$z1 = C$	$+ D))$
II	$y3 = +$	$D))$
	Recursive entry (b) with $y3$, $z3$	
III	$z3 = D$	$))$
II	$z1 =$ List3$[y3, z1, z3]$	
	$= (+, C, D)$	$)$
I	$x =$ List3$[y1, x, z1]$	
	$= (\times, (+, A, B), (+, C, D))$	NIL

4.3 *The Divisors of an Integer*

The problem is, given an integer N, to produce a list of all its divisors. Obviously, it would be possible to do this by dividing in turn by all the integers less than N to see which were exact divisors. However, the program given below uses a method which is more elegant, and is itself instructive in many respects.

The first step is to construct a list of the prime factors of the integer, which is done by the function

$$\begin{aligned}&\textbf{let } PFR[x] = (Rem[x, 2] = 0) \rightarrow Cons[2, PFR[x/2]], PF1[x, 3]\\ &\quad \textbf{where } PF1[x, y] = (x = 1) \rightarrow \text{NIL},\\ &\qquad\qquad (x < y^2) \rightarrow Cons[x, \text{NIL}],\\ &\qquad\qquad Rem[x, y] = 0 \rightarrow Cons[y, PF1[x/y, y]],\\ &\qquad\qquad PF1[x, y + 2]\end{aligned}$$

If x is even we remove 2 as a factor and find the prime factors of $x/2$. The function $PF1$ is almost self-explanatory: if x is less than y^2 it must be a prime factor already; if it is a multiple of y, one factor is y and the process is repeated on x/y; otherwise the process is repeated using the next odd number in sequence for comparison purposes. For example,

$$\begin{aligned}PFR[12] &= Cons[2, PFR[6]]\\ PFR[6] &= Cons[2, PFR[3]]\\ PFR[3] &= PF1[3, 3] = Cons[3, NIL]\end{aligned}$$

which gives $PFR[12] = (2, 2, 3, \text{NIL})$

$$\begin{aligned}PFR[165] &= PF1[165, 3] = Cons[3, PF1[55, 3]]\\ PF1[55, 3] &= PF1[55, 5] = Cons[5, PF1[11, 5]]\\ PF1[11, 5] &= Cons[11, \text{NIL}]\end{aligned}$$

which gives $PFR[165] = (3, 5, 11, \text{NIL})$

Having obtained the prime factors the next step is to obtain the group factors. These are generated by the following process. For each prime factor which occurs once a list is generated consisting of two items, 1 and the factor. For a prime factor which occurs twice a list of three items is generated, containing 1, the factor and its square, and so on. Thus if the prime factors are

$$(2, 2, 3)$$

then the group factors are

$$((1, 2, 4), (1, 3))$$

and if the prime factors are

$$(2, 2, 3, 3, 3, 7)$$

the group factors are

$$((1, 2, 4), (1, 3, 9, 27), (1, 7))$$

Having obtained the group factors in this form, the divisors can be obtained by forming all the possible products generated by taking one item from each sub-list. For example for the group factor list ((1, 2, 4), (1, 3)) the divisors are 1×1, 1×3, 2×1, 2×3, 4×1, 4×3.

We can now produce a function which produces the group factor list from the prime factor list. For clarity, we use the bracket notation for lists, so that $(x) = Cons[x, \text{NIL}]$, $((x)) = Cons[Cons[x, \text{NIL}], \text{NIL}]$, $(x,y) = Cons[x, Cons[y, \text{NIL}]]$, and the function is given by

$$\textbf{let } GF[f] = Null[f] \rightarrow ((1)), GF1[Hd[f], (1), \text{NIL}, f]$$

$$\textbf{where } GF1[x, k, z, f] =$$

$$Null[f] \rightarrow Cons[k, z],$$

$$(Hd[f] = x) \rightarrow GF1[x, Cons[x \times Hd[k], k], z, Tl[f]],$$

$$GF1[Hd[f], (1), Cons[k, z], f]$$

Of the four arguments of the function $GF1$, x is the factor currently reached in the factor list, f is the remainder of the factor list, z builds up to be the group factor list and k builds up as a sublist of powers of a repeated factor. Thus if

$$f = (2, 2, 3, 3, 3, 7)$$

$$\begin{aligned}
GF[f] &= GF1[2, (1), \text{NIL}, (2, 2, 3, 3, 3, 7)] \\
&= GF1[2, (2, 1), \text{NIL}, (2, 3, 3, 3, 7)] \\
&= GF1[2, (4, 2, 1), \text{NIL}, (3, 3, 3, 7)] \\
&= GF1[3, (1), ((4, 2, 1)), (3, 3, 3, 7)] \\
&= GF1[3, (3, 1), ((4, 2, 1)), (3, 3, 7)] \\
&= GF1[3, (9, 3, 1), ((4, 2, 1)), (3, 7)] \\
&= GF1[3, (27, 9, 3, 1), ((4, 2, 1)), (7)] \\
&= GF1[7, (1), ((27, 9, 3, 1), (4, 2, 1)), (7)] \\
&= GF1[7, (7, 1), ((27, 9, 3, 1), (4, 2, 1)), \text{NIL}] \\
&= ((7, 1), (27, 9, 3, 1), (4, 2, 1))
\end{aligned}$$

The divisors are generated from the group factors in two stages. We first generate the "Cartesian Product" of the sub-lists of the group factor list, using the function $Product[L]$ which is discussed in more detail below.

If $L = ((a, b), (p, q, r), (x, y))$

then $L' = Product[L] = ((a, p, x), (a, p, y), (a, q, x), \ldots (b, r, y))$

From this list the divisors are generated by mapping a suitable function onto the list L' in the form

$$\begin{aligned} &Map\ [f, L'] \\ &\textbf{where}\ f[N] = Lit[g, 1, N\ ] \\ &\qquad \textbf{where}\ g[x, y] = x \times y \end{aligned}$$

(The reader will recognize that if N is a list of integers then $f[N]$ is the product of all the integers in the list.) Thus if the functions *Map*, *Lit*, *Product*, *GroupFactors* and *PrimeFactors* are first defined, the function to generate the divisors can be defined as

$$\begin{aligned} &\textbf{let}\ Divisors[x] = Map[f, Product[GroupFactors[PrimeFactors[x]]]] \\ &\qquad \textbf{where}\ f[L] = Lit[g, 1, L] \\ &\qquad\qquad \textbf{where}\ g[x, y] = x \times y \end{aligned}$$

4.4. *The Cartesian Product Function*

This is quite difficult. One definition is given by

$$\begin{aligned} \textbf{let}\ Product[L] = {} & Null[L] \rightarrow List1[\text{NIL}], \\ & Null[Hd[L]] \rightarrow \text{NIL}, \\ & Concat[Map[f, Product[Tl[L]]], \\ & \qquad Product[Cons[Tl[Hd[L]], Tl[L]]]] \\ & \textbf{where}\ f[k] = Cons[Hd[Hd[L]], k] \end{aligned}$$

where $Concat[x, y]$ is a function to concatenate the lists x and y. (See section 4 of Chapter 2.) Consider the operation of this function on a list L where

$$L = ((a, b, c), (p, q), (x, y))$$

We get $Product[L] = Concat[Map[f, Product[((p, q), (x, y))]], Product[((b, c), (p, q), (x, y))]]$

with $f[k] = Cons[a, k] = f_a$, say.

Now, by definition, $Prod[(\text{NIL}, (p, q), (x, y))] = \text{NIL}$

and $Concat[z, \text{NIL}] = z$,

and therefore we get, (using the symbol $\Leftrightarrow$ to denote concatenation of lists),

$$\begin{aligned} & Map[f_a, Product[((p, q), (x, y))]] \\ \Leftrightarrow\ & Map[f_b, Product[((p, q), (x, y))]] \\ \Leftrightarrow\ & Map[f_c, Product[((p, q), (x, y))]] \end{aligned}$$

Similarly, for $Product[((p, q), (x, y))]$

we get $Map[f_p, Product[((x, y))]] \Leftrightarrow Map[f_q, Product[((x, y))]]$

Finally,

$$Product[((x, y))] = Concat[Map[f_x, Product[\text{NIL}]], Product[((y))]]$$

with $f_x[k] = Cons[x, k]$

Now $Product[\text{NIL}] = List1[\text{NIL}]$

therefore $Map[f_x, Product[\text{NIL}]] = Cons[x, List1[\text{NIL}]] = ((x))$

and hence $Product[((x, y))] = Concat[((x)), Product[((y))]] = ((x), (y))$

and so on.

Although the program is ingenious, this is a very inefficient process. More efficient is the process defined by

$$\textbf{let } Product[L] = Null[L] \rightarrow List1[\text{NIL}], F1[Hd[L], Product[Tl[L]]]$$
$$\textbf{where } F1[k, z] = Null[k] \rightarrow \text{NIL}, F2[Hd[k], z, F1[Tl[k], z]]$$
$$\textbf{where } F2[x, y, z] = Null[z] \rightarrow y,$$
$$Cons[Cons[x, Hd[z]], F2[x, y, Tl[z]]]$$

This process can be written more elegantly in the form

$$\textbf{let } Product[L] = Lit[f, List1[\text{NIL}], L]$$
$$\textbf{where } f[k, z] = Lit[g, \text{NIL}, k]$$
$$\textbf{where } g[x, y] = Lit[h, y, z]$$
$$\textbf{where } h[p, q] = Cons[Cons[x, p], q]$$

5. FUNCTION EVALUATING MECHANISMS

A list-processing computation consists mostly of the evaluation of functions: in implementing this on a conventional computer we are essentially simulating a function-evaluation mechanism. Most such mechanisms are built round the concept of a *stack*, or *nesting store*, or *push-down* store. The stack is a last in—first out store: in the simplest forms items may only be stored at the *top* of the stack, and may only be removed from the top. Such a stack consists of a series of storage cells and a *stack-pointer* S which points to the next available cell. When an item is placed on the stack it is placed in cell S, and S is incremented by 1. When an item is taken from the stack it is taken from cell $S - 1$ and S is decreased by 1. When a stack is used as part of a function-evaluating mechanism, it is arranged that functions take their arguments from the top of the stack, and leave their result there: the only effect of a function is to place a result on the top of the stack, and therefore there is no difference in principle between a function and a simple variable as components of an expression. When used in this way there is associated with the stack another pointer, the *link-pointer* P, which locates the link of the current function. The link contains the "return address" and the value

of P for the current partly evaluated function, so that when the new function has been evaluated and its result placed on the stack, evaluation of the enclosing function can be resumed. For example consider the evaluation of a simple function

$$F[x, y] = (2x + 6y)/xy$$

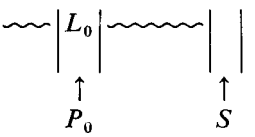
which can be written in prefixed operator form as

$$Div[Add[Mult[2, x], Mult[6, y]], Mult[x, y]].$$

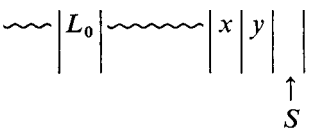
The arithmetic functions *Add*, *Mult*, etc. will take their arguments from the two stack cells preceding the link, and place the result on top of the first of them. Treating the stack as a vector, and identifying the ith cell by $ST[i]$, the operation of *Add* is given by

$$ST[P - 2], P, S := (ST[P - 2] + ST[P - 1]), P_0, P - 1$$

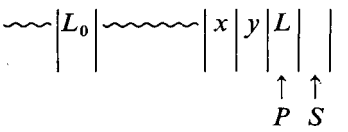
Pictorially, the state of the stack at various stages in the evaluation of $Add[x, y]$ is as follows:



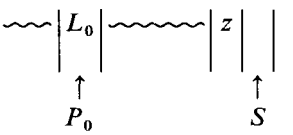
L_0 is the link for the previous function. S is the next available cell



x and y are loaded



Function *Add* is entered: L is the link, which includes P_0



$z = x + y$

Notice that the ultimate effect of $Add[x, y]$ is exactly the same, from the point of view of the enclosing function, as just loading z.

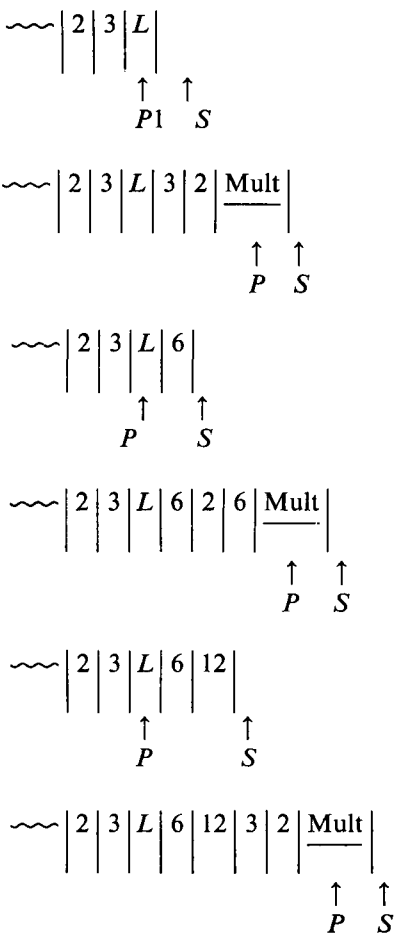
Stack operations are conveniently represented by the 'reversed Polish' notation. For example

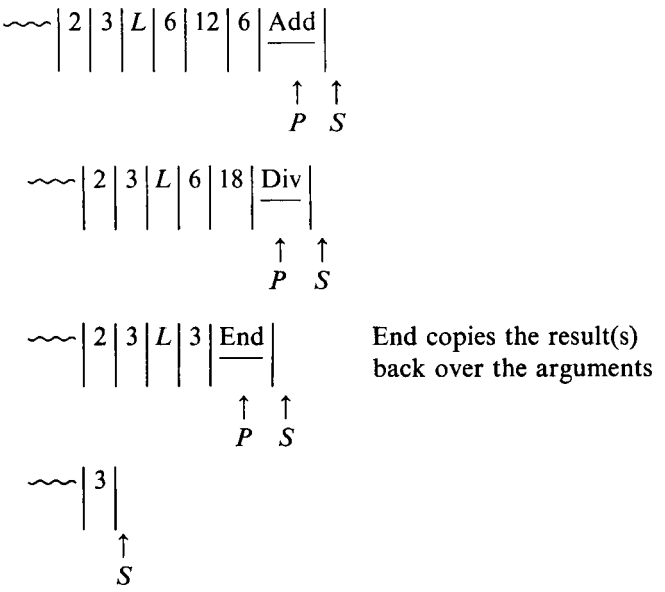
$$f(x, y) = (2x + 6y)/xy$$

becomes

$$f \Rightarrow y,\ x,\ \underline{\text{Mult}},\ y,\ 6,\ \underline{\text{Mult}},\ x,\ 2,\ \underline{\text{Mult}},\ \underline{\text{Add}},\ \underline{\text{Div}},\ \underline{\text{End}}$$

In this string a plain identifier means "load variable to top of stack", and an underlined identifier means "apply function". The reversed Polish string is in fact, as it stands, a program for a stack-based evaluation mechanism. (Notice that the transformation from prefixed-operator notation to a reversed-Polish string is a very simple operation.) For example, consider the evaluation of $f[3, 2]$, which proceeds in the sequence





The reader will notice that we have replaced the simple concept of all operations taking place on the head of the stack by a more general concept where items can be moved about on the stack, being identified relative to a current link.

A function expects to find its arguments on the stack in the cells immediately before the link. If the evaluation of the function requires local variables, these are placed on the stack above the link. This has two advantages. First, they are anonymous, and disappear when the evaluation is completed. Second, if the function is used recursively, then at each new entry a new link goes to the stack and therefore a new set of local variables is automatically created, and there is no conflict of storage requirements.

Consider as a fairly typical example the definition of the factorial function

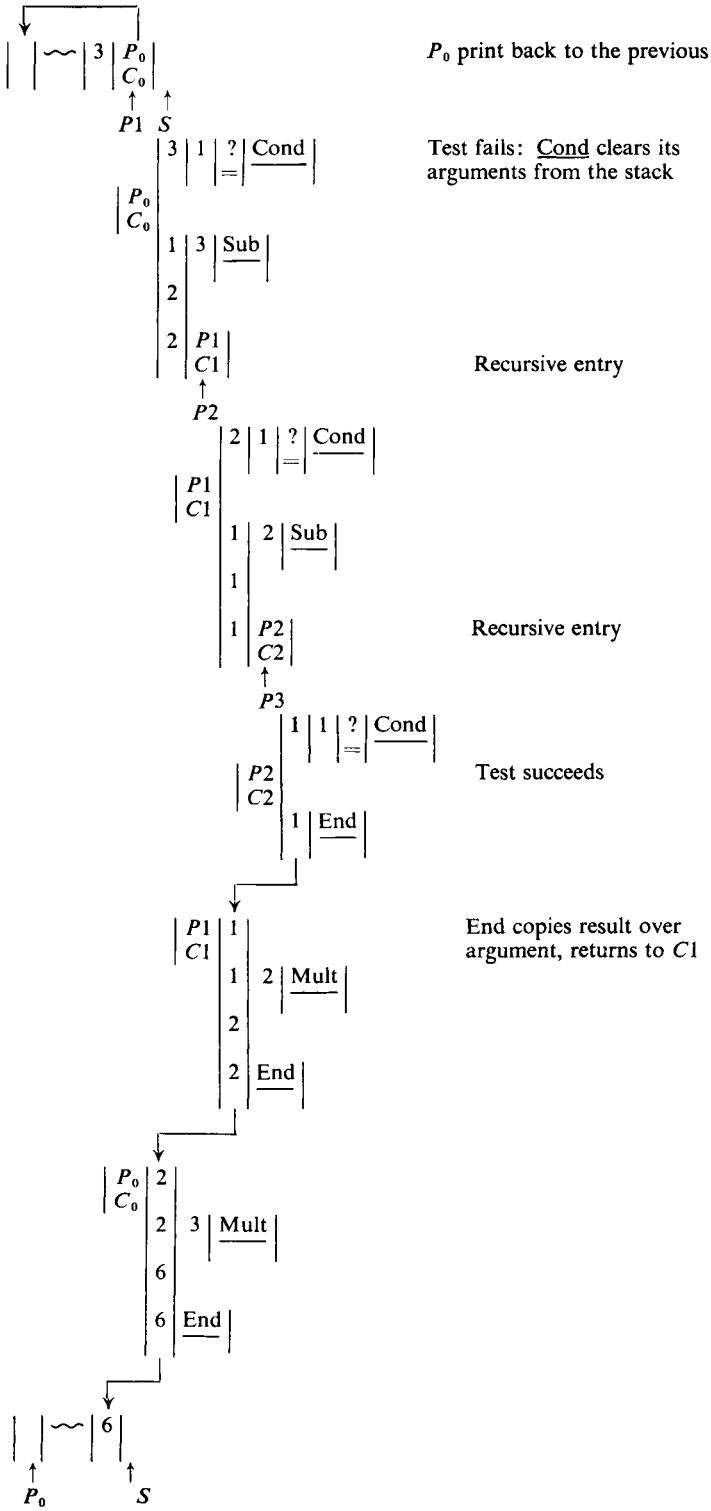
$$\textbf{let}\, f[x] = (x = 1) \rightarrow 1,\ x \times f[x - 1]$$

Using an informal representation of a conditional operator, this can be written as

$$f \Rightarrow x,\ 1,\ ?=,\ \underline{\text{Cond}} \xrightarrow{No} 1,\ x,\ \underline{\text{Sub}},\ f,\ \overset{C1}{\downarrow} x,\ \underline{\text{Mult}},\ \underline{\text{End}}$$

$C1$ is the "return address" to resume evaluation of f after a recursive entry.

The diagram shows the state of the stack at various stages in the evaluation of $f[3]$. Only items at the top of the stack are shown in detail: items to the left of the first on a line are the same as on the preceding line.


P_0 print back to the previous link
Test fails: Cond clears its arguments from the stack
Recursive entry
Recursive entry
Test succeeds
End copies result over argument, returns to $C1$
Cond
Sub
Mult
End
P_0
S
$P1$
$C1$
$P2$
$C2$
$P3$

In the last example the function was defined as a conditional expression involving only the argument and constants. As an example of dealing with a "result of" definition, take the alternative definition of the factorial function, given by

let $f[x]$ = **result of**

§ **let** $x', y = x, 1$

until $x' = 0$ **do** $x', y := x' - 1, x'y$

result $:= y$ §

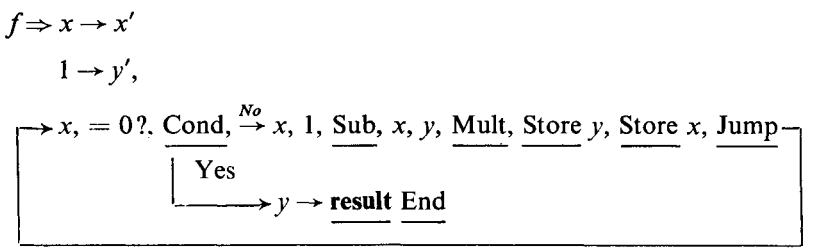
For simplicity we will replace the double assignment controlled by the **until** clause by the equivalent commands

§ $y := x'y$

$x' := x' - 1$ §

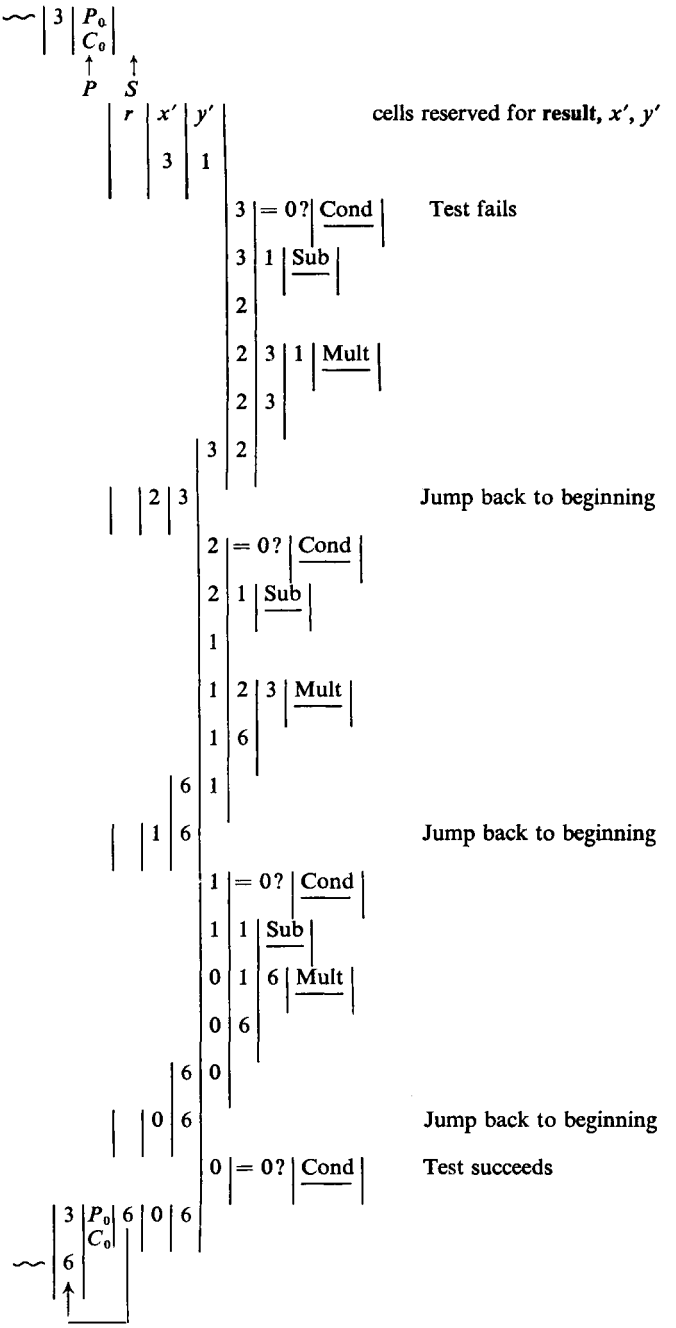
When the function is entered its argument will be on the stack, immediately below the link. A cell will be left unused immediately above the link, in which the **result** can ultimately be placed, and the next two cells will be reserved for the local variables x and y. Whenever there is an assignment to x, say, the corresponding value is placed in the reserved cell on the stack, and therefore during the compiling process, **result**, x and y can be replaced by the so-called "canonical names", $ST[P + 1]$, $ST[P + 2]$, $ST[P + 3]$. The region of the stack used for anonymous intermediate quantities starts at $ST[P + 4]$.

The definition can be represented by



The notation is similar to that used in the previous example: $a \rightarrow b$ means "copy a into cell allocated to b", and Store c means "store item at top of stack in cell allocated to c".

The state of the stack at various stages in the evaluation of $f[3]$ is shown in the diagram.


cells reserved for **result**, x', y'
Test fails
Jump back to beginning
Jump back to beginning
Jump back to beginning
Test succeeds

The discussion has been in terms of function evaluation, but the methods described can be extended to provide a general program-execution mechanism. For example a routine which does not refer to global variables can be treated in exactly the same way as a "result-of" function definition, except that it does not produce a result. More complication ensues if the routine refers to global variables, but the difficulties are not insuperable. Once routines have been satisfactorily dealt with, most programs are catered for, since in a language such as CPL or ALGOL a block can be regarded as a routine without parameters.

CHAPTER 4

ARTIFICIAL LANGUAGES

J. M. FOSTER

1. INTRODUCTION

One of the principal tools for studying a natural language is its grammar. In the grammatical approach to a language we try by means of a fixed set of rules to determine which sentences are properly formed. A particular set of rules will allow a particular set of sentences; we try to choose the rules so that the allowed sentences include all the sentences of the natural language which we feel are proper, and no others. The difficulties of this will be apparent, not least among them being the intuitive determination of the proper sentences which ought to be allowed. No scheme of rules provides an adequate and practical description of a complex natural language, such as English or Chinese. Any set of rules does, however, describe an artificial language which may be useful where a precisely defined language is needed.

Communication with a computer provides an example of such a need. Any program for a computer must be written to conform with a set of rules which rigorously determines legal programs. Similarly the data for a program must be provided in a format which is precisely defined by the program. The languages thus defined will not usually resemble natural languages; they may consist of strings of numbers or look like algebraic notation, but it may be possible to apply to them the techniques devised for modelling natural languages.

One difficulty arises immediately. The form of the rules may be chosen with one of two aims. The rules may enable the legal sentences to be enumerated, that is, by the successive application of the rules in some order it may be possible to generate any legal sentence, of which there will usually be an infinite number. Alternatively, given a sentence the rules may decide whether or not it is legal. That these processes are not identical is well known; indeed one of the schemes to be described provides a classic case of an undecidable proposition (Post, 1947; Markov, 1947). Even where it is not impossible, deciding whether a sentence is legal may be a very long process. Schemes for natural languages have usually been cast in the first of these ways, whereas for computer purposes we need to be able to decide whether a

particular program or set of data is correct, and if so to find out how it was built up.

Some of the ways of synthesizing artificial languages and some of their properties are described first and the problems of analysing them discussed later.

2. DESCRIPTIONS OF ARTIFICIAL LANGUAGES

The schemes which are described in this section are based on ideas which have long been current, and the classification used here was first given by Chomsky (1957, 1959). Suppose we have a grammatically correct English sentence containing a noun phrase. Then all sentences obtained by replacing the noun phrase by other examples of noun phrases will also be grammatically correct. For example the sentence

"Some cows have horns"

contains the noun phrase "Some cows". The substitution of the noun phrases "Horses", "All unicorns" and "Green thoughts" yields

"Horses have horns"
"All unicorns have horns"
"Green thoughts have horns"

all of which are grammatical. Notice that a noun phrase can contain a noun phrase. In the sentence

"No cows with blue spots have horns"

the noun phrase "No cows with blue spots" contains the noun phrase "blue spots" which is made up of the adjective blue and the noun phrase spots.

The rules are expressed in terms of this idea of substitution. We are given a finite set of substitution rules such that if a particular combination of elements occurs in a legal sentence the sentence obtained by replacing it by another combination of elements is also legal, and all legal sentences are generated thus from a single sentence which is defined to be legal. When a sentence has been reached to which no further rules can be applied it is said to be terminal.

As an example we give a set of rules for producing all ways of inserting brackets into strings of the letter a so that between matching brackets there are just two sub-formulae. The rules are

$$\begin{aligned} S &\rightarrow a \\ S &\rightarrow (SS) \end{aligned} \tag{2.1}$$

where S is a legal sentence. The interpretation of these productions is that the string of symbols on the right can replace the string on the left, so that

$$
\begin{array}{l}
S \\
(SS) \\
(S(SS)) \\
(a(SS)) \\
(a((SS)S)) \\
(a((aS)S)) \\
(a((aa)S)) \\
(a((aa)a))
\end{array}
$$

is an example of a derivation of the string $(a((aa)a))$ from S. This will be denoted by

$$S \Rightarrow (a((aa)a)).$$

The string $(a((aa)a))$ is terminal.

Chomsky divides grammars into four types according to the forms of the rules. In type 0 there is no restriction on the rules, which may have any strings on either the right or the left of the substitution arrow. Thus

$$aBbcDD \rightarrow fgHxa$$

would be an allowed rule with the meaning that if $aBbcDD$ occurs in a legal string it may be substituted by $fgHxa$ giving another legal string. Type 0 languages are too general to be useful and the problem of deciding whether a particular string is legal is in general undecidable. These languages will not be discussed further.

In what follows we shall reserve capital letters to denote symbols which can be substituted and the lower-case letters and arithmetic operators for those symbols which cannot be substituted. Terminal strings will consist of lower-case letters and arithmetic operators, and strings containing capital letters will not be terminal. Greek letters will be used to stand for strings of symbols.

In type 1 languages the rules are restricted to the form

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

where A is a single symbol, α and β are particular strings which may be empty, and γ is a non-empty string of symbols. This can be read "A can be substituted by γ in the context α, β".

Type 2 languages, which form the most used class, are defined by the restriction that all rules must have the form

$$A \rightarrow \gamma$$

where A is a single symbol and γ a non-empty string of symbols. The language (2.1) is an example of this class.

In type 3 languages the rules are of the form

$$A \to a$$

or

$$A \to bB$$

where A, B, a and b are single symbols.

It is easy to see from the definitions that type 0 languages include type 1 which include type 2 which include type 3. Examples can also be found of type 0 languages which are not type 1, of type 1 languages which are not type 2 and of type 2 languages which are not type 3. Since these give some feeling for the significance of the classification two examples will be discussed.

Let us denote by $a^p b^q$ the language consisting of all strings which have the form of p occurrences of a followed by q occurrences of b for all $p, q \geqslant 1$ and of no other strings. We prove the following theorem.

THEOREM 1. *The language $a^n b^n$ is not a type 3 language.*

Proof. Suppose that we have a type 3 grammar G for $a^n b^n$. Let us derive a new grammar G′ from this in the following way. If G contains any symbol A which occurs only in rules of the form

$$A \to a_1,\ A \to a_2, \text{ etc.}$$

and not in rules of the form

$$A \to bC$$

then remove the rules $A \to a_1$, etc., from the grammar and replace occurrences of A on the right-hand side of rules by a_1, a_2, etc., in turn. Repeat this until all non-terminal symbols occur in at least one rule of the form

$$A \to bC$$

Call the grammar that results G′. It is clear that G′ is equivalent to G.

Consider the derivation of some particular element, say *aaabbb*, in G′. The last step was

$$aaabbX$$

$$aaabbb$$

for some X. But there must be at least one rule of the form

$$X \to yZ$$

Therefore

$$aaabbyZ$$

is a legal string. Since Z must give rise to a terminal string of length at least one the language determined by G′ includes

$$aaabby\alpha$$

for some y and α. This cannot belong to a^nb^n. Hence the language a^nb^n is not a type 3 language, since any type 3 grammar which gives all the strings a^nb^n must also give other strings.

Note that a^nb^n is a type 2 language given by the rules

$$S \rightarrow ab$$

$$S \rightarrow aSb$$

This result is of importance because it indicates that languages of type 3 cannot express the proper matching of brackets. Since this is one of the things we are likely to want to do we must use languages of types 1 or 2.

It is also possible to prove on similar lines the following theorem.

THEOREM 2. *The language $a^pb^qa^pb^q$ is not a type 2 language.*

A proof is given by Chomsky (1959). The significance of this result will be discussed later.

Below are given a number of examples of type 2 and type 3 languages and their rules. In all of them S is defined to be a legal sentence.

Examples

Type 3

(i) The language a^n.

$$S \rightarrow a$$

$$S \rightarrow aS$$

(ii) The language a^pb^q.

$$S \rightarrow aS$$

$$S \rightarrow aT$$

$$T \rightarrow b$$

$$T \rightarrow bT$$

(iii) All algebraic expressions formed from the letters a and b and the arithmetic operators $+$ and $\times$, for example $a + b \times a \times a$.

$$\begin{aligned} S &\rightarrow a \\ S &\rightarrow b \\ S &\rightarrow aT \\ S &\rightarrow bT \\ T &\rightarrow +S \\ T &\rightarrow \times S \end{aligned} \tag{2.2}$$

(iv) All combinations of a and b starting with a.

$$\begin{aligned} S &\to a \\ S &\to aT \\ T &\to a \\ T &\to aT \\ T &\to b \\ T &\to bT \end{aligned}$$

(v) The language a^{2n+1}

$$\begin{aligned} S &\to a \\ S &\to aaS \end{aligned}$$

Type 2

The following languages could be expressed with a type 3 grammar but are given here in type 2.

(vi) All algebraic expressions formed from the letters a and b and the signs $+$ and $\times$.

$$\begin{aligned} S &\to T \\ S &\to S + T \\ T &\to P \\ T &\to T \times P \\ P &\to a \\ P &\to b \end{aligned} \tag{2.3}$$

(vii) All combinations of a and b starting with a.

$$\begin{aligned} S &\to a \\ S &\to Sa \\ S &\to Sb \end{aligned}$$

(viii) The language a^{2n+1}.

$$\begin{aligned} S &\to a \\ S &\to aSa \end{aligned} \tag{2.4}$$

The following languages cannot be expressed with type 3 grammar.

(ix) All algebraic expressions formed from letters a and b, the arithmetic operators $+$ and $-$ and round brackets.

$$\begin{aligned} S &\to T \\ S &\to S + T \\ T &\to P \\ T &\to T \times P \\ P &\to (S) \\ P &\to a \\ P &\to b \end{aligned}$$

(x) The language formed from any strings of a's and b's followed by its mirror image.

$$S \rightarrow aa$$
$$S \rightarrow bb$$
$$S \rightarrow aSa$$
$$S \rightarrow bSb$$

(xi) The language ALGOL.

A set of productions to define the grammar of ALGOL is given by Naur *et al.* (1960).

3. THE ANALYSIS OF SENTENCES BELONGING TO ARTIFICIAL LANGUAGES

If sentences from artificial languages are to be used for communicating with computers we need at least to be able to decide whether a sentence is legal and preferably to deduce the way in which it was built up. In type 0 languages this cannot in general be done. In languages with grammars of types 1, 2 or 3, however, it can be proved to be possible. Suppose we are given a grammar of type 1, 2 or 3. An equivalent grammar can be produced which contains no rules of the form

$$\alpha A \beta \rightarrow \alpha B \beta$$

by carrying out substitutions on the lines of those made in the proof of Theorem 1. Then the application of any rule either increases the total length of the string or has the form

$$\alpha A \beta \rightarrow \alpha a \beta$$

Hence by application of the rules in an appropriate order we can enumerate all the strings belonging to the language and having length not greater than n for any particular n. If we now have a sentence of length p which we wish to test, we can compare it with a list of legal sentences of length p and know that it is correct, if and only if it is included in the list. Although this shows the possibility of deciding the legality of a string it is clearly not a practicable method.

Irons (1961) has written a program for translating from languages whose grammar is of type 2 into computer machine language using a method based on the analysis of the grammatical structure. In particular this program can be used to translate ALGOL into machine code. Programs do exist which can analyse any type 2 language, but these general programs are very slow in operation and practical schemes such as those of Irons (1961) and Paul (1962) will not accept all type 2 languages.

As an example we show how the language given by the productions (2.3) could be translated into an imagined machine code with the understanding

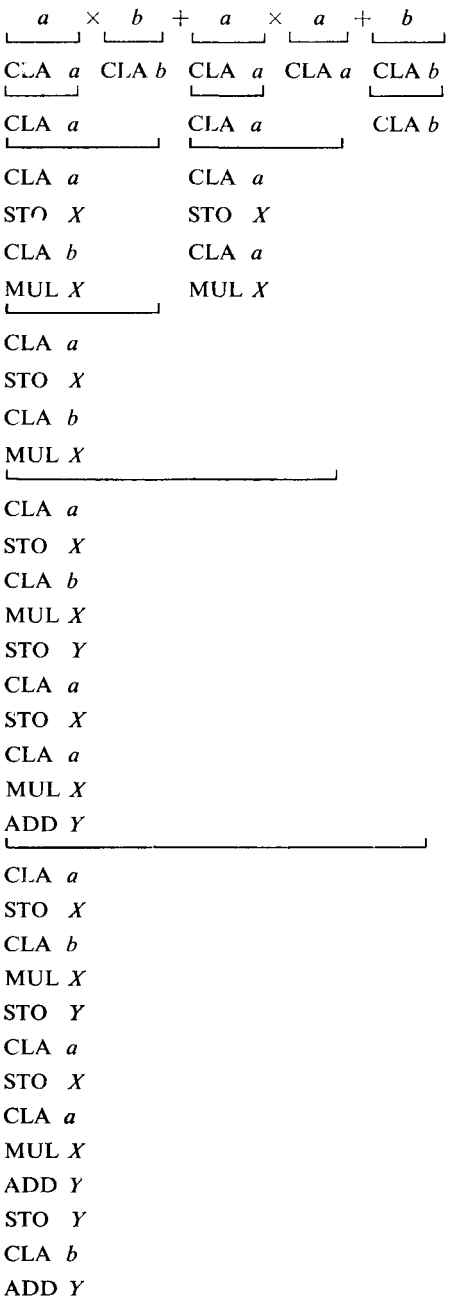

a × b + a × a + b
CLA a CLA b CLA a CLA a CLA b
CLA a CLA a CLA b
CLA a CLA a
STO X STO X
CLA b CLA a
MUL X MUL X
CLA a
STO X
CLA b
MUL X
CLA a
STO X
CLA b
MUL X
STO Y
CLA a
STO X
CLA a
MUL X
ADD Y
CLA a
STO X
CLA b
MUL X
STO Y
CLA a
STO X
CLA a
MUL X
ADD Y
STO Y
CLA b
ADD Y

FIG. 1

that multiplications are to be carried out before additions in cases of ambiguity. In the left column are written the substitution rules and in the right column is written what is to be substituted if the corresponding production is found to have been used.

$S \rightarrow T$	translation of T
$S \rightarrow S + T$	translation of T
	STO Y
	translation of T
	ADD Y
$T \rightarrow P$	translation of P
$T \rightarrow T \times P$	translation of T
	STO X
	translation of P
	MUL X
$P \rightarrow a$	CLA a
$P \rightarrow b$	CLA b

Consider the translation of

$$a \times b + a \times a + b$$

The analysis is

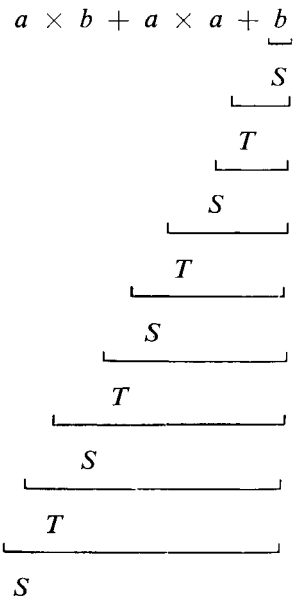
$$\begin{array}{ccccccccc} \underline{a} & \times & \underline{b} & + & \underline{a} & \times & \underline{a} & + & \underline{b} \\ \underline{P} & & P & & \underline{P} & & P & & \underline{P} \\ \underline{T} & & & & \underline{T} & & & & T \\ \underline{T} & & & & T & & & & \\ \underline{S} & & & & & & & & \\ \underline{S} & & & & & & & & \\ S & & & & & & & & \end{array} \tag{3.1}$$

which gives the translation shown in Fig. 1.

The program at the end of Fig. 1 can be seen to evaluate the expression, though it is clearly extremely inefficient. The inefficiency is not due to the

method but to the simplicity of the example. Ledley and Wilson (1962) show how by changing the production rules the translation can be increased in efficiency.

If we are to use the analysis of sentences for such purposes it is most important that the structure of the analysis should reflect the meaning of the sentence. For example the two languages (2.2) and (2.3) both define exactly the same set of sentences, so either could be used purely to decide the legality of a sentence. But if it is required for purposes of translation the grammar (2.3) is much superior to (2.2), since the latter does not reflect the precedence of the multiplication operation over addition. This can be seen by examining the analysis of the following string in the grammar (2.2) and comparing it with the previous analysis in grammar (2.3).



It is also possible to interpret grammar (2.3) as implying that equal operations are carried out from left to right.

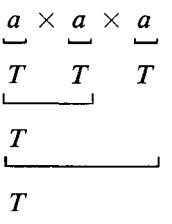
In any language there may be more than one possible analysis of a sentence. If the grammar is to be used for translation this is undesirable. For example the grammar

$$S \rightarrow T$$

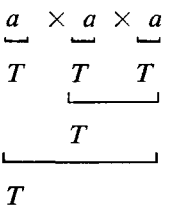
$$T \rightarrow T \times T$$

$$T \rightarrow a$$

gives for $a \times a \times a$ the two analyses



and



A translator might carry out either of the multiplications first.

When an analysis of this sort is to be used as the basis for translating a more complex and practical language, such as ALGOL, the difficulty may arise that not all the restrictions which determine legal programs can be expressed in the grammar. There is no clear-cut place where grammar leaves off and meaning determines the good sentences. The sentence

"Green thoughts have horns"

is a grammatical sentence according to our previous discussion, but the grammar might have been defined to exclude it.

In ALGOL part of the definition of legal programs can be done with a type 2 grammar but part of it is left to be described in words. The program

begin real *aaa* ; *aaa* : = 1 **end**

is legal ALGOL but

begin real *aaaa* ; *aaa* : = 1 **end**

is not. This is expressed by saying that the identifiers used in expressions must be declared. If the statement of Theorem 2 is consulted it will be noticed that there is a potential difficulty in expressing the occurrence of the same string in two different places in a sentence in a type 2 language. Floyd (1962) has shown that ALGOL cannot be completely described by a type 2 grammar for just this reason. This means that to define ALGOL we must either go to a type 1 grammar, which is probably impractical, or we must add some extra restrictions not expressible in the form of productions, most probably in English.

Another example occurs in CPL. Here the statement brackets § and $ can be followed by identifying numbers, for example

$$\S 1 \ldots \S 1.1 \ldots \$ 1.1 \ldots \$ 1$$

If in this example $ 1.1 is omitted the opening bracket labelled 1.1 will be closed just before $ 1. Once again it is necessary to express the fact that two different strings are identical and once again a type 2 grammar is inadequate.

On the other hand some of the features available in type 2 languages are not required for practical purposes. It has been mentioned that programs for analysing all type 2 languages are very slow and that practicable programs will accept only a subset of the possible type 2 languages. An example of a language which gives difficulty is (2.4).

Because of these considerations efforts have been made to find classification of grammars which correspond more closely to what is needed and which are easy to analyse. Floyd (1963) has given a definition of "precedence grammars" which are easy to analyse and which are closer to what is required. ALGOL is not a precedence grammar, though a preliminary scan of an ALGOL program could be made which would turn it into such a grammar.

In conclusion a program is given in the notation of CPL to carry out the translation of language (2.3) as described in (3.1) and Fig. 1. Suppose that a string from the language is presented to the input of the computer in such a way that the procedure *Next* gives in sequence the symbols of the string. If the string is

$$a \times b \times a$$

the first value of *Next* is a, the second $\times$, etc. Let the string be terminated by a full stop. Consider the program

$$\textbf{let } P[x, n] = x \neq \text{“.”} \textbf{ and } w[x] > w[n] \rightarrow P[P[Next, x], n],$$
$$\textbf{result of } \S \text{ write } [n]$$
$$\textbf{result} := x \ \$$$
$$P[Next, \text{“.”}]$$

where

$$w[\text{“}a\text{”}] = 3$$
$$w[\text{“}b\text{”}] = 3$$
$$w[\text{“}+\text{”}] = 1$$
$$w[\text{“}\times\text{”}] = 2$$
$$w[\text{“.”}] = 0$$

This program will write at the output the translation of the input into Reverse Polish.

For example the program is here applied to the string

$$a + b \times a.$$

State of input	State of program	State of output
$a + b \times a.$	—	—
$+ b \times a.$	$P[a, .]$	—
$b \times a.$	$P[P[+, a], .]$	—
$b \times a.$	$P[+, .]$	a
$\times a.$	$P[P[b, +], .]$	a
$a.$	$P[P[P[\times, b], +], .]$	a
$a.$	$P[P[\times, +], .]$	ab
$.$	$P[P[P[a, \times], +], .]$	ab
—	$P[P[P[P[., a], \times], +], .]$	ab
—	$P[P[P[., \times], +], .]$	aba
—	$P[P[., +], .]$	$aba\times$
—	$P[., .]$	$aba\times +$
—	—	$aba\times +.$

The program can be modified to give a translation into machine code of the form described.

```
let t = NIL
let P[×, n] = x ≠ "." and w[x] > w[n] → P[P[Next, x], n],
              result of § t := n = "+" → f["STO Y", "ADD Y"],
                                n = "×" → f["STO X", "MUL X"],
                                n = "a" → Cons["CLA a", t],
                                n = "b" → Cons["CLA b", t],
                                t
              result            := x
              where             f[i, j] = Cons[Cons[s, Cons[i, Cons[r,
                                              Cons[j, NIL]]]], Tl[Tl[t]]]
              where             § r = Hd[t]
                                  s = Hd[Tl[t]] §
P[Next, "."]
Copy [Flatten[t]]
```

Flatten is the procedure described in Chapter 2 and *Copy* is assumed to print out the contents of the list which forms its argument.

4. REFERENCES

CHOMSKY, A. N. (1957) *Syntactic Structures*, Mouton.

CHOMSKY, A. N. (1959) On certain formal properties of grammars, *Information and Control* **2**, 137–67.

FLOYD, R. W. (1962) On the non-existence of a phrase structure grammar for ALGOL 60, *Comm. A.C.M.* **5**, 483–484.

FLOYD, R. W. (1963) Syntactic analysis and operator precedence, *Jour. A.C.M.* **10**, 316–33.

IRONS, E. T. (1961) A syntax directed compiler for ALGOL 60, *Comm. A.C.M.* **4**, 51–5.

LEDLEY, R. S. and WILSON, J. B. (1962) Automatic-programming-language translation through syntactical analysis, *Comm. A.C.M.* **5**, 145–55.

MARKOV, A. A. (1947) On the impossibility of certain algorithms in the theory of associative systems (English Translation), *Comptes Rendue de l'Accademie des Sciences de l'U.R.S.S.* **55**, 583–6.

NAUR, P. (Ed) (1960) Report on the algorithmic language ALGOL 60, *Comm. A.C.M.* **3**, 299–314.

PAUL, M. (1962) A general processor for certain formal languages. Pages 65–74 of *Symbolic Languages in Data Processing*. Gordon and Breach: London.

POST, E. L. (1947) Recursive unsolvability of a problem of Thue, *Journal of Symbolic Logic* **12**, 1–11.

CHAPTER 5

A λ-CALCULUS APPROACH

P. J. LANDIN

The central part of this chapter is contained in Landin (1964). Apart from minor alterations, the new material is the Introduction, section 2 on "Syntax and Semantics", and two Appendices in which the implications of the evaluating process are displayed by means of a detailed example of evaluation, and a detailed example of how the mechanism can be represented in a computer.

1. INTRODUCTION

A considerable amount of effort has been devoted to the analysis of syntax of mechanical languages, and it has yielded results in the shape of more systematic design and processing of languages. This chapter is an introduction to a complementary activity that might perhaps be called "semantic analysis" since it is concerned with characterizing certain aspects of mechanical languages that are not brought out in what is customarily called "syntactic analysis". However, since the use of the terms "syntax" and "semantics" in the computing field is currently controversial, it seems advisable to qualify this remark with a brief account of some of the features that arise when attempting to describe a language. So in the section on "Syntax and Semantics", immediately following this introduction, we briefly summarize the syntactic approach to languages, thus characterizing our present topic, albeit negatively, without relying on previous connotations of words whose usage is still fluid.

Thereafter comes the main body of the chapter, and it falls broadly into two parts each of several sections. The first introduces a technique of semantic analysis, and the second provides a mathematical basis for it. This technique consists of establishing a correspondence between the texts of the language to be analysed and expressions of a structurally simpler language, based on Church's λ-calculi (see McCarthy (1960), Rosenbloom (1950), Church (1941) and Curry and Feys (1958) for introductions to λ-notation of progressively increasing depth and length) for which the problem of "semantic description" is less diffuse. This structurally simpler language comprises certain expressions called here *applicative expressions* (AEs).

Applicative expressions are characterized in abstract terms, without reference to a particular written representation of them. When presenting specific AEs we shall, of course, be compelled to adopt some conventions about how to write them, but we shall do so informally, and it will be convenient to use different conventions, and even different representations of the same AE, on different occasions. There is an analogy here with an axiomatic characterization of, say, natural numbers. Thus a statement about an AE bears the same relation to a particular written representation of it as a statement about a number bears, say to, a Roman, or binary, or decimal representation of it.

We have said that we shall write AEs in various ways. Indeed it is immediately clear that once a correspondence has been established between some familiar notation and AEs, the familiar notation can be considered as a way of writing AEs. This trick needs a point of departure and the one we choose is approximately Church's own notation. It can be viewed as a standard to which all developments must be referred, and whose unpalatable rigour the developments coat with "syntactic sugar." We shall very early be able to grant ourselves the licence of sweetening it with familiar algebraic formulae. It can be seen that the analysis of linguistic features in terms of AEs is a "bootstrap" process. For example, once it is established that auxiliary definitions such as that contained in

$$a + bx + cx^2 \textbf{ where } x = p - 2q$$

can be analysed in terms of AEs, then it is enough in future to analyse things in terms of AEs *plus* auxiliary definitions.

AEs are enough to match some features of current programming languages, namely lists (including operand lists), conditional expressions, auxiliary definitions or declarations (including function definitions and recursive definitions) and the unrestricted local use of names in a nested "scope" structure. They are not enough to match the characteristic features of "imperative" (as opposed to "descriptive") languages, namely assignment and jumping. Thus AEs do not in themselves provide a means of analysis adequate for any current programming language. However, there is reason to believe that they can be extended satisfactorily (see Landin, 1965).

2. SYNTAX AND SEMANTICS

One of the defining features of a language is the class of character-strings that are "admissible", or "meaningful", or "well-formed" texts of it. In so far as this feature is vague the language itself is not fully defined. In what follows we shall refer to a description of this class as a "syntax description" of the language. Conceivably a syntax description might consist of an

enumeration of all admissible texts, but this is unwieldy for any language that concerns us, and, of course, impossible when the admissible texts can be indefinitely long. Syntax descriptions, therefore, generally use non-enumerative devices, varying as to how formalized they are, and also as to the basic notions underlying them. In practice a class of texts is usually defined in terms of other auxiliarily defined classes of symbol-strings that occur as segments of the texts to be characterized. For example some classes can be characterized by the set of "trigrams" (i.e. symbol-strings of length three) that occur as segments, or even by the set of "digrams" (see for example Bemer (1959), Floyd (1963)). More often a syntax description must be less systematic, e.g. using such notions as "an *A* consists of any *B* that doesn't end in a *C*, followed by any *D* that doesn't start with an *E* and that contains at least two *F*'s" (see Barnet and Futrelle (1962), Brooker *et al.* (1963)).

Frequently the form of such definitions is limited by using only class union and concatenation, e.g. "an *A* consists either of any *B* followed by any *C* followed by any *D*, or of any *E* followed by any *F*" (see Backus (1959), Irons 1961)). The special case in which the concatenated operands are restricted to class-names, rather than composite expressions, is called "Backus form". This is the form used in the official syntax description of ALGOL 60, with the notation

$$\langle A \rangle ::= \langle B \rangle \langle C \rangle \langle D \rangle \mid \langle E \rangle \langle F \rangle$$

In this case each admissible text owes its admissibility to the fact that it can be parenthetically structured in such a way that each "phrase", i.e. each parenthesized segment, belongs to one of the relevant auxiliary classes. For a particular admissible text there might be several ways of doing this, but if there is just one then the syntax description does more than merely specify which texts are admissible; it also imposes a unique "syntax analysis" on each admissible text. Unless this uniqueness condition holds it is not meaningful to speak of *the* analysis of a text except relative to some rule for selecting one of the possible analyses.* (An analytical algorithm as currently used on computers may be considered as such a rule.)

A syntax description of a language is sometimes used to help explain to a user what texts he can admissibly write; and also to assist in the systematic design of a mechanical processor for the language. If it is formalized it may even be incorporated into the mechanical processor in the sense that there is a systematic way of varying the processor as the syntax description (and language) is varied. Proposals for making use of the various forms of syntax

* In the unrevised ALGOL 60 report [Naur, 1960] there were texts for which the uniqueness condition failed by an oversight (e.g. "**if** *p* **then for** *i* := *k* **do if** *q* **then** *R* **else** *S*"). In the revised version [Naur, 1963] it also fails (consider e.g. "*a* := *b*"), but in a less serious way (for '*a*' and '*b*' can be either arithmetical or Boolean).

description mentioned in the last paragraph have been made in the papers mentioned there. However in the following remarks we shall confine our attention to the most restricted form, i.e. syntax descriptions that yield a unique analysis of each admissible text.

The first thing we observe is that, even given this restriction and excluding trivial sorts of variation (e.g. in phraseology or formalism), there is no unique way of describing the syntax of a language. In fact any class of texts can be described in many different ways, all yielding different analyses of at least some of the texts. Such syntax descriptions vary in practical usefulness, and their usefulness depends partly on how "semantically significant" the analysis is, i.e. on how closely the "phrases" correspond to the intuitively meaningful segments. For example it is possible to describe the syntax of ordinary algebra in such a way that

$$/a) + b) - \sin$$

is a phrase. But it is difficult to think of a use for such a syntactic description (difficult but not impossible—one can imagine a hardware design that makes the recognition of this particular configuration necessary for efficient compiling).*

So, in practice, syntax descriptions do have semantic overtones. To take an extreme case, consider a language whose admissible texts are *all* the strings in its alphabet. There are some trivially simple syntactic descriptions of such a language, e.g.

$$\langle \text{string} \rangle ::= \langle \text{letter} \rangle \mid \langle \text{string} \rangle \langle \text{letter} \rangle$$

but it is likely that a useful syntax description would be more complicated and would correspond more to the "meaning" of the texts. (Far from being perverse, this extreme case has great practical importance. For it is inevitable that every text presented to a machine should have some outcome, albeit inscrutable; and it is now generally recognized as desirable that every text should have a *designed* outcome, rather than one that emerges as an accidental consequence of the design for "admissible texts" in the more restricted sense.)

However, although a useful choice of syntax description does convey more about the language than merely which texts are admissible, it is silent on the

* Another conceivable syntax description for ordinary algebra might include

⟨expression⟩⟨infixed operator⟩⟨expression⟩

as one form of ⟨expression⟩. By failing to specify precedence among arithmetical operators, this will yield multiple analyses of say "$a \times b + c \times d$", including "semantically non-significant" ones in which "$b + c$" is a phrase. This was the method used in the ALGOL 58 report [Perlis and Samelson, 1959], with precedence treated as supplementary to the formal syntax.

subject of what each admissible text, and each constituent phrase, "means". A "complete" description of a computing language is, roughly speaking, one that specifies, for each text, first whether it is admissible, and secondly, if so, what is the outcome of "running" it. (In the special case of a *programming* language this means specifying the outcome for every relevant initial state of the system.)

The following pages are about a particular technique for giving "complete descriptions" in this sense. Of the two main parts the first, about "expressions", comprises sections 3–13. It defines the structure of AEs, and develops their correspondence with certain notations taken from algebra and programming. Observe that, being abstract, AEs have no "syntax" in the sense that ordinary programming languages have. Their "structure" plays the role of syntax, and constitutes what McCarthy calls the "abstract syntax" of the language. The notations used here to demonstrate semantic analysis are to some extent invented to favour the exposition. However they closely resemble certain features of current programming languages, especially ALGOL 60 and CPL, and the analysis can be carried over to the corresponding parts of these languages. Moreover, in the case of ALGOL 60, the development needed to give a complete description of the entire language has been done (see Landin 1965).

The second main part, called "Evaluation" which is discussed in sections 14–18, provides a "complete description" (as opposed to a mere "abstract syntax description") of the language of AEs themselves. It does this in two ways, first by a recursive definition (presented, it so happens, as an AE), and then more mechanistically, in terms of an abstract machine that interprets AEs. As might be expected, just as AEs model some features of current languages, so the abstract machine models some features of current techniques for implementing such languages. Both of these models bring out aspects that probably have fundamental logical significance.

3. APPLICATIVE STRUCTURE

Many symbolic expressions can be characterized by their "operator/operand" structure. For instance

$$a/(2b + 3)$$

can be characterized as the expression whose operator is '/' and whose two operands are respectively '*a*', and the expression whose operator is '+' and whose two operands are respectively the expression whose operator is '×' and whose two operands are respectively '2' and '*b*', and '3'. Operator/operand structure, or "applicative" structure, as it will be called here, can be exhibited more clearly by using a notation in which each operator is written explicitly

and prefixed to its operand(s), and each operand (or operand-list) is enclosed in brackets, e.g.

$$/(a, +(\times(2, b), 3)).$$

This notation is a sort of standard notation in which all the expressions in this paper could (with some loss of legibility) be rendered.

The following remarks about applicative structure will be illustrated by examples in which an expression is written in two ways: on the left in some notation whose applicative structure is being discussed, and on the right in a form that displays the applicative structure more explicitly, e.g.

$$\begin{array}{ll} a/(2b + 3) & /(a, +(\times 2, b), 3)) \\ (a + 3)(b - 4) + & +(\times(+(a, 3), -(b, 4)), \\ (c + 5)(d - 6) & \quad \times(+(c, 5), -(d, 6))). \end{array}$$

In both these examples the right-hand version is in the "standard" notation. In most of the illustrations that follow the right-hand version will not adhere rigorously to the standard notation. The particular point illustrated by each example will be more clearly emphasized if irrelevant features of the left-hand version are carried over in non-standard form. Thus the applicative structure of subscripts is illustrated rigorously by

$$a_j b_{jk} \qquad \times(a(j), b(j, k))$$

but, with the treatment of multiplication already established, we may instead write

$$a_j b_{jk} \qquad a(j)b(j, k).$$

Some familiar expressions have features that offer several alternative applicative structures, with no obvious criterion by which to choose between them. For example

$$3 + 4 + 5 + 6 \qquad \begin{cases} +(+(+(3, 4), 5), 6) \\ +(3, +(4, +(5, 6))) \\ \Sigma'(3, 4, 5, 6) \end{cases}$$

where Σ' is taken to be a function that operates on a list of numbers and produces their sum. Again

$$a^2 \qquad \begin{cases} \uparrow(a, 2) \\ square\ (a) \end{cases}$$

where $\uparrow$ is taken to be exponentiation.

Sometimes the choice may be more material to the meaning. For instance, without background information it is impossible to decide whether or not

$$f(y + 1) + n(y - 1)$$

contains a sub-expression whose operator is multiplication. We are not concerned here with offering specific rules for answering such questions. What interests us is that in many cases such a rule can be considered as a rule about applicative structure.

4. λ-NOTATION

Using Church's λ-notation we can impute applicative structure to some familiar notations that use "private" (or "internal", or "local", or "dummy", or "bound") variables, such as the second and third occurrence of 'x' in

$$\int_0^x x^2\,dx \qquad\qquad \int(0, x, \lambda x.x^2).$$

Here we consider $\int$ as a triadic function operating on two numbers and a function. Its third operand "$\lambda x.x^2$" can be read "the function of x that x^2 is". The λ-notation is a device for indicating explicitly which variable is being used as a dummy. In this example there is no room for doubt; we might say simply "the function x^2". But in general this is not so. For example, consider "$\lambda j.a_{ij}b_{jk}$", i.e. "the function of j that $a_{ij}b_{jk}$ is", as used in

$$\Sigma_{0\leqslant j<n}a_{ij}b_{jk} \qquad\qquad \Sigma''(0, n, \lambda j.a(i,j)b(j,k))$$

where Σ'' is a triadic function that is analogous to $\int$.

A symbol in a mathematical expression may occur in such a way that the value of the expression cannot be determined without supplementary information specifying the numerical value, or functional meaning, or other meaning, of the symbol. Alternatively it may occur in such a way that such supplementary information is unnecessary. In these two situations the occurrence of the symbol is called, respectively, "free" and "bound" in the expression. For example, the occurrence of 'y' in

$$\int_0^y x^2\,dx$$

is free, whereas both occurrences of 'x' are bound. On the other hand, the occurrence of 'x' in

$$x^2$$

is free.

As an alternative turn of phrase to "there is a free (bound) occurrence of x in ...", we sometimes say "x occurs free (bound) in ...". Notice that a symbol may occur both bound and free in the same expression, e.g. x in

$$\int_0^x x^2\,dx$$

If a symbol occurs bound in an expression, the value of the expression is not altered when the symbol is replaced by some other symbol in every bound occurrence of it throughout the expression, e.g.

$$\int_0^x x^2\,dx = \int_0^x y^2\,dy$$

Notice that one free occurrence of 'x' (irrespective of any bound occurrences) is enough to force someone evaluating the expression to rely on supplementary information about the value of 'x'. We say a symbol "*is free*" (as opposed to "occurs free") in an expression, *if and only if* it occurs free in the expression. Hence it is not free if and only if it occurs only bound or not at all. E.g. in

$$\int_a^x \int_b^y (x^2 + y^2 + c^2)\,dy\,dx \qquad\qquad \int(a, x, \lambda y.\int(b, y, \lambda x.x^2 + y^2 + c^2))$$

'a', 'x', 'b', and 'c' *occur* free, while 'x' and 'y' occur bound; on the other hand, 'a', 'x', 'b', and 'c' are free, while all other symbols, including 'y', 'z', and 'e', are not free.

In everyday mathematical symbolism it is not always a clearcut matter to decide whether a particular occurrence of a symbol is free or bound in a particular expression. One of the consequences of establishing applicative structure is to remove this vagueness.

5. AUXILIARY DEFINITIONS

The use of auxiliary definitions to qualify an expression can also be rendered in terms of λ. E.g.

$(u - 1)(u + 2)$ $\qquad\qquad$ $(\lambda u.(u - 1)(u + 2))(7 - 3)$

where $u = 7 - 3$.

Notice that $\lambda u.(u - 1)(u + 2)$ is a function and hence it is appropriate to write this expression with an operand immediately following it, i.e. in a context that is more familiarly occupied by an identifier such as '*sin*' or '*f*', designating a function. Notice also that an expression that denotes a function does not necessarily occur in such a context; witness some previous examples and also

$$\int(0, \pi/2, sin).$$

We shall consistently distinguish between "operators" and "functions" as follows. An *operator* is a sub-expression of a (larger) expression appearing in a context that, when written in standard form, would have an operand (or operand-list) to the right of it. A *function* bears the same relation to an

operator as a number, e.g. the fourth non-negative integer, does to a numerical expression, e.g. $(16 - 7)/(5 - 2)$. The "value" of this expression is a number; similarly we shall speak of the "value" of an expression that can occur as an operator. Just as the value of an expression that occurs as an operand combined with "$\sqrt{}$" must, to make sense, be a number, so the value of an expression that occurs as an operator must be a function. However, any expression that can occur sensibly as an operator can also occur sensibly as an operand.

Applicative structure can be indicated unambiguously by brackets. Legibility is improved by using a variety of bracket shapes. In particular we shall tend to use braces for enclosing (long) operators and square brackets for enclosing operands and operand lists. According to this practice the first example in the present section is better written as

$(u - 1)(u + 2)$ $\qquad\qquad$ $\{\lambda u.(u - 1)(u + 2)\}[7 - 3]$

where $u = 7 - 3$

However, except that we observe correct mating, no formal significance will be attached to differences of bracket shape. That is to say, the rules for making sense of a written expression do not rely on them; no information would be lost by disregarding the differences in a correctly mated expression.

There is another informal device by which we shall bring out the internal grouping of long expressions, namely indentation. For instance the connection between the items of an operand-list, or the two components of an operator/operand combination, will frequently be emphasized by indenting them equally.

The use of several auxiliary definitions, rather than just one, can be rendered in terms of λ. For example if the definitions are mutually independent they can be considered as a "simultaneous" or "parallel" definition of several identifiers, e.g.

$u(u + 1) - v(v + 1)$ $\qquad\qquad$ $u(u + 1) - v(v + 1)$

where $u = 2p + q$ $\qquad\qquad$ **where** $(u, v) = (2p + q, p - 2q)$.

and $v = p - 2q$

So if Church's notation is extended to permit a *list* of identifiers between the 'λ' and the'.', a group of mutually independent auxiliary definitions raises no new issue, e.g.

$u(u + 1) - v(v + 1)$ $\qquad\qquad$ $\{\lambda(u, v).u(u + 1) - v(v + 1)\}$

where $u = 2p + q$ $\qquad\qquad$ $[2p + q, p - 2q]$

and $v = p - 2q$.

If the definitions are inter-dependent the correspondence is more elaborate. Some examples of this situation will be given below.

When we say that the applicative structure of a specific piece of algebraic notation is such-and-such, we are providing unique answers to certain questions about it, such as "What is its operator?" "What are its operands?" Our discussion of *specific* algebraic notations will now be interrupted by a discussion of what precisely these questions are. That is to say, the next section is devoted to explaining what is meant by "applicative structure" rather than to exhibiting the applicative structure of specific notations.

This attempt to characterize applicative structure will use a particular technique called here "structure definitions", and used later in the chapter to characterize other sorts of structure. The next section but one explains this technique. After these two sections, the discussion of the applicative structure of *specific* notations will be resumed.

6. APPLICATIVE EXPRESSIONS

The expressions in this paper are constructed out of certain basic components which are, for our purposes, "atomic"; i.e. their internal structure (if any) does not concern us. They comprise single- and multi-character constants and variables, including decimal numbers. All these will be called *identifiers*. There will be no need for a more precise characterization of identifiers.

By a *λ-expression* we mean, provisionally, an expression characterized by two parts: its *bound variable* part, written between the 'λ' and the '.'; and its λ-body, written after the '.'. (A more precise characterization appears in sections 7 and 8.)

Some of the right-hand versions appearing above contain a λ-expression. Some of those below contain several λ-expressions, sometimes one inside another. This account shows that many expressions can be considered as constructed out of identifiers in three ways: by forming λ-expressions, by forming operator/operand combinations, and by forming lists of expressions. Of these three ways of constructing composite expressions, the first two are called "functional abstraction" and "functional application," respectively. We shall show below that the third way can be considered as a special case of functional application and so, in so far as our discussion refers to functional application, it implicitly refers also to this special case.

We are, therefore, interested in a class of expressions about any one of which it is appropriate to ask the following questions:

Q1. Is it an identifier? If so, what identifier?

Q2. Is it a λ-expression? If so, what identifier or identifiers constitute its

bound variable part and in what arrangement? Also what is the expression constituting its λ-body?

Q3. Is it an operator/operand combination? If so, what is the expression constituting its operator? Also what is the expression constituting its operand?

We call these expressions *applicative expressions* (AEs).

Later the notion of the "value of" (or "meaning of", or "unique thing denoted by") an AE will be given a formal definition that is consistent with our correspondence between AEs and less formal notations. We shall find that, roughly speaking, an AE denotes something as long as we know the value of each identifier that occurs free in it, and provided also that the expression does not associate any argument with a function that is not applicable to it. In particular, for a combination to denote something, its operator must denote a function that is applicable to the value of its operand. On the other hand, any λ-expression denotes a function; roughly speaking, its domain (which might have few members, or even none) contains anything that makes sense when substituted for occurrences of its bound variable throughout its body.

Given a mathematical notation it is a trivial matter to find a correspondence between it and AEs. It is less trivial to discover one in which the intuitive meaning of the notation corresponds to the value of AEs in the sense just given. A correspondence that meets this condition might be called a "semantically acceptable" correspondence. For instance someone might conceivably denote the sum

$$v_r + v_{r+1} + \ldots + v_{r+s-1}$$

of a segment of a vector by

$$v_r^{(s)}.$$

The most direct rendering of this as an AE is something like

$$sum\ (v(r),\ s).$$

However this is not a semantically acceptable correspondence since it wrongly implies dependence on only one element of v, namely v_r. The same criterion prevents λ from being considered as an operator, in our sense of that word; more precisely it rules that

$$\lambda(x,\ x^2 + 1)$$

incorrectly exhibits the applicative structure of '$\lambda x.\, x^2 + 1$'.

We are interested in finding semantically acceptable correspondences that enable a large piece of mathematical symbolism (with supporting narrative) to be rendered by a single AE.

7. STRUCTURE DEFINITIONS

AEs are a particular sort of composite information structure. Lists are another sort of composite information structure. Several others will be used below, and they will be explained in a fairly uniform way, each sort being characterized by a "structure definition". A *structure definition* specifies a class of composite information structures, or *constructed objects* (COs) as they will be called in future. It does this by indicating how many components each member of the class has and what sort of object is appropriate in each position; or, if there are several alternative formats, it gives this information in the case of each alternative. A structure definition also specifies the identifiers that will be used to designate various operations on members of the class, namely some or all defined by

(*a*) *predicates* for testing which of the various alternative formats (if there are alternatives) is possessed by a given CO;

(*b*) *selectors* for selecting the various components of a given CO once its format is known;

(*c*) *constructors* for constructing a CO of given format from given components.

The questions Q1 to Q3 of section 6 comprise the main part of the structure definition for AEs. What they do not convey is the particular identifiers to be used to designate the predicates, selectors and constructors. Future structure definitions in this paper will be laid out in roughly the following way.

An AE is either

an *identifier*,
or a *λ-expression* (*λexp*) and has a *bound variable* (*bv*) which is an identifier
or identifier-list,
and a *λ-body* (*body*) which is an AE,
or a *combination* and has an *operator* (*rator*) which is an AE,
and an *operand* (*rand*) which is an AE.

This is intended to indicate that '*identifier*', '*λ-expression*' and '*combination*' (and also the abbreviations written after them if any) designate the predicates, and '*bv*', '*body*', '*rator*', '*rand*' (mentioning here the abbreviated forms) designate the selectors. We consider a predicate to be a function whose result for any suitable argument is a "truth-value", i.e. either **true** or **false.** For instance, if X is a λ-expression, then the predicate *λexp* applied to X yields **true**, whereas *identifier* yields **false**; i.e. we have for example

$$\lambda exp\ X = \textbf{true}$$

$$identifier\ X = \textbf{false}.$$

(It will be observed that, by considering predicates as functions, we are led into a slight conflict with the normal use of the word "apply". For instance in normal use it might be said that the predicate *even* "applies to" the number six, and "does not apply to" the number seven. We must here avoid this turn of phrase and say instead that *even* "holds for", or "yields **true** when applied to", six; and "does not hold for", or "yields **false** when applied to", seven.)

The constructors will not usually be named explicitly. Instead we shall use obviously suggestive identifiers such as '*constructλexp*'. E.g. the following equations hold:

$$\begin{aligned} &\textit{identifier}\ (\textit{constructλexp}\ (J, X)) = \textbf{false} \\ &\textit{λexp}\ (\textit{constructλexp}\ (J, X)) = \textbf{true} \\ &\textit{bv}\ (\textit{constructλexp}\ (J, X)) = J \\ &\textit{constructλexp}\ (\textit{bv}\ X, \textit{body}\ X) = X \end{aligned}$$

and many others. (More precisely, each of these equations holds provided J and X are such as to make both sides meaningful. Thus the first three hold provided J is an identifier or list of identifiers and X is an AE. Again, the last holds provided X is a λ-expression.)

A structure definition can also be written more formally, as a definition with a left-hand side and a right-hand side. The left-hand side consists of all the identifiers to which the structure definition gives meaning. The right-hand side is an AE containing references to the component-classes involved. For example, assuming that identifiers are character-strings, the formalized structure definition of AEs will refer to the class of character-strings. Also any formalized structure definition will contain references to one or more of a small number of functions concerned with classes of COs. However, in this paper we shall not formalize the notion of structure definitions, and shall write any we need in the style illustrated above.

Superficially, structure definitions resemble syntax descriptions in Backus form. However a structure definition defines abstract information structures, not written representations of them. In particular it raises no question analogous to the uniqueness question in syntax descriptions. Put another way, any result of a constructor is guaranteed to be amenable to the appropriate selectors (i.e. guaranteed to yield precisely one result for each selector).

8. FUNCTION DEFINITIONS

In ordinary use, definitions frequently give a functional, rather than numerical, meaning to the definiendum by using a dummy argument variable. This can be rendered as an explicit definition with a λ-expression for its right hand side, e.g.

$$f(y) = y(y + 1) \qquad f = \lambda y.y(y + 1).$$

So an expression using an auxiliary function definition can be rendered by using two λ-expressions, one for its operator and one for its operand, e.g.

$$f(3) + f(4) \qquad \{\lambda f.f(3) + f(4)\}[\lambda y.y(y + 1)].$$
$$\textbf{where } f(y) = y(y + 1)$$

A group of auxiliary definitions may include both numerical and functional definitions, e.g.

$$f(a + b, a - b) + \qquad \{\lambda(a, b, f).f(a + b, a - b) +$$
$$f(a - b, a + b) \qquad f(a - b, a + b)\}$$
$$\textbf{where } a = 33 \qquad [33, \textbf{44}, \lambda(u, v).uv(u + v)].$$
$$\textbf{and } b = \textbf{44}$$
$$\textbf{and } f(u, v) = uv(u + v)$$

When a λ-expression is written as a sub-expression of a larger expression, the question may arise: how far to the right does its body extend? This question can always be evaded by using enough brackets, e.g.

$$(\lambda(u, v).(uv(u + v))).$$

However, to economize in brackets, we adopt the convention that it extends as far as is compatible with the brackets, except that it is stopped by a comma. Another way of saying this is that the "binding power" of the '.' is less than that of functional application, multiplication and all the written operators such as '+', '/', etc., but exceeds that of the comma. For example

$$f(g(a)) + g(f(b)) \qquad \{\lambda(f, g).f(g(a)) + g(f(b))\}$$
$$\textbf{where } f(z) = z^2 + 1 \qquad [\lambda z.z^2 + 1, \lambda z.z^2 - 1].$$
$$\textbf{and } g(z) = z^2 - 1$$

An identifier may occur in the *bound variable* part of a λ-expression (either constituting the entire bound variable, or as one item of it). Apart from this, every written occurrence of an AE is in one of the four sorts of context

(*a*) It is the λ-body of some λ-expression.
(*b*) It is the operator of some combination.
(*c*) It is the operand of some combination.
(*d*) It is a "complete" AE occurring in a context of English narrative, or other non-AE.

Each of the three formats of AE can appropriately appear in any of the four sorts of contexts. We have already seen that λ-expressions, like identifiers, can appropriately occur both as operators and as operands. Below we shall find combinations appearing as operators, and λ-expressions appearing as λ-bodies. These last two possibilities are both associated with the

possibility that a function might produce a function as its result. Together with more obviously acceptable possibilities, they almost complete the full range of ways in which a particular sort of AE can appear in a particular sort of context. (The one remaining case is that of an identifier occurring as a λ-body, which occurs in a later example.)

The right-hand side of an auxiliary definition might itself be qualified by an auxiliary definition, e.g.

$$\begin{array}{ll} u/(u+5) & \{\lambda u.u/(u+5)\} \\ \textbf{where } u = a(a+1) & \quad [\{\lambda a.a(a+1)\}[7-3]]. \\ \quad \textbf{where } a = 7-3 & \end{array}$$

In particular this might happen with an auxiliary definition of a function e.g.

$$\begin{array}{ll} f(3)+f(4) & \{\lambda f.f(3)+f(4)\} \\ \textbf{where } f(x) = ax(a+x) & \quad [\{\lambda a.\lambda x.ax(a+x)\}[7-3]]. \\ \quad \textbf{where } a = 7-3 & \end{array}$$

This last example contains a λ-expression whose body is another λ-expression. Notice that such a λ-expression describes a "function-producing" function and hence can meaningfully give rise to a combination whose operator is a combination, e.g.

$$\{\{\lambda a.\lambda x.ax(a+x)\}[7-3]\}[3].$$

We shall slightly abbreviate such expressions by omitting brackets round an operator that is a combination, i.e.

$$\{\lambda a.\lambda x.ax(a+x)\}[7-3][3].$$

This amounts to an "association to the left" rule. We also abbreviate by omitting brackets round a single identifier,

$$\{\lambda a.\lambda x.ax(a+x)\}[7-3]3.$$

Similarly we may write '$fa+f3+Dfb$' for '$f(a)+f(3)+\{D(f)\}[b]$', and rely on context to distinguish between 'f applied to a' and 'f times a' (as indeed is customary when writing '$f(a+1)$').

Since we shall use multicharacter identifiers (excluding spaces), this abbreviation means that the reader will sometimes be obliged to use his intelligence, together with the context, to decide whether, e.g.

$$prefix\ x\ nullist$$

is to be read as

$$\{prefix[x]\}[nullist]$$

or

$$\{prefi[x][x]\}[nul[list]]$$

or many other conceivable alternatives. Generally we shall use spaces wherever they are helpful without appearing ungainly.

We now turn to three forms of expression that play an important role in programming languages, namely lists (in particular argument-lists), conditional expressions and recursive definitions. The next three sections are devoted to showing how these can be rendered as operator/operand combinations using certain basic functions.

9. LISTS

In an earlier section we gave a structure definition for AEs that made no explicit provision for *lists* of operands. Our illustrations have begged this issue by using dyadic and triadic functions. It will turn out below that discussion of the evaluation of AEs can be simplified if we can avoid classifying operands into "single operands" and "operand-lists", and avoid classifying functions into those that take one argument and those that take several. We now show how this is done.

Lists can be characterized by a structure definition:

A *list* is either *null*
or else has a *head* (*h*)
and a *tail* (*t*) which is a list.

(For conciseness we use these notations instead of the *Hd* and *Tl* of Chapter 2.)

A null-list has length zero. A non-null list has length one or more; if its items are $a_1, a_2, \ldots, a_k$, ($k \geqslant 1$), then its head is a_1 and its tail is the (null or non-null) list whose $k - 1$ items are $a_2, \ldots, a_{k-1}$ and a_k. So we let

$$1st = h$$
$$2nd\ L = h(tL)$$
$$3rd\ L = h(t(tL)), \text{ etc.}$$

defining the functions *1st*, *2nd*, etc., in terms of the selectors *h* and *t*. So the "items" of a list are the things that result from applying *1st*, *2nd*, etc., to it.

On the lines mentioned earlier, the two identifiers *constructnullist* and *constructlist* designate constructors for lists, taking respectively zero and two arguments. So we have, for example,

$$null(constructnullist(\,)) = \textbf{true}$$
$$null(constructlist(x, L)) = \textbf{false}$$
$$h(constructlist(x, L)) = x$$
$$constructlist(hL, tL) = L$$

and several more such equations hold.

We shall not distinguish between lists in this sense and the argument lists of dyadic and triadic functions. That is to say, we consider a triadic function to be a function whose arguments are limited to lists of length three. So an operator denoting a triadic function is not necessarily prefixed to an operand-list of three items; e.g. if L is a list of two numbers, the expression

$$\{\lambda(x, y, z).x + y + z\}[constructlist(3, L)].$$

is acceptable.

We use *nullist* to designate a list of length zero, and consider an empty bracket pair as merely an alternative way of writing *nullist*. Also we consider commas as merely an alternative way of writing a particular sort of combination, which we now explain.

Associated with any object x there is a function that transforms any given list into a list of length one more, by adding x at the beginning of it. We denote this function by

$$prefix(x).$$

So if L is a list whose k items are $a_1, a_2, \ldots, a_k$, then

$$\{prefix(x)\}(L)$$

denotes a list whose $k + 1$ items are $x, a_1, \ldots, a_k$. The function *prefix* is function-producing and so gives rise to combinations whose operators are combinations. It can be defined in terms of *constructlist* by

$$prefix(x) = \lambda L.constructlist(x, L).$$

By a natural extension of the notation for function definitions this can also be written

$$prefix(x)(L) = constructlist(x, L).$$

The applicative structure we are now imposing on the operand-lists of length two or more, and of length zero is illustrated by the examples

$f(a, b, c)$	$f(prefix\ a(prefix\ b(prefix\ c\ nullist)))$
$a + b$	$+(prefix\ a(prefix\ b(nullist)))$
$constructnullist(\ )$	$constructnullist(nullist).$

Notice that while it is meaningful to ask whether a function is dyadic (i.e. has arguments restricted to lists of length two), there is no significance to asking whether a function is monadic since any function may be denoted in combination with a single operand rather than a list of operand expressions.

For the rare cases in which we wish to refer to a list with just one item, we use the function defined by

$$unitlist(x) = prefix\ x\ nullist.$$

We shall use for '*prefix x L*' the abbreviation

$$x{:}L.$$

So, e.g.

$$x, y, z = x{:}(y, z) = x{:}(y{:}\mathit{unitlist}\ z) = x{:}(y{:}(z{:}(\,)))$$

We shall treat ':' as more "binding" than ',', e.g.

$$2nd(2nd(L, x{:}M, N)) = 1st\ M.$$

The last example refers to a list whose items include a list. We admit this possibility and write, e.g.

$$(a, b), (c,(\,), e), \mathit{unitlist}\, f.$$

In what follows, a list whose items include lists (i.e. a list which has items that are amenable to *null*, *1st*, *2nd*, etc.) will be called a "list-structure".

10. CONDITIONAL EXPRESSIONS

We now show how AEs provide a match for conditional expressions, e.g. for

$$\textbf{if}\ a < b\ \textbf{then}\ a^7\ \textbf{else}\ b^7. \tag{10.1}$$

This expression somewhat resembles

$$i^{\text{th}}(a^7, b^7)$$

where i is a computed index number, used to select an item from a list which is not referred to elsewhere. So, we consider '*if*' to be an identifier designating a function-producing function such that

$$\mathit{if}\,(\textbf{true}) = 1st$$
$$\mathit{if}\,(\textbf{false}) = 2nd.$$

Then (10.1) is equivalent to the AE

$$\mathit{if}\,(a < b)(a^7, b^7).$$

This rendering is not, however, adequate. For it would match

$$\textbf{if}\ a = 0\ \textbf{then}\ 1\ \textbf{else}\ 1/a \tag{10.2}$$

by

$$\mathit{if}\,(a = 0)(1,\ 1/a). \tag{10.3}$$

But the value of this expression, i.e. to be more explicit, of

$$\mathit{if}\,(a = 0)(\mathit{prefix}\ 1(\mathit{prefix}(1/a)(\,)))$$

depends on the value of the sub-expression '$1/a$', and hence only exists if $1/a$ exists. So (10.3) is not an acceptable rendering of (10.2) if a is zero and

division by zero is undefined. More generally, this method of rendering conditional expressions as AEs does not meet our criterion of semantic acceptability unless the domain of every function is artificially extended to contain any argument that might conceivably arise on either "branch" of a conditional expression. We now present another method that avoids any such commitment.

Consider instead the alternative

$$if\,(a = 0)(\lambda x.1,\ \lambda x.1/a)(3) \tag{10.4}$$

where 'x' is an arbitrarily chosen variable and '3' is an arbitrarily chosen operand. Unlike (10.3), (10.4) has a value even if $a = 0$; for, '$\lambda x.1/a$' denotes a function even if $a = 0$ (albeit with null domain—this is in accordance with our view of the "value" of an expression, as introduced informally in section 6 above and formalized in section 14 below). So (10.4) is precisely equivalent to (10.2) in the sense that either they are equivalent or they are both without value.

The arbitrary 'x' and '3' in (10.4) can be obviated.* For the *bv* of a λ-expression can be a list of identifiers, and in particular a list whose length is zero. Such a λ-expression is applicable to an argument list of the same length. This suggests that all conditional expressions can be rendered in a uniform way in the form

$$\textbf{if } a < b \textbf{ then } a^7 \textbf{ else } b^7 \quad if\,(a < b)(\lambda(\,).a^7,\ \lambda(\,).b^7)(\,)$$

$$\textbf{if } a = 0 \textbf{ then } 1 \textbf{ else } 1/a \quad if\,(a = 0)(\lambda(\,).1,\ \lambda(\,).1/a)(\,).$$

11. RECURSIVE DEFINITIONS

The use of self-referential, or "circular", or what have come to be called in the computer world, "recursive" definitions can also be rendered in operator/operand terms. By a circular definition we mean an implicit definition having the form

$$x = \ldots x \ldots x \ldots x \ldots$$

i.e. a definition of x in which x is referred to one or more times in the definiens. For example suppose 'M' designates a list-structure, then

$$(a,\ M,\ (b,\ c))$$

denotes a list-structure whose second item is the list-structure M. The equation

$$L = (a,\ L,\ (b,\ c)) \tag{11.1}$$

* The device given here was suggested by W. H. Burge.

is satisfied by the "infinite" list-structure containing three items, of which the first is a, the third is (b, c) and the second is the infinite list-structure whose first item is a, and whose third item is (b, c) and whose second, . . . and so on.

So equation (11.1) may be considered as a circular definition that associates this "infinite" list-structure with the identifier 'L'.

Again

$$f(n) = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } nf(n - 1)$$

i.e.

$$f = \lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } nf(n - 1)$$

may be considered as a circular definition of the factorial function. (In this brief discussion the important question of whether each circular definition characterizes a *unique* object will be skipped.)

Making use of λ, any circular definition can be rearranged so that there is just *one* self-referential occurrence, and moreover so that the single occurrence constitutes the operand of the definiens, e.g.

$$L = (a, L, (b, c)) \qquad L = \{\lambda L'.(a, L', (b, c))\}L \tag{11.2}$$

$$f(n) = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } nf(n - 1) \qquad f = \{\lambda f'.\lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } nf'(n - 1)\}f. \tag{11.3}$$

Notice that, had we used 'L' and 'f' instead of 'L'' and 'f'' they would still have been bound and so would not have constituted self-referential occurrences.

A circular definition of the form

$$x = Fx$$

(such as (11.2) or (11.3)) characterizes an object as being invariant when transformed by the function F, i.e. as the "fixed-point" of F. If we use 'Y' to designate the function of finding the fixed-point of a given function, such a circular definition can be rearranged, so that it is formally no longer circular, in the form

$$x = YF.$$

Thus (11.2) and (11.3) become

$$L = (a, L, (b, c)) \qquad L = Y\lambda L.(a, L, (b, c))$$

$$f(n) = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } nf(n - 1) \qquad f = Y\lambda f.\lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } nf(n - 1).$$

Notice that, according to the treatment of conditional expressions in section 10, the existence of $f(0)$ does not involve the existence of $f(-1)$. Notice also that Y may produce a function, and hence gives rise to combinations whose operators are combinations, e.g.

$$\{Y\lambda f.\lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } nf(n - 1)\}6$$

is a meaningful combination. In fact its value is 720.

This device can also be used for a group of "jointly circular" or "simultaneously recursive" definitions, e.g.

$$\begin{aligned} fx &= F[f, g, x] \qquad (f, g) = Y\lambda(f, g).(\lambda x.F[f, g, x], \\ \textbf{and } gx &= G[f, g, x] \qquad\qquad \lambda x.G[f, g, x]). \end{aligned} \tag{11.4}$$

So the fixed-point of a function might be a list of functions. This gives rise to the possibility that a dyadic function might appear with what looks like one, rather than two, arguments, e.g. when the jointly circular functions (11.4) appear in an auxiliary definition in the form

$$\begin{aligned} &f(ga) + g(fb) \qquad\qquad \{\lambda(f, g).f(ga) + g(fb)\} \\ &\textbf{where } fx = F(f, g, x) \qquad [Y\lambda(f, g).(\lambda x.F(f, g, x), \lambda x.G(f, g, x))]. \\ &\quad \textbf{and } gx = G(f, g, x) \end{aligned}$$

Notice that the circularity is explicitly indicated in the right-hand version, whereas the left-hand version is only recognizable as circular by virtue of our comments about it or by common sense. In the next section we shall extend our hitherto informal use of **where** so as to provide a match for any use of λ.

12. THE DIFFERENCE BETWEEN STRUCTURE AND WRITTEN REPRESENTATION

Our notation for AEs is deliberately loose. There are many ways in which we can write the same AE, differing in layout, use of brackets and use of infixed as opposed to prefixed operators. However they are all written representations of the *same* AE, in the sense that the information elicited by the questions Q1, Q2 and Q3 of section 6 are the same in each case.* This is the essential information that characterizes the AE. We call this information the "structure" of the AE. Our laxity with written representations is based on the knowledge that any expressions we write could, at the cost of legibility, have been written in standard form, with exclusively prefixed operators and every bracket in place.

One of the syntactic liberties that we shall take is to use **where** instead of λ. More precisely, we shall use an expression of the form

$$L \textbf{ where } X = M$$

as a "syntactic variant" of

$$\{\lambda X.L\}[M]$$

* Thus for example, "$\lambda x.x$" and "$\lambda y.y$" are different AEs, as can be seen by observing that

$$bv(cons\lambda exp('x', 'x')) \neq bv(cons\lambda exp('y', 'y'))$$

even in cases that go rather further than the familiar use of **where**. e.g.

$$\begin{array}{ll} n^2 + 3n + 2 & \{\lambda n.n^2 + 3n + 2\}[n + 1] \\ \textbf{where } n = n + 1 & \\ xy(x + y) & \{\lambda y.\{\lambda x.xy(x + y)\} \\ \textbf{where } x = a^2 + a\sqrt{y} & \qquad [a^2 + a\sqrt{y}]\} \\ \textbf{where } y = a^2 + b^2 & [a^2 + b^2]. \end{array}$$

We use indentation to indicate that the **where** qualifies a sub-expression, e.g., 'y' occurs both bound and free in each of the examples

$$\begin{array}{ll} xy(x + y) & \{\lambda(x, y).xy(x + y)\} \\ \textbf{where } x = a^2 + a\sqrt{y} & \quad [a^2 + a\sqrt{y}, a^2 + b^2] \\ \quad \textbf{and } y = a^2 + b^2 & \\ xy(x + y) & \{\lambda x.xy(x + y)\} \\ \textbf{where } x = a^2 + a\sqrt{y} & \quad [\{\lambda y.a^2 + a\sqrt{y}\}[a^2 + b^2]]. \\ \qquad \textbf{where } y = a^2 + b^2 & \end{array}$$

The **where** notation can be extended to allow for circular definitions and jointly circular definitions, thus formalizing a feature of auxiliary definitions that has previously required verbal comment. An occurrence of 'Y' is indicated by the word **recursive** or, more shortly, **rec**, for example in

$$\begin{array}{ll} f(7 - 3) & \{\lambda f.f(7 - 3)\} \\ \textbf{where rec } f(n) = & \quad [Y\lambda f.\lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } nf(n - 1)]. \\ \quad \textbf{if } n = 0 \textbf{ then } 1 & \\ \qquad \textbf{else } nf(n - 1) & \\ f(ga) + g(fb) & \{\lambda(f, g).f(ga) + g(fb)\} \\ \textbf{where rec } fx = F(f, g, x) & \quad [Y\lambda(f, g).(\lambda x.F(f, g, x), \lambda x.G(f, g, x))]. \\ \quad \textbf{and } gx = G(f, g, x) & \end{array}$$

It will be observed that our discussion of applicative structure has doubled back on itself. We started by remarking the possibility of analysing certain more or less familiar notations in terms of functional application and functional abstraction. We are now remarking the possibility of looking upon these notations as "really" AEs, written with syntactic variations that make them more palatable. Clearly, once a semantically acceptable correspondence between AEs and some other notation has been established, it can be looked at in either way.

The above explanation of **where** and AEs leaves some details unsettled, but should be enough to make the use of **where** in what follows (*a*) comprehensible

and (*b*) plausibly a mere "syntactic sugaring" of AEs. Further discussion of **where**, or of other sorts of syntactic sugar, is outside the scope of this chapter.

Another example of alternative notations concerns conditional expressions. Interchangeably with the **if** . . . **then** . . . notation we use the $\rightarrow$ notation as illustrated by the two examples

$$\begin{array}{ll} p \rightarrow a & \textbf{if } p \textbf{ then } a \\ \textbf{else} \rightarrow b & \quad \textbf{else } b. \\ p \rightarrow a & p \rightarrow a \\ q \rightarrow b & \textbf{else} \rightarrow (q \rightarrow b \\ \textbf{else} \rightarrow c & \quad \textbf{else} \rightarrow c). \end{array}$$

Any particular set of rules about representing AEs by written text (the correspondence with **where** is one such set of rules) has two aspects:

(*a*) a rule for deriving the structure of an AE, given a text that represents it,

(*b*) a rule for deriving a text that represents an AE, given its structure.

The formalization of these rules, and in particular their formalization as AEs, is another topic that is outside the scope of this chapter.

13. THE POWER OF APPLICATIVE EXPRESSIONS

We have described how certain expressions can be considered as being constructed from "known" identifiers, or "constants", by means of functional application and functional abstraction. We might look at the situation another way round and consider how many expressions can be constructed, starting with a given selection of constants, using these same means of construction. More precisely we might compare working within such constraints to working within some other set of constraints, e.g. some algebraic programming language or machine code, or system of formal logic. It transpires that the seven objects, *null*, *h*, *t*, *nullist*, *prefix*, *if* and *Y* provide a basis for describing a wide range of things.

Roughly speaking, when taken together with functional application and functional abstraction, they perform the "house-keeping" or "red-tape" roles that are performed by sequencing, indices and copying in a conventional programming language, and by narrative in informal mathematics. For example

(1) With a few basic numbers and numerical functions they are sufficient to describe the numbers and functions of recursive number theory. So they are in some sense "as powerful as" other current symbolisms of mathematical logic and computer programming. The question whether this sense has much practical significance is one that will not be discussed here.

(2) With a few basic symbols, and functions associated with classes of symbol-strings, they are sufficient to describe syntax (of, say, ALGOL 60, or for AEs themselves), from the point of view both of synthesizing and of analysing.
(3) With a few basic classes, and functions associated with classes of composite information structures, they are sufficient to formalize "structure definitions", as introduced above (for example the structure definitions of AEs themselves).
(4) With a few structure definitions they are sufficient to characterize formally the "value" of an AE, and to describe a mechanical process for "producing" it. This is the use to which AEs will be put in the rest of this chapter.

A discussion of the relative convenience of various notations in the fields mentioned here is outside our present scope.

EVALUATION

14. THE VALUE OF AN APPLICATIVE EXPRESSION

Every AE in previous examples, including every sub-expression of every AE, has a "value", which is either a number, or a function, or a list of numbers, or a list of functions, etc. More precisely, an AE X has a value (or rather *might* have a value) relative to some back-ground information that provides a value for each identifier that is free in X. This background information will be called the *environment* relative to which evaluation is conducted. It will be considered as a function that associates with each of certain identifiers, either a number, or a list, or a function, etc. Each identifier to which an environment E gives a value is called a *constant* of E and each object "named", or "designated", by a constant of E (possibly by several) is called a *primitive* of E. So E is a function whose domain and range comprise respectively its constants and its primitives.

If we let

$$val(E)(X)$$

denote the value of X relative to E (or *in* E for short), the function that '*val*' designates can be specified by means of three rules, R1, R2 and R3. These correspond to the three questions, Q1, Q2 and Q3 that were introduced in section 6 to elucidate the structure of AEs.

R1. If X is an identifier, $valEX$ is EX;
(R2. appears below);
R3. If X is a combination, $valEX$ can be found by first subjecting both its operator and operand to $valE$, and then applying the result of the former to the result of the latter.

The rules R1 and R3 are enough to specify *valEX* provided that *X* contains no λ-expressions. For example consider an environment in which the identifier *k* is associated with the number 7 and the identifier *p* with the truthvalue **false**, and other identifiers have their expected meanings. Then R1 and R3 suffice to fix the value of, say,

$$if((2^{19} < 3^{12}) \vee p)(sin, cos)(\pi/k).$$

This example illustrates the need for evaluating the operator of a combination as well as its operand.

The value of a λ-expression is a function. We characterize this function by specifying what result, if any, it produces for an arbitrary argument. Specifically its result for any given argument can be found by evaluating the body of the λ-expression in a newly derived environment. For example suppose *E* is the environment postulated above, and *X* is the λ-expression '$\lambda r.k^2 + r^2$'. Then its value in *E* is a function whose result for any given argument, say 13, can be found by evaluating '$k^2 + r^2$' in a new environment *E'*, derived from *E*. To be precise, *E'* agrees with *E* except that it gives the value 13 to the identifier *r*. This gives R2. If *X* is a λ-expression, *valEX* is a function whose result, if any, for argument *x* is the value, if any, of *bodyX* in a new environment. This environment consists of *E*, modified by pairing the identifier(s) in *bvX* with corresponding components of the given argument *x* (and using the new value for preference if any variable in *bvX* coincides with a constant of *E*).

We denote this derived environment by

$$derive(assoc(bvX, x))E.$$

We shall describe in subsequent sections a mechanical process for obtaining the value, if it exists, of any given AE relative to any given environment. This process can be implemented with pencil and paper, or (as we shall briefly sketch) with a digital computer. The rules R1, R2 and R3 provide a criterion for deciding whether or not the outcome of this process is in fact the value as we understand it.

The three rules can be formalized as a definition of *val* in the form

$$\begin{aligned}
&\textbf{recursive } valEX = identifierX \rightarrow EX \\
&\qquad \lambda expX \rightarrow f \\
&\qquad\quad \textbf{where } fx = val(derive(assoc(bvX, x))E) \\
&\qquad\qquad\qquad\qquad (bodyX) \\
&\qquad \textbf{else} \rightarrow \{valE(ratorX)\}[valE(randX)].
\end{aligned} \tag{14.1}$$

For example, suppose *thrice* is the function-producing function defined by

$$thrice(f)(x) = f(f(f(x))).$$

Then it follows from our definition of *val* that the values of the AEs

square 5
thrice square 5
thrice square (*thrice square* 5)
thrice (*thrice square*) 5
thrice thrice square 5

are respectively 5^2, 5^{2^3}, 5^{2^6}, 5^{2^9} and $5^{2^{27}}$. The reader may be better equipped to check this assertion when he has read the next section, which describes an orderly way of evaluating AEs.

The set of objects that can be denoted by an AE relative to an environment E, is the range of the function *valE*. It contains all the primitives of E, and everything produced by such an object, and every function that can be denoted by a λ-expression.

15. MECHANICAL EVALUATION

In order to mechanize rule (14.1), we represent an environment by a list of "environmental-layers"; each layer consists of

its *index*, which is a list-structure of identifiers,

and its *contents*, which is an object congruent to the index.

There is a function designated by "*location*" such that if E^* is this structure and X is an identifier then

$$locationE^*X$$

denotes the selector that selects the value of X from E^*. So if E^* represents the environment E, then

$$valEX = locationE^*XE^*.$$

The two occurrences of 'E^*' here correspond to the two references to a book when one looks up something in the index, and then looks up the thing itself.

We represent the value of a λ-expression by a bundle of information called a "closure", comprising the λ-expression and the environment relative to which it was evaluated. We must therefore arrange that such a bundle is correctly interpreted whenever it has to be applied to some argument. More precisely

a *closure* has

an *environmental part* which is a list whose two items are:

(1) an environment, or more precisely, a CO representing an environment,
(2) an identifier or list of identifiers,

and a *control part* which consists of a list whose sole item is an AE.

The value relative to E^* of a λ-expression X is represented by the closure denoted by

$$constructclosure((E, bvX), unitlist(bodyX)). \quad (15.1)$$

This particular arrangement of the information in a closure has been chosen for our later convenience.

In what follows we say simply "environment" instead of "CO representing an environment". Also, we omit the asterisk from 'E^*'.

We now describe a "mechanization" of evaluation in the following sense. We define a class of COs, called "states", constructed out of AEs and their values; and we define a "transition" rule whose successive application starting at a "state" that contains an environment E and an AE X (in a certain arrangement), leads eventually to a "state" that contains (in a certain position) either *valEX* or a closure representing *valEX*. (We use the phrase "result of evaluation" to cover both objects and closures. We suppose that the identifier '*closure*' designates a predicate that detects whether or not a given result of evaluation is a closure.)

A *state* consists of a *stack*, which is a list, each of whose items is an intermediate result of evaluation, awaiting subsequent use;

and an *environment*, which is a list-structure made up of name/value pairs;

and a *control*, which is a list, each of whose items is either an AE awaiting evaluation, or a special object designated by '*ap*', distinct from all AEs;

and a *dump*, which is a complete state, i.e. comprising four components as listed here.

We denote a state by

$$(S, E, C, D).$$

The environment-part (both of states and of closures) would be unnecessary if λ-expressions containing free variables were prohibited. Also the dump would be unnecessary if all λ-expressions were prohibited.

Each step of evaluation is completely determined by the current state (S, E, C, D) in the following way.

1. If C is null, suppose the current dump D is

$$(S', E', C', D').$$

Then the current state is replaced by the state denoted by

$$(hS:S', E', C', D').$$

2. If C is not null, then hC is inspected, and

(2a) If hC is an identifier X (whose value relative to E occupies the position *locationEX* in E), then S is replaced by

$$locationEXE:S$$

and C is replaced by tC. We describe this step by the statement "Scanning X causes *locationEXE* to be loaded".

(2b) If hC is a λ-expression X, scanning it causes the closure derived from E and X (as indicated in (15.1)) to be loaded on to the stack.

(2c) If hC is *ap*, scanning it changes S as follows: hS is inspected and

(2c1) If hS is a closure, derived from E' and X', then:

S is replaced by the nullist,
E is replaced by $prefix(constructlayer(bvX', 2nd\ S))E'$,
C is replaced by $unitlist(bodyX')$,
D is replaced by $(t(tS), E, tC, D)$.

(2c2) If hS is not a closure, then scanning *ap* causes S to be replaced by

$$((1st\ S)(2nd\ S):t(tS)).$$

(2d) If hC is a combination X, C is replaced by

$$randX:(ratorX:(ap:tC)).$$

The successive states that these rules generate during the evaluation of "*thrice thrice square* 5" are shown in Appendix 1.

Formally this transformation of one state into another is given by

$Transform(S,E,C,D) =$
  $nullC \rightarrow [hS:S', E', C', D']$
    where $S', E', C', D' = D$
  else $\rightarrow$
    $identifier\ X \rightarrow [locationEXE:S, E, tC, D]$
    $\lambda exp\ X \rightarrow$
      $[constructclosure((E, bvX), unitlist(bodyX)):S, E, tC, D]$
    $X = ap \rightarrow closure(hS) \rightarrow$ (15.2)
        $[(\ ), prefix(constructlayer(J,2ndS))E',$
          $C',$
          $(t(tS), E, tC, D)]$
        where $E', J = environmentpart(hS)$
          and $C' = controlpart(hS)$
        else $\rightarrow [(1stS)(2ndS):t(tS), E, tC, D]$
    else $\rightarrow [S, E, randX:(ratorX:(ap:tC)), D]$
    where $X = hC$

We assume here that an AE composed of a single identifier is the same object as the identifier itself. This suggests a more general assumption that whenever one of the alternative formats of a structure definition has just one component, the corresponding selector and constructor are both merely the identity function. We also assume that a state is identical to a list of its four components. This suggests a more general assumption that whenever a structure definition allows just one format, the constructor is the identity function. Without these assumptions our definition would be a bit more elaborate. A formal account of structure definitions would lead to a more careful discussion of these points.

Notice that, whereas the formula (14.1) described a rule for deriving from an AE its value, the formula (15.2) describes a rule for advancing a certain information-structure through one step. If X is an AE, and E is an environment such that *valEX* is defined, then starting at any state of the form

$$S,\ E,\ X{:}C,\ D$$

and repeatedly applying this transformation, we shall eventually reach the state denoted by

$$valEX{:}S,\ E,\ C,\ D.$$

That is to say, at some later time X will have been scanned and its value relative to the current environment will have been loaded (on to the stack). In particular, if S and C are both null, i.e. if the initial state is

$$(\ ),\ E,\ unitlist X,\ (S',\ E',\ C',\ D')$$

there will be a subsequent state

$$unitlist(valEX),\ E,\ (\ ),\ (S',\ E',\ C',\ D')$$

which will be immediately succeeded by the state denoted by

$$valEX{:}S',\ E',\ C',\ D'.$$

These assertions can be verified by performing the appropriate substitutions in the definition (15.2) of *Transform*.

16. BASIC FUNCTIONS

By a "basic" function of E we mean a function other than a closure, that can arise as a result of evaluation. At the most the basic functions comprise

(1) primitive functions;
(2) any functions produced by basic functions.

For, any result of a closure must also be a result of a primitive (or be a result of a result of a primitive, or, etc.).

However, this may be an over-estimate of the number of basic functions, for it is clearly possible that a primitive might be a closure. For instance the evaluation of

$$\{\lambda f.f3 + f4\}[\lambda x.x^2 + 1]$$

relative to E involves evaluating

$$f3 + f4$$

relative to an environment in which 'f' names the closure that we may roughly denote by

$$\mathit{constructclosure}((E, \text{'}x\text{'}), \mathit{unitlist}(\text{'}x^2 + 1\text{'})).$$

Of the six sorts of step described in section 15, namely (1), (2*a*), (2*b*), (2*c*1), (2*c*2) and (2*d*), all except (2*c*2) are mere rearrangements. (2*c*2) arises whenever *ap* finds that the head of the stack is a basic function.

17. OTHER WAYS OF MECHANIZING EVALUATION

It should be observed that this is only one of many ways of mechanizing the evaluation of AEs, all producing the value, as specified in section 15. For instance it is not essential that the operand of a combination be evaluated before its operator. The operand might be evaluated after the operator; it might even be evaluated piecemeal when and if it is required during the application of the value of the operator. Again, the evaluation of a λ-expression might be accompanied by partial evaluation of its body. The AE might be subjected to pre-processing of various kinds, e.g. to disentangle combinations once for all or to remove its dependence on an arbitrary choice of identifiers occurring bound in it. The pre-processing might be more elaborate and perform symbolic reduction.

The particular evaluation process described in section 15 will be called *normal* evaluation, and its significance partly lies in that many other evaluation processes can be described in terms of it; i.e. they can be specified as a transformation of the AE into some other AE, followed by normal evaluation of the derived AE. Further discussion of evaluation processes and of their mutual relationships is outside our present scope.

18. EVALUATING WITH A DIGITAL COMPUTER

This section describes how a "state", in the sense of section 15, can be represented in the instantaneous state of a digital computer, and how the transformation formalized in (15.2) can be represented by a stored program. The method chosen here is one of many and is distinguished by its simplicity in description, rather than by its cheapness. It will hold no surprises for anyone familiar with the "chaining" techniques of storage and location pioneered in list-processing systems. It is presented as a demonstration of

possibility, not of practicability. What follows is a brief sketch. For the sake of greater concreteness, a possible detailed development is given in Appendix 2.

Representing Each Composite Object by an Address

Each component of a state, from the entire state downwards, and including such COs as are definable objects, can be represented in a computer by an address. The way of doing this is closely related to the structure definitions used to introduce the various COs concerned. For, given that the components can be represented by addresses, the complete CO can be represented by a short segment of store, large enough to contain these addresses (and, if the CO is one admitting alternative formats, a distinguishing tag). So the complete CO can also be represented by the *address* of this short segment. There is need for one fixed area in the store, large enough to hold an address and representing *the* current state. The merit of this method is that the predicates, selectors, and constructors can be represented by stored programs whose size and speed is independent of the size of the COs operated on. Hence this is also true of the information-rearranging steps that occur during evaluation, namely (1), (2*a*), (2*b*), (2*c*1) and (2*d*) of section 15; for each of these is a composition of predicates, selectors and constructors.

Each of these steps can be represented by a stored program of ten or twenty orders in most current machine-codes. Obviously, the possibility arises of designing a machine-code that favours these steps. However the implementation sketched here has less claim to such embodiment than some others whose properties are briefly referred to below.

Shared Components

One consequence of this method is the presence of "shared" components For instance, suppose the environment denoted by

$$derive(assoc(bv\mathrm{X}, x))E$$

is being formed in step (2*c*1). It is possible that a copy of the address representing E is "incorporated" into the new environment. As long as environment components are not updated, the extent of sharing is immaterial. However there are two possible developments in which it would become important to consider precisely what components are shared.

(*a*) We might vary the evaluation process by introducing a preparatory investigation of each AE, to determine whether any of the transformations of COs that occur during its evaluation can be performed by overwriting rather than by reconstructing.

(*b*) We might generalize AEs by introducing a fourth format playing the role of an assignment.

Representing Each Non-composite Object by an Address

The possibility of using our storage technique depends on

(1) being able to represent each *non-composite* definable object by an address: namely, identifiers, primitives and all results of evaluation other than closures and composite definable objects;
(2) being able to represent each basic function f by a stored program such that, if the head of the stack represents x, the program replaces it by an address representing fx.

Representing Y

If we consider a specific (powerful) set of primitives, comprising some basic numbers, some numerical functions, the basic list-processing primitives, and Y, only the latter involves any unfamiliar technique. Y can be represented by a stored program that, given an argument F at the head of the stack, performs the following steps:

1. Take a fresh cell z, whose address is Z.
2. Use Z as a spurious argument for F, producing a result-address Z'.
3. Copy the word addressed by Z' to the cell z. Then Z is the required result of Y.

This representation of Y is adequate for the uses of it mentioned in section 11 on "Recursive definitions".

Source of Storage

The stored programs for constructing COs must have access to a source of fresh storage cells, which (unless the machine is to be congested rapidly) must in turn be able to retrieve for re-use used cells that have become irrelevant.

Other Ways of Representing Our Mechanization with a Digital Computer

It was earlier observed that the mechanization in terms of SECD-states is only one of many ways of mechanizing evaluation. Likewise, given a particular mechanization, there may be many ways of representing it with a digital computer. In particular the method just sketched is not the only way of mechanizing SECD-states.

For example, of all the occasions on which a fresh cell is required, there are certain sub-sets that can reasonably be acquired and disposed of in a "last in/first out" (LIFO) pattern. Hence by distributing these requirements among more than one source of fresh cells it is possible to exploit consecutive addressing. In particular by restricting the structure of AEs it is possible to rely exclusively on the LIFO pattern. Such restrictions suggest a pre-evaluational transformation for eliminating expensive structures in favour of equivalent cheaper ones. Such variations are outside our scope.

19. CONCLUSION

We have mentioned several lines of development other than those described in detail. Some of these consist in part of a "sideways advance", a rephrasing of previous work in a new jargon. However a new jargon might have features that justify this procedure. The best claim in the present case seems to be based on the extent to which it isolates several aspects of the use of computers that frequently appear inextricably interwoven.

One such feature is the distinction between structure and representation effected by "structure definitions". For instance the structure of expressions was distinguished from their written representation. Again, the structure of the information that is recorded during evaluation was distinguished from its representation in a computer.

Another separation achieved is that between considerations special to a particular subject-matter, and considerations relevant to every subject-matter (or "universe of discourse", or "field of application", or "problem orientation"). The subject-matter is determined precisely by the choice of primitives and is not affected by the choice of names for them, or of rules for writing expressions (except that these rules might narrow the subject-matter by making some AEs unwritable).

The relationship between expressions and their written representation encompasses all that is customarily called the "syntax" of a language and part of what is customarily called its "semantics". The chosen name/value relation, together with the primitives themselves (that is to say, the applicative relationships between them) constitute the rest of what is customarily called the "semantics" of a language in so far as it is distinct from the semantics of other languages.

These remarks about "languages" are subject to an important qualification. They apply only to languages that can be considered as AEs plus syntactic sugar. While most languages in current use can partly be accounted for in these terms, entirely "applicative" languages have yet to be proved adequate for practical purposes. Whether or not they will be, and whether their

interesting properties can be extended to hold for currently useful languages, are questions outside the scope of this chapter.

20. RELATION TO OTHER WORK

Most of the ideas presented here are to be found in the literature. In particular Church and Curry, and McCarthy and the ALGOL 60 authors, are so large a part of the history of their respective disciplines as to make detailed attributions inevitably incomplete and probably impertinent.

The criterion of "semantic acceptability", whereby a proposed rendering in terms of AEs can be judged correct or incorrect, is closely related to what some logicians (e.g. Quine (1960) call "referential transparency", and to what Curry (1958) calls the "monotony" of equivalence).

Structure definitions are in some sense merely a convenient way of avoiding the uninformative strings of a's and d's that occur in LISP's 'cadar', 'cadaddr', etc. (McCarthy 1960). However they have another merit, that of being less associated with a particular internal representation, and, in particular, with a particular ordering of the components. (Gilmore (1963) effectively uses "selectors" to avoid entanglement with a specific written representation of expressions.)

Church (1941), Curry and Feys (1958) and Rosenbloom (1950) all include discussions of how to eliminate various uses of bound variables in terms of just one use, namely functional abstraction; also of how to eliminate lists and functions that select items from lists, in terms of functional application. The function Y is called Θ by Rosenbloom and **Y** by Curry, and roughly speaking $Y\lambda$ is McCarthy's *label.*

Formalizing a system in its own terms is now a familiar occupation. The relative simplicity of the function *val*, compared, say, with LISP's *eval*, *apply*, etc. (McCarthy 1960, 1962), is due partly to the fact that it treats the operator and operand of a combination symmetrically.

The formalization of a machine for evaluating expressions seems to have no precedent. Gilmore's machine is specified by a flow diagram. The relative simplicity of the function *Transform*, compared with his specification, is also due in part to the above-mentioned symmetry. Closures are roughly the same as the "FUNARG" lists of McCarthy (1960) and the PARD'S of Dijkstra (1962). (This method of "evaluating" a λ-expression is to be contrasted with "literal substitution" such as is used in Church's normalization process, in Gilmore's machine, and in the system of Wirth (1963). For example, it obviates the otherwise ubiquitous qualification: "with such systematic changes of bound variable as are necessary to avoid collisions".)

APPENDIX

An Illustration of Evaluation

WHAT follows is an indication of the steps taken by the SECD-machine to evaluate

thrice thrice square 5

relative to an environment E_0, in which *thrice* is defined by "$\lambda f.\lambda x.f(f(fx))$", '5' denotes 5, and *square* is a basic function whose application takes just one step (e.g. done by floating-point multiplication with 30-bit exponents). It can be read either as an illustration of the behaviour of the SECD-machine, or as an illustration of a pencil-and-paper method of evaluating AEs.

The following conventions are used:

(a) Each line indicates a transitory state under the headings STACK, ENVIRONMENT, CONTROL, DUMP. A gap indicates no change. To conform with current practice the stack is written with its head (i.e. its youngest item) on the *right*, not on the left as in many other lists in this book. A single item occurring in a stack-part or control-part indicates a list of length one.

(b) We abbreviate "*square*" to '*s*', and "*thrice*" to '*t*' (unrelated to our earlier use of '*t*' to designate the list-beheading function *tail*).

(c) The λ-expression "$\lambda x.f(f(fx))$" occurs frequently in the control-parts of SECD-states and environments. It is denoted shortly by 'λx'.

(d) New environments are named and specified (in so far as is relevant) in the form

$$E_2 = (f = s, x = 5)$$

(e) New dumps are named and specified in terms of old ones by a statement such as

$$D_1 = (5, s), E_0, (ap, ap), D_0$$

(f) The closure that arises by evaluating 'λx' in environment E is denoted by '$\lambda x::E$'.

The steps are shown in Table 1. There are, in all, 173 steps summarized in Table 2.

The question whether scanning *ap* involves a closure or a basic function depends on context. Of the 14 occasions on which $f(f(fx))$ is scanned in Table 2, f is a closure 5 times and a basic function (namely *square*) 9 times.

TABLE 1

STACK → head	ENVIRONMENT	CONTROL head ←	DUMP
( )	E_0	$tts5$	$D_0 = S_{-1},E_{-1},C_{-1},D_{-1}$
5		$5,tts,ap$	
5,s		tts,ap	
5,s,t		s,tt,ap,ap	
5,s,t,t		tt,ap,ap	
		t,t,ap,ap,ap	
		t,ap,ap,ap	
		ap,ap,ap	
( )	$E_1 = (f = t)$	λx	$D_1 = (5,s),$ $E_0,$ $(ap,ap),$ D_0
$\lambda x::E_1$		( )	
5,s,$\lambda x::E_1$	E_0	ap,ap	D_0
( )	$E_2 = (f = t, x = s)$	$f(f(fx))$	$D_2 = 5,E_0,ap,D_0$
	...		
s,t		ap,f,ap,f,ap	
( )	$E_3 = (f = s)$	λx	$D_3 = ( ),$ $E_2,$ $(f,ap,f,ap),$ D_2
$\lambda x::E_3$		( )	
$\lambda x::E_3$	E_2	f,ap,f,ap	D_2
$\lambda x::E_3$,t		ap,f,ap	
( )	$E_4 = (f = \lambda x::E_3)$	λx	$D_4 = ( ),$ $E_2,$ $(f,ap),$ D_2
$\lambda x::E_4$		( )	
$\lambda x::E_4$	E_2	f,ap	D_2
$\lambda x::E_4$,t		ap	
( )	$E_5 = (f = \lambda x::E_4)$	λx	$D_5 = ( ),E_2,( ),D_3$
$\lambda x::E_5$		( )	
$\lambda x::E_5$	E_2	( )	D_2
5,$\lambda x::E_5$	E_0	ap	D_0
( )	$E_6 = (f = \lambda x::E_4, x = 5)$	$f(f(fx))$	$D_6 = ( ),E_0,( ),D_0$
	...		
5,$\lambda x::E_4$		ap,f,ap,f,ap	
A			
( )	$E_7 = (f = \lambda x::E_3, x = 5)$	$f(f(fx))$	$D_7 = ( ),$ $E_6,$ $(f,ap,f,ap),$ D_6
5,$\lambda x::E_3$		ap,f,ap,f,ap	

B			
( )	$E_5 = (f = s, x = 5)$	$f(f(fx))$	$D_8 = ( ),$ $E_7,$ $(f,ap,f,ap),$ D_7
	...		
5,*s*		*ap,f,ap,f,ap*	
25		*f,ap,f,ap*	
25,*s*		*ap,f,ap*	
5^4		*f,ap*	
5^4,*s*		*ap*	
5^8		( )	
5^8	E_7	*f,ap,f,ap*	D_7
$5^8,\lambda x::E_3$		*ap,f,ap*	
Like B			
	...		
$(5^8)^8$	E_7	*f,ap*	D_7
$(5^8)^8,\lambda x::E_3$		*ap*	
Like B			
	...		
$((5^8)^8$	E_7	( )	D_7
5^2	E_6	*f,ap,f,ap*	D_6
$5^{2^9},\lambda x::E_4$		*ap,f,ap*	
Like A			
	...		
$5^{2^{18}}$	E_6	*f,ap*	D_6
$5^{2^{18}},\lambda x::E_4$		*ap*	
Like A			
	...		
$5^{2^{27}}$	E_6	( )	D_6
$5^{2^{27}}$	E_0	( )	D_0
$5^{2^{27}}:S_{-1}$	E_{-1}	C_{-1}	D_{-1}

TABLE 2
WHILE SCANNING

	*tts*5	$\lambda x \times 4$	$f(f(fx)) \times 14$	total
exit	1	1 4	1 14	19
identifier	4		4 56	60
λ−exp		1 4		4
ap closure	3 { 3		3 42 { 15	45 { 18
ap basic			27	27
combination	3		42	45
total	11	2 8	8 154	173

APPENDIX II

A Specific Representation of the SECD-Machine

SECTION 18 on "Evaluating with a digital computer" sketched a method of realising the SECD-machine with a computer. We now illustrate that section by a more detailed explanation. As before we are concerned here with possibility rather than practicability. However what follows can also be taken as a point of departure for the design of an *efficient* implementation. We consider a memory of consecutively addressed cells, each capable of holding an address. Following our previous approach, each CO can be represented in two or more consecutive cells, depending on how many components it has and whether it needs a format-tag. Hence we are interested in two sorts of representatives of a CO, *S*—a *long* one (its *long rep*) that includes short representatives of the immediate components of *S*, and a *short* one (its *short rep*) that is the address of the first of a string of cells containing its long rep. In the special case when the number of immediate components is zero we shall not use a long rep; the format-tag itself will be used as the short rep. Similarly non-composite objects might have only a short rep, or might need a long rep (for example a variable length representation of big numbers such as $5^{2^{27}}$).

In our representation of the SECD-machine there is a certain cell, *M*, that always contains the short rep *m* of the current state. When a new CO is constructed, its long rep is usually placed in a newly requisitioned string of cells. However there are some occasions in which it is possible to re-occupy cells that have been previously requisitioned. For example each transition constructs a new SECD-state and could always be realised in four consecutive new cells, reoccupying *m*, so that it points to the first of them. However when merely the S and C components are being changed, it is convenient to leave *M* unaltered and reoccupy two of the (previously requisitioned) cells among the four it points to; again, when resuming a dumped state, *M* is reoccupied, but left pointing to an old cell, not a newly requisitioned one. We describe no other way of recovering previously used store, so this implementation is severely limited in usefulness by the number of cells available. For example the evaluation of

thrice thrice square 5

worked through in Appendix 1, requires several hundred cells.

The short rep of a null-stack, -environment or -control-part, is the format-tag NIL. The long reps of AEs, closures, environments, control-parts and SECD-states are shown in Table 3. Apart from "NIL" there are three "tags" involved: 'ID', 'λ' and 'CLOS'. Apart from these, each cell contains the short rep of some component.

TABLE 3

AEs: identifier	ID | identifier	(2 cells)
λ-exp	λ | bv | body	(3 cells)
combination	rator | rand	(2 cells)
Closures:	CLOS | environment | bv | control-part	(4 cells)
Non-null stacks:	youngest item | tail	(2 cells)
Non-null environments:	youngest layer | parent environment	(2 cells)
Layer of environment:	bv | value	(2 cells)
Non-null control-part	first item | tail	(2 cells)
SECD-state:	stack | environment | control | dump	(4 cells)

In addition to the cell M, containing m, the interpretive mechanism uses one other fixed cell F, containing f, the address of the first free cell.

We now take the six sorts of transition step in turn, and picture how each is represented by a change of computer state.

When the control-part of the current state is null, the relevant parts of the machine are shown in Fig. 1.

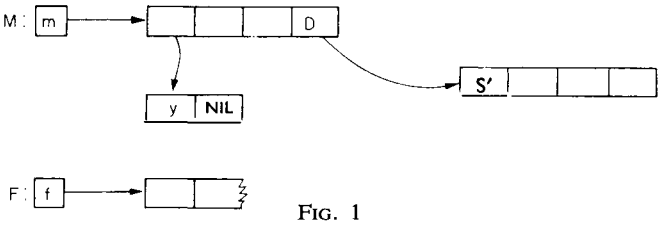


FIG. 1

(Except in the case of new cells and of the small strings of cells in a single long rep, we make no attempt to represent the order of the cells involved.) The final state is that of Fig. 2.

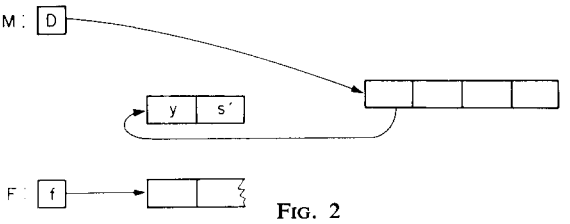


FIG. 2

The required change can, therefore, be pictured by using bold lines to indicate the transfers as shown in Fig. 3.

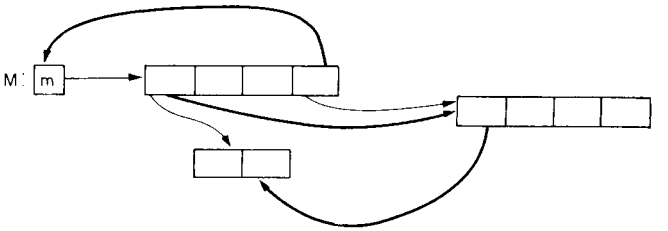


FIG. 3

It can also be symbolized as a multiple (simultaneous) transfer in a notation

$$\left.\begin{matrix}((m+3))\\(m)\\(m+3)\end{matrix}\right\} \rightarrow \left\{\begin{matrix}(m)+1\\(m+3)\\M\end{matrix}\right.$$

Each entry on the left denotes some bit-string (usually an address which is to be copied). Each entry on the right denotes some bit-string (always an address) that addresses the cell that is to receive the copy. So, for example,

$$m \rightarrow M$$

indicates the (vacuous) transfer of the bit-string m to the cell M (whose content is already m). Brackets indicate 'contents of'; so, for example, '(M)' and 'm' denote the same bit-string. So our vacuous transfer can be written

$$(m) \rightarrow m$$

Notice that, on the other hand,

$$m \rightarrow m$$

indicates a non-vacuous transfer, in which the cell addressed by m is reset to point to itself. Similarly, each of

$$M \rightarrow M$$

$$(m) \rightarrow (m)$$

indicates a (different) non-vacuous transfer that leaves a cell pointing to itself.

It follows that

(a) The short reps of the four components of the current state occupy m, $m+1$, $m+2$ and $m+3$. They *are* (m), $(m+1)$, $(m+2)$ and $(m+3)$.

(b) The short reps of the successive current stack items occupy (m), $((m)+1)$, $(((m)+1)+1)$, etc. They *are* $((m))$, $(((m)+1))$, $((((m)+1)+1))$, etc.

(c) The short reps of the successive current environment levels occupy $(m + 1)$, $((m + 1) + 1)$, $(((m + 1) + 1) + 1)$, etc.

(d) The short reps of the successive current control items occupy $(m + 2)$, $((m + 2) + 1)$, $(((m + 2) + 1) + 1)$, etc.

(e) The short reps of successively older dumps occupy $m + 3$, $(m + 3) + 3$, $((m + 3) + 3) + 3$, etc.

It also follows that in the change of state symbolized by

$$X \rightarrow Y$$

exterior brackets round 'Y' indicate that the machine currently contains a pointer to the cell whose contents are to be changed; and exterior brackets round 'X' indicate that the machine already contains a copy of the bit-string which is to be the new occupant of Y.

After this preamble we can specify how each of the other transition steps is represented in two ways; a diagram containing bold arrows, and a symbolic expression. They convey the same information.

Scanning an identifier (Fig. 4)

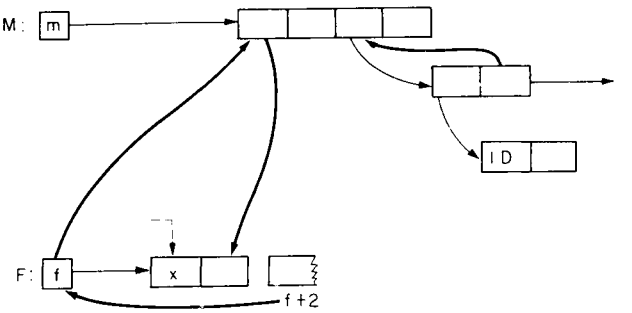


FIG. 4

$$\left.\begin{matrix} x \\ (m) \\ f+2 \\ f \\ ((m+2)+1) \end{matrix}\right\} \rightarrow \left\{\begin{matrix} f \\ f+1 \\ F \\ m \\ m+2 \end{matrix}\right.$$

This uses two new cells. Here x is the bit-string discovered by matching the identifier $(((m + 2)) + 1)$ in the current environment.

Scanning a λ-expression (Fig. 5)

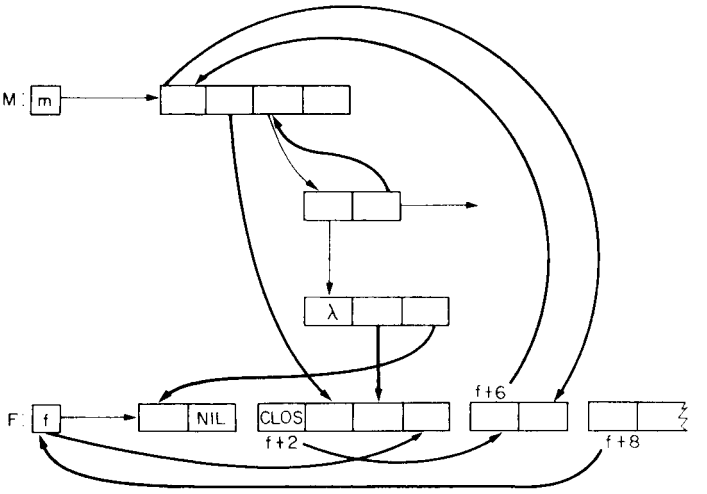


FIG. 5

$$\left.\begin{array}{c}(((m+2))+2)\\ \text{NIL}\\ \text{CLOS}\\ (m+1)\\ (((m+2))+1)\\ f\\ f+2\\ (m)\\ f+6\\ ((m+2)+1)\\ f+8\end{array}\right\} \rightarrow \left\{\begin{array}{l}f\\ f+1\\ f+2\\ f+3\\ f+4\\ f+5\\ f+6\\ f+7\\ m\\ m+2\\ F\end{array}\right.$$

This uses 8 new cells.

Scanning ap and finding a closure (Fig. 6)

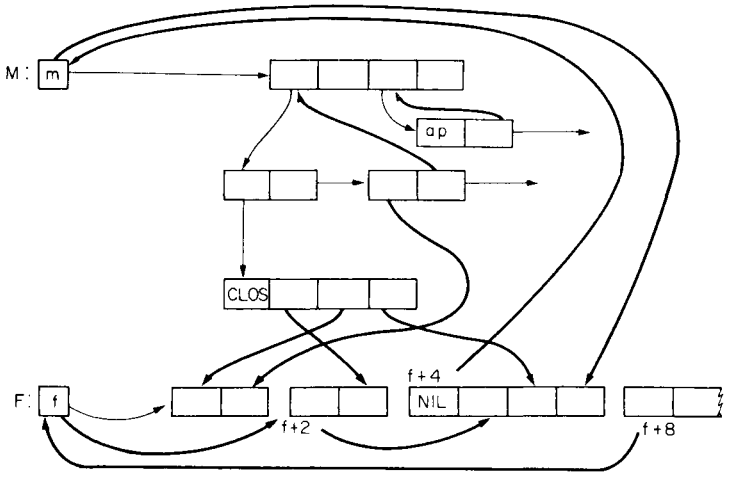


FIG. 6

$$\left.\begin{array}{r}(((m))+2)\\(((m)+1))\\f\\(((m))+1)\\\text{NIL}\\f+2\\(((m))+3)\\m\\(((m)+1)+1)\\((m+2)+1)\\f+8\\f+4\end{array}\right\} \rightarrow \left\{\begin{array}{l}f\\f+1\\f+2\\f+3\\f+4\\f+5\\f+6\\f+7\\m\\m+2\\F\\M\end{array}\right.$$

This uses 8 new cells.

Scanning ap and finding a basic function

Each basic function f is represented by a stored program that resets (m) with a bit-string that is a short rep of fx, where x is the object of which $((m))$ is a short rep. This program may use new cells.

Finding a combination (Fig. 7)

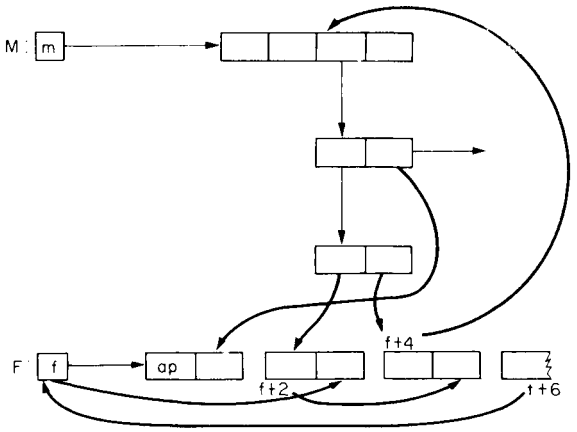


FIG. 7

$$\left.\begin{array}{c} ap \\ ((m+2)+1) \\ (((m+2))) \\ f \\ (((m+2))+1) \\ f+2 \\ f+4 \\ f+6 \end{array}\right\} \rightarrow \left\{\begin{array}{l} f \\ f+1 \\ f+2 \\ f+3 \\ f+4 \\ f+5 \\ (m)+2 \\ F \end{array}\right.$$

This uses 6 new cells.

We have now explained in detail the implementation of the six different transition steps (exit, load identifier, load λ-expression, apply closure, apply basic, disentangle combination). Thus, disregarding any new cells used by the basic function *square*, the evaluation of

thrice thrice square 5

requires 574 new cells, used as follows:

64	scan identifiers	(2 each)	128
4	scan λ-exp	(8 each)	32
18	entries	(8 each)	144
45	combinations	(6 each)	270
			574

It is to be expected that at this rate any useful computation would far exceed the capacity of a reasonably high-speed computer memory. However we have been concerned here to give a concrete example that can be explained in a few pages.

REFERENCES

BACKUS, J. W. (1959) The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. *Proc. ICIP UNESCO*, Paris.

BARNET, M. P. and FUTRELLE, R. P. (1962) Syntactic analysis by digital computer. *Comm. A.C.M.* **5,** 515–26.

BEMER, R. W. (1959) An editorial note in "Techniques". *Comm. A.C.M.* **2,** 10–11.

BROOKER, R. A., MACCALLUM, I. R., MORRIS, D. and ROHL, J. S. (1963) The compiler compiler. *Ann. Rev. in Automatic Programming*, **2,** 229–75.

CHURCH, A. (1941). *The Calculi of Lambda-Conversion*, Princeton University Press.

CURRY, H. B. and FEYS, R. (1958) *Combinatory Logic*, 1, Amsterdam, North Holland Publishing Co.

DIJKSTRA, E. W. (1962) An ALGOL 60 translator for the X1, *Automatic Programming Bulletin*, 13.

FLOYD, R. W. (1963) Syntactic analysis and operator precedence. *J.A.C.M.* **10,** 316–33.

GILMORE, P. C. (1963) An abstract computer with a LISP-like machine language without a label operator, in *Computer Programming and Formal Systems*, ed. Braffort, P., and Hirschberg, D., North Holland Publishing Co., Amsterdam.

IRONS, E. T. (1961) A syntax directed compiler for ALGOL 60. *Comm. A.C.M.* **4,** 51–5.

LANDIN, P. J. (1964) The mechanical evaluation of expressions. *Computer J.* **6,** 308–20.

LANDIN, P. J. (1965) A correspondence between ALGOL 60 and Church's λ notation. *Comm. A.C.M.* **8,** 89–101, 158–65.

MCCARTHY, J. (1960) Recursive functions of symbolic expressions and their computation by machine, Part 1, *Comm. A.C.M.*, **3,** 184–95.

MCCARTHY, J., ABRAHAMS, P. W., EDWARDS, D. J., HART, T. P. and LEVIN, M. I. (1962) *LISP* 1.5, *Programmer's Manual*, Cambridge, M.I.T.

NAUR, P. (Ed.) (1960) Report on the algorithmic language ALGOL 60. *Comm. A.C.M.* **3,** 299–314.

NAUR, P. (Ed.) (1963) Revised report on the algorithmic language ALGOL 60. *Comm. A.C.M.* **6,** 1–17.

PERLIS, A. J. and SAMELSON, K. (Ed.) (1959) Report on the algorithmic language ALGOL by the ACM committee on programming languages and the GAMM committee on programming, *Num. Math.* **1,** 41–60.

QUINE, W. V. (1960) *Word and Object*, New York, Technology Press and Wiley.

ROSENBLOOM, P. (1950) *The Elements of Mathematical Logic*, New York, Dover.

WIRTH, N. (1963) A generalization of ALGOL. *Comm. A.C.M.* **6,** 547–54.

CHAPTER 6

A SURVEY OF NON-NUMERICAL APPLICATIONS

S. GILL

1. INTRODUCTION

In the second part of this book we shall consider some of the actual applications of computers in areas outside those covered by numerical analysis. The term "non-numerical" is convenient, but is not to be taken too literally, because the concept of number plays such an important part in mathematics that one cannot get away from it for long. Indeed, whenever one begins to talk about processing a series of items, one is soon led to consider enumerating them, so that the positive integers at least are nearly always with us.

Most computer applications today (certainly most of those in science and engineering) are essentially concerned with numbers as their "raw material". Numerical analysis is concerned with devising appropriate procedures for such calculations. A great deal of it is related to the effects of using finite approximations to continuous functions (i.e. with rounding and truncation errors), and the problems raised by discontinuities and other inconvenient properties of functions of numbers. Compared with these questions, the matter of organizing and defining the actual processes to be performed is comparatively incidental.

Recently, however, there has been a considerable growth in the use of computers to handle information that is not basically concerned with numbers or with the concepts of classical mathematical analysis. In such applications the problems of describing and controlling the process come to the fore, and in fact they can be studied more conveniently in this context. It must be noted, however, that these problems are by no means absent from numerical calculations, and the task of designing languages and compilers for numerical work raises many of them.

2. BUSINESS AND COMMERCE

Applications to business and commerce form an increasing proportion of all computer applications and will soon form the majority. Although

commercial calculations involve a great many numbers, the arithmetic involved is usually very simple, and much of the data is in a variety of non-numerical forms, such as identifications of things or people, classifications of various kinds, etc. The very mixed nature of the information occurring in business brings its own problems in devising suitable ways of instructing computers in this field. The concepts involved may not rise to a very high level of mathematical abstraction, but the enormous size of some business computer programs calls for powerful programming techniques to aid in writing them quickly and fitting them together properly.

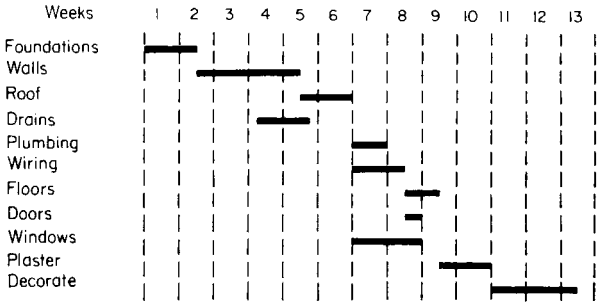


FIG. 1.

However, the more interesting applications from the point of view of this book are those in which the problems or procedures can be stated or executed particularly directly and concisely by using some of the techniques described in the earlier chapters.

Many problems in business management are concerned with choosing or predicting the sequence of operations occurring in some industrial process. The so-called "PERT" idea is a systematic way of working out the dates on which various jobs within a large project will begin and end, given an assumed duration for each job and the interdependences between them. ("PERT" stands for "Project Evaluation and Review Technique".) PERT has only appeared and come into common use within the last few years, and there is no doubt that it owes its inspiration to the kind of outlook engendered by computers.

The schedule of jobs contributing to a project is typically drawn as in Fig. 1. This is how managers have been drawing them for years, and there is nothing too difficult about making an initial draft of such a schedule even for a fairly complicated project, beyond the obvious and unavoidable difficulty of knowing how long each job will take. Once the project is started, however, the manager's job becomes much more difficult. Revisions to the job times are made continuously, and if the schedule were redrawn at a later stage it

might well look like Fig. 2. Each revision made is likely to affect the schedule in a way which calls for special action by the manager, and it is, therefore, important that the effect of each revision should be worked out as quickly as possible. Drawing up the schedule once may be fairly easy, but to redraw it many times at short notice is not. The question therefore arises of devising a

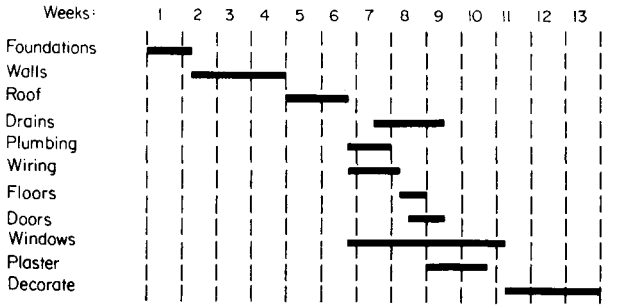


FIG. 2.

systematic procedure, which might perhaps be carried out by a computer, to incorporate revisions as they arise without having to refer the whole schedule back to the person who originally drew it up.

To do this calls for some information about the relationships between the various jobs that does not appear on the diagrams in Figs. 1 and 2. The

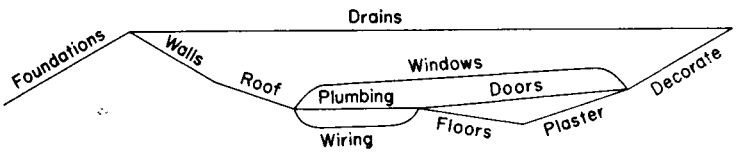


FIG. 3.

person who draws these up is aware of the fact that some jobs cannot begin until others have finished, and he uses this knowledge in preparing the schedule, but does not state it explicitly. To operate a PERT system he must be asked to go back one stage in his thinking, and to state all the relationships of this kind that exist between the jobs. Once he has done this it is possible for an automatic system to prepare not only the initial schedule but every successive revision of it.

The relationships between the jobs can be shown in the form of a diagram like that of Fig. 3. When using a computer, suitable codes and formats must be devised for feeding this information into the computer and storing it there.

PERT is typical of a large class of computer applications concerned with

the timing of processes, which call for a thorough knowledge of ways of programming "non-numerical" computations. PERT can be looked upon as a simulation technique, in that the computer effectively predicts the course of the project. Computers have also been used to carry out many thousands of such simulations, not in order to predict the outcome in any one specific case, but to obtain statistics about the productivity of an organization when handling a typical load of work. This is of considerable use to operational research experts in devising suitable layouts and strategies for many different kinds of factory and transport system. Simulation applications are, in fact, so common that several special programming languages have been devised for such tasks, and a great variety of cases and conditions can now be taken into account.

Simulation has also been used by psychologists to attempt to form theories of human mental behaviour, by experimenting with computer programs. For example, programs that seek to solve problems by trial and error have been written with a view to comparing their performance, stage by stage, with that of a human.

3. TRAFFIC CONTROL

From the abstract simulation of factory operations, traffic, etc., computers have now begun to move into the actual physical control of these things. Of the various kinds of traffic the one that has been given most attention is air traffic. This is not because air traffic problems are easily solved by computer, but simply because this is at present our most critical traffic control problem. Aeroplanes get faster and more numerous every year, and the human controllers are being driven more and more frantic. To some extent the situation can be relieved by just putting more people on the job, but the more people there are the more time they spend communicating with each other instead of with the pilots.

So for some time now a considerable amount of attention has been given to the possibility of replacing air traffic controllers by computers. Already computers have been used to digest the available information about the traffic situation and to present this in suitable forms for the human controllers, and experimental systems have been built for actually issuing control instructions.

The control of air traffic is a tough logical problem, characterized by a three-dimensional environment, lack of physical lane markers, and the fact that aircraft in flight must maintain speed within certain limits. Road traffic does not possess these awkward characteristics, and presents a logically simpler problem; the fact that computers have so far hardly been used at all for this purpose is due to the lack of urgency in it. However

Toronto is now installing a computer to control the traffic at 100 or more intersections in the city. It is hoped that the improvement in traffic flow will be equal to that which might have been achieved by building several new roads, which would have cost a great deal more than the computer.

Rail traffic also presents a comparatively simple logical problem since it is constrained to move along fixed tracks, but so far there is very little sign of any railway taking computer control seriously. Computers have, however, been installed to keep up-to-the-minute records of the whereabouts of freight shipments and of wagons, and to construct timetables.

School timetables seem at first sight simpler than railway timetables, and the requirements can certainly be stated more shortly, but, nevertheless, their construction poses a quite difficult mathematical problem (see Chapter 9). However, Gottlieb (1964) has recently reported some promising results from an experimental computer program for solving this problem, and there are now good prospects of being able to solve it in practical cases before long.

4. INFORMATION RETRIEVAL

A large number of tasks come under the general heading of "information retrieval" and are leading people to look to computers for help. Basically, retrieving information amounts merely to consulting some kind of file, but there may be a certain amount of logical processing involved for which a computer might be used. For example it may not be possible to arrange the file so that every enquiry can be answered by going straight to a single item; it may be necessary to refer to several items, or to search for items having certain characteristics. Information retrieval is simplest in a specialized field where the facts are all of a similar nature and can be easily catalogued. An example is travel timetables, which could be quite easily scanned by a computer to answer the majority of enquiries very quickly. Another application is in the chemical industry, where it is required to pick out chemicals with given properties; or in medicine, to find an explanation of a given set of symptoms; or in law, to find legislation or precedents relating to a given case. All these are on the verge of becoming important practical uses for computers.

Such applications will in due course become more general, taking in more and more miscellaneous information and answering a greater variety of enquiries. As this happens, the language in which the information and enquiries are received will begin to look more like plain English, though it will be a long time indeed before a computer will be able to absorb the *Encyclopaedia Britannica* and to answer everyday questions. To do this will require, amongst other things, some considerable advances in automatic procedures for analysing and synthesizing English sentences.

5. LANGUAGE TRANSLATION AND ANALYSIS

Automatic translation between natural languages has been the objective of an enormous research effort during the last ten years, but it is still not quite within our grasp. The translations that have been done by computers so far are of very poor quality and very limited use. However a great deal has now been learnt about the syntactic analysis of natural languages by computer, and this is adding to our knowledge of languages generally.

There are in fact many ways in which computers can be used to handle natural language material, without attempting the very difficult task of translation. Already the rules of grammar have been tested and refined by incorporating them in computer programs for analysing specimen sentences, or for synthesising random sentences, to show up any shortcomings. There are also many simpler jobs, such as forming word counts and other statistics for the comparative study of literature, and the preparation of indexes and concordances. One type of index that has recently come into vogue, because it is easily prepared by a computer, is the so-called "key word incon text" type. This is an index of the titles of documents, arranged in alphabetical order of the major words in them (so that most titles appear several times, once for each major word). Each entry is shifted to the right or left so that the word under which it is being listed appears in a column in the *centre* of the page. When consulting the index, relevant words can be found by running the eye down this column; when a word has been found the title containing it can be read off by scanning that line from left to right.

6. PRINTING AND COMMUNICATIONS

The editing of printed material is another area that has recently been given some attention, particularly since the introduction of photographic typesetting is likely to call for new methods of making corrections. Most computer programs are concerned to some extent with the layout of printed results, so there is nothing essentially new about using computers for this kind of work, but if a program is to be designed specifically for this task a particularly effective notation must be found in which to specify the editing processes required.

Plain language messages are also encountered in telecommunication. Ordinary speech is not yet amenable to computer analysis (although some experimental devices have done this in limited contexts), but Telex (teleprinter) messages are in a form that can be handled directly by computers, and there are already some communication systems that include computers. The duties of the computer in this context are miscellaneous: it can route messages according to traffic conditions; it can duplicate messages intended

for more than one destination; it can keep accounts and statistics; it can compress messages by making use of any known redundancies in the language; and it can help in error prevention, either by applying redundancy checks or by arranging for the automatic repeating of messages.

7. DESIGN

Computer engineering finds several uses for computers, mostly non-numerical. The simulation of various aspects of computer operation can be a useful aid to designers of computers, but the central task is that of processing the actual design itself. The essence of the design appears initially as a "logical" diagram, showing the interconnections between various standard circuit units. This must be transformed into "wiring lists" showing, in appropriate form and sequence, every connection to be made in the actual machine. There are also various by-products such as checks on the loads borne by the power supplies and signal sources. As in the case of PERT, the real advantage of using a computer for this work appears when the design amendments begin to arrive. The rapid assimilation of these into the system reduces considerably the confusion and delay in getting a new machine into production. One company even produces revised logical diagrams from the computer, using a specially adapted printer, in a standard format designed for easy reference by maintenance engineers, etc.

Part of the work of designing computers consists of deciding on the actual physical layout of the components and units. The main consideration in doing this is to reduce the lengths of the wires carrying signals, so as to minimize interactions between them, stray capacities, and time delays due to the finite speed of electrical signals. So far these matters have not been critical, and layout has been done by hand, but automatic layout is now the subject of a considerable amount of research. An additional constraint is imposed by the use of printed circuits, requiring that interconnections between components be confined to a very small number of layers, in any one of which no two wires may cross.

The printed circuit problem is essentially a topological one, rather than a geometrical one. However there are already several instances of computers handling geometrical information—including, for example, air traffic surveillance, which has already been mentioned. Another example is the preparation of cutting instructions for automatic machine tools. These instructions have to be presented to the tools in tedious detail, rather like computer programs in machine code, and we can use techniques similar to those used in compilers, to prepare the final instructions by computer, starting from some more general and concise description of the geometry of the required product.

Attempts have also been made to use a computer to aid the draughtsman in rapidly preparing and revising engineering drawings. The most successful attempts have been made by causing the computer to process the drawing as the draughtsman conceives it, displaying it on a cathode ray tube as it develops. New features can be drawn in using a "light pen", a hand-held photo-cell which is placed on the screen, and whose position can be detected by the computer. (This is done by feeding the signal from the photo-cell back to the computer as an input; the computer finds the cell by trial and error, by illuminating various points on the screen to see which ones cause a signal in the cell.) The computer can then help the draughtsman in various ways, e.g. by rendering perfect straight lines, circular arcs, etc., as required; by magnifying portions of the drawing for attention to details; by shading areas, and inserting lettering; by rearranging major parts of a drawing and so making revision comparatively easy, and so on. While this is being done the drawing is recorded in the store of the computer as a succession of features, each coded symbolically. This record can be preserved, e.g. on magnetic tape, and made available again later for any further revisions which may be required, and perhaps also used as part of the input to a program producing instructions for a machine tool.

It is possible that this kind of thing may be extended to three dimensions, so that an architect, for example, may get perspective views of a new building presented automatically.

8. THEOREM PROVING

Another example of the use of a computer to manipulate geometrical information was a program written by Gelernter (1960) to seek proofs of theorems in geometry. This was in fact only one of many attempts to use a computer to prove theorems, but it is the only one known to the writer to have been directed specifically at geometrical theorems.

The idea of using a computer has a rather special place in mathematical logic, which is very much concerned nowadays with the theoretical possibility or impossibility of devising systematic ways of solving classes of problems, such as proving certain types of theorem. These systematic ways are essentially programs for an idealized computer. Of course the theoretical possibility of a program does not mean that an efficient practical program can be written, but logicians have had to consider ways of specifying computing procedures (or "algorithms"), and a few of them approached the task of programming an actual computer to prove theorems (see Chapter 7). Simple theorems have been proved at high speed, but programs written so far have taken an exorbitant amount of time to prove even a mildly interesting theorem, and it is clear that ways of improving the efficiency must be found

before these programs can be of any practical use. However it is likely that effective programs will be developed in due course, and that computers will become common tools for the pure mathematician.

Some theorem-proving programs proceed by making a systematic test of alternatives; others explore possibilities in a less regular way, perhaps making some random choices (the so-called "heuristic" approach). Similar techniques can be used in solving other problems, e.g. in playing various kinds of game (see Chapter 8). Programs have been written for playing chess, draughts, nim, noughts and crosses, and several other games.

9. LEARNING

It is in the area of game-playing that most attention has been given to the possibility of utilizing the computer's experience in several situations to improve the effectiveness of the program, i.e. getting the computer to "learn". This can be done either by keeping a record of previous games, so that if a situation is encountered that happens to have appeared before, the results of the previous games can be used to guide the play; or by expressing the strategy as a function of variable parameters, and gathering statistics showing how the probability of success depends on the values of the parameters. Neither method is very effective in extending the potency of the program far beyond that provided by the programmer, although a mild improvement in performance can usually be observed after a long run of trials. It will be necessary to give computers very much greater analytical powers if automatic learning is going to become a substitute for human programming effort (see chapter 8).

This begs the question of how powerful a program must be, in order to be able to improve its own analytical powers by learning, so that no further human programming is needed. The word "powerful" is of course undefined; it could mean complicated, or difficult to write, or simply big. If we accept the hypothesis that the human brain is purely mechanistic, then an upper limit would seem to be provided by the information content of the human chromosomes, or of that part of them that relates to the brain. No one has put forward an argument to suggest a lower limit. Programs written so far have achieved only trivial and superficial forms of self-improvement, and there is no sign that they are yet approaching "criticality". The possibility exists that some radically new principle will be discovered by which a program of a modest size can be made indefinitely self-improving, but this possibility is dwindling rapidly as the research effort in this area mounts. It seems more likely now that we shall have to progress through a long series of special forms of learning process, becoming gradually more powerful and general.

As a final example, one special application of automatic learning might be to music. Several programs for composing music have been written, with all the relevant rules provided by the programmer. These rules are not easy to enunciate (beyond a few obvious ones which prevent the music from being really bad), so existing programs have been hard to write and the results have been mediocre. It is conceivable that in the future a learning program might analyse some model compositions and derive the rules for itself.

10. REFERENCES

CSIMA, J. and GOTTLIEB, G. C. (1964) Tests on a computer method for constructing school timetables. *Comm. ACM.* **7**, 160–163.

GELERNTER, H., HANSEN, J. R. and LOVELAND, D. W. (1960) Empirical explorations of the geometry theorem proving machine, *Proc. of the Western Joint Computer Conference* **17**, 143.

CHAPTER 7

THEOREM-PROVING IN COMPUTERS

D. C. COOPER

1. INTRODUCTION

As may be seen from the list of references at the end of this chapter a large number of papers have appeared during the last seven years on proving theorems in computers. What has motivated this and is it all worth while? How far have we progressed and what lines of attack are there? The bulk of this chapter will consist of a review of the relevant literature, showing the various methods proposed so far for mechanical theorem-proving and what they have actually achieved. We shall not be concerned with how the various methods are implemented in a computer but be content with exhibiting the various algorithms used in the language of the subject itself, taking it for granted that it is then possible to produce a computer program.

Techniques for rapidly converting algorithms into computer programs in order to try out ideas are of course very important. In the field of theorem-proving these algorithms are concerned with manipulating various structures, with testing whether a given structure is of a certain form and with applying transformations of various kinds. These manipulations are intrinsically non-numerical in character, although of course it is possible to code up the expressions and transformations numerically. They are thus well suited to the non-numeric information-processing languages described elsewhere in this book. Indeed, progress in the field of proving actual theorems in computers must proceed much more rapidly in the environment of a large high-speed digital computer with a good problem orientated symbol manipulation language, since one is able to try out various ideas on a computer with much less effort on the inessential problem of converting the process to be tried into an actual machine program. (The need for a large fast computer is mainly due to this mismatch of the problem to the design of the computer.) Wang (1960b, page 232) gives some remarks on the coding of a particular theorem-proving program, Wang (1960a, 1963b) makes some comparisons between proving and calculating, whilst some of the papers on actual theorem-proving programs such as those of Prawitz, Prawitz and Voghera (1960) give details of the machine implementation.

Why prove theorems in computers? The answer is surely the same as to the more general question 'why prove theorems'? A great deal of scientific research can be thought of within the framework of proving theorems. The first and really interesting creative activity is the formulation of interesting theorems, which we can then try to prove. We cannot expect computers, in the immediate future anyway, to help very much with the former problem, though Wang (1960a) has carried out some experiments in this direction using the propositional calculus. It seems very natural to carry out research into using computers to prove theorems, rather than formulate them. Proving a theorem, as we remember from our school-days, can sometimes be a straightforward process. Any method tried seems to work, or sometimes seems to depend on a particular trick suggested by the nature of the theorem and by our previous experience with like theorems. In any case there are often a great many simple steps which are obvious to make and which we tend to skip over in drafting a proof, possibly making errors, and these are steps which it seems natural to get computers to perform for us.

An objective which may well be achieved in the near future is to have a man draft out the steps in the proof of a theorem, or as in a program of J. A. Robinson (1963) to suggest likely substitutions, and then to have a machine check the man's steps and fill in the missing ones. A further objective would be to get the machine to suggest lines of attack. This is part of a more general problem known as 'pattern recognition' which is receiving a great deal of attention in the computer world today. It would thus appear that we need computers to check proofs, a mechanical procedure they are well adapted to, and to produce proofs since filling in the missing steps can be thought of as a sub-theorem of the original theorem. McCarthy (1962) in an article describing a computer program implementing a proof-checking system describes the checking of mathematical proofs as potentially one of the most interesting and useful applications of automatic computers and suggests ways in which such a program could be used. The first computer program for proving theorems, that of Newell, Shaw and Simon (1957) was, as we shall see later, more in the spirit of suggesting lines that might produce proofs than in the form of a definite theorem-proving algorithm.

However, it remains true that we wish to use all possible theory to produce as good a theorem-proving program as possible, and this is the aim of most of the papers quoted at the end of this chapter. It is too idealistic to hope that the end of such research will be a program to which we give as input say Goldbach's conjecture or Fermat's last theorem, and sit back for an answer from the computer, 'Yes! this is a valid theorem'. Yet the more powerful a theorem-proving program we have the more use it will be, both for proving theorems entirely by itself and also as a tool for use by man in searching for proofs of interesting theorems beyond the capability of the machine alone.

Apart from regarding theorem-proving as the end in itself, a theorem-proving routine could well form a useful part of some other program. As we progress in our sophistication in the use of computers so we need to introduce more complex "reasoning power" into our programs. To make a useful integrated system in a computer the ability of the computer to deduce new facts from facts it already has, or to prove or disprove facts it thinks are true, will surely provide much added power to the system. The necessity for a theorem-proving sub-routine in a fact retrieval system has been pointed out by W. S. Cooper (1964), who described an experimental program on the IBM 7090 in the form of a question-answering system in elementary chemistry. This incorporates a primitive theorem-proving mechanism so that questions may be posed whose answers depend on making logical deductions from the known facts. A similar but more general scheme is described by Raphael (1964).

Readers interested in the motivation, the general difficulties and further general remarks on 'Mechanical Mathematics', as Wang calls the field, should consult the papers of Wang quoted at the end of this chapter.

2. DISCIPLINES IN WHICH TO PROVE THEOREMS

A great many processes we carry out can be thought of as 'theorem-proving', even though they are not directly such. We shall only be concerned with those activities *normally* thought of as theorem-proving. Almost all the literature of the subject has concerned itself directly with theorems of symbolic logic, but papers have also appeared on theorem-proving in geometry, for example by Gelernter (1959) and Gelernter, Hansen and Loveland (1960), and in the theory of equations by A. Robinson (1960, 1963).

It is natural to start with theorems of symbolic logic, in particular with the predicate calculus, since all mathematical arguments may be formalized in such terms. Thus to assert that a statement S is a theorem of some particular mathematical discipline is to assert that $A_1 \wedge A_2 \wedge A_3 \ldots \wedge A_n \rightarrow S$ is a theorem of logic, where $A_1, A_2, \ldots, A_n$ are axioms of the particular discipline. Although it is certainly true that each particular discipline brings with it methods of proof particularly suited to it, the ability to deal with theorems of the predicate calculus must form a basis upon which to build. Research into the kind of language in which to express our theorems so that a computer may help us is vital; we will not get very far if our only communication is via the predicate calculus. We need a practical language (or perhaps a special language for each discipline) easy to express our thoughts in, and one which the computer may translate into some system in which it can express theorems and try to prove them, or assist us in proving them. However it seems certain that this language will need some formalism similar to that of

the predicate calculus. Thus research into theorem-proving programs in the predicate calculus is a very valuable first step. It would also perhaps be possible to bias this language towards the easier proving of theorems. Thus J. A. Robinson (1964) gives a logical basis for the predicate calculus different from previous bases, one in which it is easier to investigate automatic theorem-proving. Space precludes an adequate treatment of the propositional and predicate calculi; for this any standard book on mathematical logic should be consulted. However in order to introduce terminology we give a summary and examples of theorems from each calculus.

The Propositional Calculus

The basic entities here are *statements* for which we use the symbols p, q, r, etc.; these statements may have two values; they may be *true* or *false*. From these statements by using the *truth-functional connectives* we can form the *well-formed formulas* (abbreviated wffs) of the propositional calculus. These are compound statements whose truth or falsity are determined by the known properties of the truth connectives once the truth or falsity of all their constituent statements is known. The usual truth-functional connectives are

$\sim u$ (not u) which is true if u is false and vice-versa

$u \vee v$ (u or v) which is true if either u is true or v is true or both are true.

This is known as the *disjunction* of u and v.

$u \wedge v$ (u and v) which is true if both u and v are true.

This is known as the *conjunction* of u and v.

$u \rightarrow v$ (u implies v) which can be defined as $(\sim u) \vee v$.

$u \leftrightarrow v$ (u if and only if v) which can be defined as $(u \rightarrow v) \wedge (v \rightarrow u)$

Here u and v may be any statements or wff.

Examples of wff's of the propositional calculus are

$$\begin{aligned} &p \\ &p \rightarrow (q \leftrightarrow p) \\ &p \vee (\sim p) \\ &(\sim p \wedge \sim q) \rightarrow (p \leftrightarrow q) \end{aligned} \tag{2.1}$$

A wff of the propositional calculus is a *theorem* if it takes the value "true" whatever values may be assigned to its constituent statements. Thus in the examples (2.1) the first two are not theorems, the last two are.

The Predicate Calculus

Strictly we should refer to this as the restricted predicate calculus; it is also known as the first order predicate calculus or quantification theory. In this calculus we allow our statements to have some structure, in particular we allow their truth or falsity to depend on a variable which can take on any value from some universe. We also introduce the idea of quantifiers corresponding to the notions of "for all values" and "for some value".

We take as *variables* $x\ y\ z$; they may take on any value from some preassigned set called the *universe* (finite or infinite). Corresponding to the statements of the propositional calculus we have *predicates* (or *predicate functions*) such as $F(x)$, $G(x, y, z)$. These may have any number of arguments and may take on the values *true* or *false*, this value depending on the particular assignment to the arguments from the universe.

These predicates may be joined by the *truth-functional connectives* previously defined to form *well formed formulas* (wffs) of the predicate calculus. We may also form a wff by taking a wff involving a variable, $W(x)$ say, and prefixing it with a quantifier, either an *existential quantifier* to form $(Ex)W(x)$ (intuitive meaning: there is an x such that $W(x)$ is true) or a *universal quantifier* to form $(x)W(x)$ (intuitive meaning: for all x, $W(x)$ is true). These wffs may of course be further combined by means of the truth functional connectives or may have further quantifiers placed in front of them.

An occurrence of a variable x in a wff as a quantified variable [such as the x in $(x)W(x)$] is said to be a *bound* occurrence and may be replaced everywhere it occurs by any other variable not occurring in the wff without changing the meaning of the wff. An occurrence of a variable in a wff not as a quantified variable is called a *free* occurrence.

The propositional calculus is part of the predicate calculus; a predicate function with no arguments corresponding to a statement. Examples of wffs of the predicate calculus are given by

$$
\begin{gathered}
(x)F(x, y) \rightarrow G(a) \\
(a)(Eb)(((Ex)(y)F(x, y) \wedge H(a)) \rightarrow H(b)) \\
(Ex)(y)F(x, y) \rightarrow (y)(Ex)F(x, y) \qquad (2.2) \\
(Ex)(Ey)(z)((F(x, y) \rightarrow (F(y, z) \wedge F(z, z))) \wedge \\
((F(x, y) \wedge G(x, y)) \rightarrow (G(x, z) \wedge G(z, z)))
\end{gathered}
$$

In the first of (2.2) x is a bound variable and a and y are free variables; in the rest of the examples all variables are bound.

A wff of the predicate calculus is a *theorem* if, for all non-empty universes, for any assignment of elements of the universe to free variables of the wff and

for all possible assignments of predicate functions with arguments in the universe to the predicates of the wff, the wff takes the value "true".

The first two examples of (2.2) are not theorems; the last two are.

Further Remarks

In explaining what a theorem is we have adopted a semantic view, i.e. we have used the concept of the intended meaning of the expressions so that any wff that is always true (whatever interpretation is given to its constituents) is a theorem. An alternative approach is the syntactic one, where we lay down certain wffs as being axioms and give some deduction rules for transforming wffs into further wffs. Our theorems are then just the axioms together with all those wffs which may be obtained by using the deduction rules applied to axioms or to previously proved theorems. This definition of theorem makes no reference to meaning, or to universes of objects, but defines theorem in terms of purely mechanical manipulations of logic expressions. The equivalence of these two definitions of theorem is known as the completeness theorem (strictly metatheorem). Completeness of a discipline means that any true 'sentence' that can be formulated in the discipline can be deduced to be true within that discipline. It is well known that both the propositional calculus and the predicate calculus are complete.

Both approaches to a theorem have been used in the literature on mechanical theorem proving. Thus some programs attempt to form a proof of a theorem in the syntactic sense, a series of steps starting from an axiom or axioms and each being deduced from previous steps; whilst other programs are more in the spirit of the semantic view-point. In fact most of the programs fall into the latter category. The general approach of many programs is: if T has to be proved a theorem, form $\sim T$ and try to define a universe and predicate functions over this universe so that $\sim T$ is "true" (this is referred to as setting up a model of $\sim T$). If one can prove that this is impossible then $\sim T$ must always be "false", i.e. T is a theorem.

Two additions to the predicate calculus should be mentioned. The first is the allowing of *functions* over the universe taking as value some element of the universe (in contrast to the previous predicate functions taking either "true" or "false" as value). These functions, or compositions of them, may then be used as arguments of the predicate functions. We extend the definition of a theorem to include all possible assignments of the functions occurring in the wff. We use small letters for such functions, keeping capital letters for predicates. Thus the following is a wff of the predicate calculus with functions (not a theorem)

$$(Ez)(x)F(x, f(z)) \rightarrow (F(f(a), f(a)) \vee G(g(a, b)))$$

Even if the original proposed theorem does not contain functions it is often useful to introduce them, as will be seen later.

A second addition to the predicate calculus (and also to the propositional calculus) is the allowing of the = sign as a primitive symbol (expressing equality of elements of the universe). We thus allow, as well as the predicate functions, predicates of the form

$$a = b \qquad \text{or} \qquad f(a, b) = g(f(c, a))$$

It is of course not necessary to introduce this as a primitive symbol; we can introduce a predicate function $E(x, y)$, say, and add to the theorem we wish to prove axioms expressing the required properties of E. However, as Wang remarks in his papers, it is more satisfactory to include = in our formalism to begin with. He, in fact, does so and the programs he describes allow = as a primitive symbol (thus complicating the algorithms). The other writers do not allow this; for example Davis, Logemann and Loveland (1962) in an example from group theory introduce a predicate $P(x, y, z)$ which is true if $x.\, y = z$ and false otherwise.

Apart from describing theorem-proving programs, the papers of Davis and Putnam (1960) and Davis (1963) contain easily read and more precise introductions to the terminology of the predicate calculus.

3. WHAT CAN WE PROVE?

What do we ask of a theorem-proving program? Ideally we require a program that takes as input some logic expression, makes sure it is sensible logic (i.e. is a wff; this is a mechanical operation and a useful exercise for a list processing language), and then tests it for a theorem finally printing the answer:—"Yes! this is a theorem" or "No! this is not a theorem".

Logicians have long since proved that this is impossible for the predicate calculus, that is there can be no algorithm which takes as input any wff of the predicate calculus and in all cases terminates with a decision as to whether the wff is a theorem or not. The predicate calculus is *undecidable*. However the propositional calculus is decidable, e.g. by the method of truth tables, and such programs have been produced.

Undecidability of a discipline is of itself no practical barrier to proving theorems in that discipline, although it does act as a warning against attempting too much in a theorem-proving program. Even if we are working in an undecidable discipline we can hope to devise procedures for looking for proofs that work in the majority of cases. Further, even if we know our discipline is decidable this is no guarantee that we can find a practically usable method for testing whether a given wff is a theorem or not.

Since we certainly cannot restrict our theorem-proving to decidable calculi what can we attempt? Friedman (1963a) states that there are three kinds of mechanical theorem-provers, which we proceed to discuss.

(*i*) *Solvable Subcase*

We can produce a program which only accepts as input wffs from some solvable subcase of the predicate calculus. One such subcase is the propositional calculus. Suppose our wff has all its quantifiers at the front (i.e. is in prenex normal form, and it is very easy to reduce any wff to this form). Then if it has all its universal quantifiers before all its existential quantifiers, or if it has only one existential quantifier, or if it has only two existential quantifiers and they have no universal quantifier between them, it is decidable.

Several such decidable subcases of the predicate calculus are listed, and the fact that they are decidable proved, in the book of Ackermann (1962). These proofs can be directly converted into theorem-proving programs for these cases (but will not be very efficient as this book is only concerned with showing the cases are decidable, i.e. he produces an algorithm for deciding if a wff, of the special form, is a theorem but is not concerned in any way with the efficiency of the algorithm).

(*ii*) *Proof Procedure*

Programs in this class take as input any wff of the predicate calculus. If the input is a theorem the program will terminate and say so. If the input is not a theorem the program (except in some special cases) will never stop.

The existence of algorithms of this kind is easily seen. The predicate calculus has a finite number of axioms and a finite number of deduction rules. One can, therefore, enumerate all possible proofs, e.g. first those with no steps, i.e. the original axioms, then those with one step, then with two steps and so on. (This argument is a little loose, e.g. in regard to the application of the rule for the substitution of one variable for another, but it can be proved that all possible proofs can indeed be enumerated.) Thus it is possible to have a program which lists all proofs, in order, and tests for each proof if the last wff is the same as the wff being tested for a theorem. Such a program (the so-called British Museum algorithm) will be a proof procedure; given as input a theorem it will eventually come to the proof of that theorem, but given as input a non-theorem it will never terminate. This particular proof procedure is, of course, useless even for simple theorems, but many of the references at the end of this chapter, as we shall see in section 6, are devoted to producing useful proof procedure programs.

(*iii*) *Semi-decision Procedure*

Programs in this class always terminate. On termination, the program will say one of three things:—"Yes! this is a theorem", "No! this is not a theorem" or "I don't know". It is intended that this should be intrinsic to the method

used rather than be taken literally. Thus it is trivially possible to change a particular program from one class to another. The addition of a 'stop after ten minutes' instruction to a program of type (ii) makes it type (iii), or the restriction of the input to a type (iii) program to the subclass that gives a yes or no answer (if that is known) makes it a type (i) program.

There is another type of program that does not fit readily into such a classification, and that is the heuristic program which is described in the next section. Strictly speaking they are semi-decision procedures, but to call them such would be misusing the phrase.

4. HEURISTIC METHODS

The first computer program to prove theorems was the Logic Theorist (LT) program of Newell, Shaw and Simon (1957) which proved theorems in the propositional calculus. They used a syntactic approach regarding the proving of a theorem as a search process, a search for the proof of the given theorem as a series of deductions. This search was guided to a large extent by the form of the input wff, and this guidance was at a pattern recognition level rather than a logic level, i.e. it made use of properties of the wff such as what its main connective was, how many and what symbols were used, and whether it was 'similar' in form to a previously proved theorem or axiom, rather than making use of the logic properties of the wff.

The program incorporated several heuristic devices. Heuristic methods are rules that, with relation to some specific problem-solving task, are likely to work in a large proportion of cases but are not guaranteed to do so. Thus there are theorems of the propositional calculus which LT, in its present form, cannot prove.

The main heuristic technique of LT was one of working backwards. Given a wff T, to try to prove it is a theorem LT looks at T and says, for example, "I can prove T if I can prove U or V or W (because T can be deduced from these by a deduction rule)". Now some more heuristics must be brought into play. We can use heuristics, based on the form of the expressions U, V and W to select which is likely to be the easiest to prove and then try and produce such a proof. Also, heuristics can be used to reject any of U, V or W that are not likely to be theorems (there is no guarantee that they are, or that this "strong non-provability" test will not reject as a theorem an expression that in fact is a theorem).

The working of LT was based on four "methods". The first of these was substitution. This method tries to prove the theorem by a series of substitutions from an axiom, or a previously proved theorem. All of these were tried but those unlikely to match the given theorem were rejected by a "similarity" heuristic involving much less work than the full 'matching'

algorithm. If substitution fails LT tries one of three other methods—"detachment", "forward chaining" and "backward chaining". Detachment states that, if we wish to prove T and if $A \rightarrow T$ is an axiom or previous theorem then try to prove A. Forward chaining states that if T is of the form $A \rightarrow B$, and if $A \rightarrow C$ is already known, then try to prove $C \rightarrow B$; backward chaining states that if T is of the form $A \rightarrow B$, and if $C \rightarrow B$ is already known, then try to prove $A \rightarrow C$.

The general operation of these four methods was controlled by a master routine which first tried substitution (the only method that actually produces a solution): if that did not work the various other methods were tried. Every time one of these produced a sub-problem the substitution method was tried on it; if this failed it was added to the subproblem list. This list was then scanned to find the "easiest" subproblem and a method, one not yet tried on this particular subproblem, was selected to use on it.

Wang (1960a) has criticised LT on the grounds that there are mechanized proof methods for the propositional calculus which are much more efficient than LT, and which ultimately find a proof of any theorem or reject a non-theorem. This is certainly true, and the performance of LT as a propositional calculus theorem-prover is unimpressive. However the performance of LT is impressive, if it is regarded as a heuristic attempt at the solution of a complex problem, that of finding a proof of a theorem in the previously defined syntactic sense, without making resource to more sophisticated metatheorems of logic. It does succeed in greatly cutting down the primitive search procedure of the British Museum algorithm.

Newell, Shaw and Simon used LT to explore the effectiveness of the various heuristics employed. It was the first attempt to introduce heuristic ideas into computer programs and served as an extremely useful specific complex problem to use in the investigation of such methods. The ideas of LT have since been expanded and heuristic methods applied to a variety of problems for whose solution we know no effective mechanical solution mechanisms. The LT program itself has served as a useful introduction to these techniques and has now been fully documented by Stefferud (1963). The program of Slagle (1961) to perform analytic integration shows how the ideas of LT can be used in other problem fields, as Slagle's executive organization is essentially the same as LT's.

The ideas of LT were further developed (see Newell and Simon 1961) into the "General Problem Solver". This program originated with an attempt to separate out the problem-solving capabilities of LT from the particular subject matter, symbolic logic. Heuristic programs, such as LT and GPS, besides indicating possibly useful mechanisms for solving complex problems in machines, also have important psychological implications. This line of research, though very interesting and important, is taking us outside the

scope of this particular chapter: readers interested should first consult Minsky (1961) and the book "Computers and Thought" edited by Feigenbaum and Feldman (1963).

Thus LT served an extremely important purpose as an introduction to and a test bed for heuristics, but it is not the way to prove theorems in the propositional calculus. However heuristic methods will be needed in theorem-proving, but applied at a higher level. Minsky (1961) puts it this way: ". . . it seems clear that a program to solve real mathematical problems will have to combine the mathematical sophistication of Wang with the heuristic sophistication of Newell, Shaw and Simon". To this should also be added a need for a greater sophistication in the inter-communication of man and machine.

Another heuristic theorem-proving program is the Geometry-theorem-proving machine of Gelernter (1959) (see also Gelernter and Rochester 1958). The powerful heuristic used here is the suggestive power of a diagram. Given a theorem of Geometry the program draws a diagram (actually it sets up coordinates for the points involved and uses analytical geometry techniques, but the program would not be altered in any essential way if a diagram was actually drawn and measurements made). With the aid of this diagram plausible subproblems can be set up. Thus if, by measurement in the diagram, two lines seem parallel or two segments equal the program can try to prove this fact. The diagram is in no way part of the proof (for example two equal segments in the diagram might be equal only because a special case has been drawn), but it is very suggestive of lines of attack. It is also useful in suggesting constructions, e.g. we can have a heuristic: if there is an axis of symmetry draw it in. The diagram heuristic is really "assume that what happens in a special case also happens in the general case" and is obviously useful in some fields but useless in others. Gelernter, Hansen and Loveland (1960) went on to consider the effect of the various heuristics and to consider the implications of their work.

5. HERBRAND'S THEOREM

All the programs so far produced for proving theorems in the predicate calculus depend on some form of Herbrand's theorem, and we discuss this first before describing the programs themselves. This theorem goes back to a dissertation of Herbrand (1930), and an account of the relevant parts of this dissertation is given by Wang (1963b). Versions of the theorem sufficient for their needs are given in Davis and Putnam (1960) and Davis (1963). In this section we mainly follow the exposition of the last two papers, except that there, if T is the theorem to be proved, they operate upon $\sim T$ in order to prove its inconsistency, whereas we will operate on T directly in order to

prove its validity (i.e. that it is a theorem). We assume that as a preliminary step T has been brought to prenex normal form.

Herbrand's theorem, in the form we are using it here, gives a way of mechanically producing from T, if T is a theorem of the predicate calculus, a theorem U of the propositional calculus. More specifically, from T we can derive a sequence of wffs of the propositional calculus

$$U_1, U_2, U_3, \ldots,$$

with the property that if T is a theorem then there is an N such that

$$U_1 \vee U_2 \vee U_3 \ldots \vee U_N$$

is a theorem of the propositional calculus.

This can obviously be used as a proof procedure by successively generating the U's and testing $U_1 \vee \ldots \vee U_S$ for a theorem; if it is then T was a theorem, and if it is not then we generate U_{S+1} and test again. This procedure will obviously terminate if T was a theorem and not otherwise, except in the special case of the U_S sequence being finite. This latter will occur only if T had no functions and also no universal quantifiers following an existential quantifier, a decidable subcase of the predicate calculus.

The production of the U_S sequence from T can be thought of in two steps.

Step 1. *Eliminate the Universal Quantifiers*

Replace the variable of every universal quantifier wherever it occurs by a new function symbol, the function to have as arguments the variables of any existential quantifier occurring before the universal quantifier.

Thus if T had the form

$$(Ex)(y)(Ez)(t)R(x, y, z, t) \tag{5.1}$$

where $R(x, y, z, t)$ is a wff of the predicate calculus without quantifiers and having $x\ y\ z\ t$ as free variables, then we would replace T by

$$(Ex)(Ez)R(x, f(x), z, g(x, z)) \tag{5.2}$$

where we assume R does not already have f and g as function symbols.

It can now be proved that (5.1) is a theorem if and only if (5.2) is a theorem. This proof is fairly simple using a semantic approach and remembering that (5.2) is a theorem means that (5.2) is a theorem for *all* possible functions f and g. Note that we are not saying that (5.1) and (5.2) are logically equivalent, i.e. that they take the same truth values with the same Universe and the same interpretations of the predicate functions. It is not true that [(5.1) $\leftrightarrow$ (5.2)] is a theorem, only that [(5.1) is a theorem $\leftrightarrow$ (5.2) is a theorem].

Step 2. *Produce the Sequence* $U_1, U_2, U_3, \ldots$ *by Instantiation over the Herbrand Universe*

Saying (5.2) is a theorem means that it takes the value 'true' for all assignments of the functions, the predicates and the free variables over all universes

of objects. What elements will be in one of these universes? It will certainly contain objects corresponding to the free variables a, b, c . . . occurring in (5.2) and also objects corresponding to $f(a)$, $f(b)$, $g(a)$, $g(a, b)$, $g(f(a), b)$ etc., where we compound the functions in any way we like. Of course, not all these objects in a particular universe with particular definitions of the functions will be different.

We now consider a, b, $f(a)$, $g(f(a), b)$ etc. as objects themselves, and define the Herbrand Universe to consist of all possible such objects. More exactly we define H_0 to be the set of free variables occurring in the wff (or just the single free variable a if none occur) and then define H_{r+1} as the set of all expressions of the form

$$f(K_1, K_2, \ldots, K_n)$$

where f is a function with n arguments occurring in the wff after step 1 has been performed, and $K_1, \ldots, K_n$ are each members of one of the sets H_0, $H_1, \ldots, H_r$ with at least one a member of H_r.

Thus, in the present example a is in H_0; $f(a)$ and $g(a, a)$ in H_1; $f(f(a))$, $g(f(a), a)$, $g(a, f(a))$, $g(f(a), f(a))$ etc. in H_2 and so on. We can define the members of H_r as being compound constants of level r; r indicates the maximum depth of function nesting.

We define the Herbrand Universe H to be the union of all the sets H_0, $H_1, H_2 \ldots$.

Unless there are no function symbols H will have an infinite number of elements, but it is easy to enumerate them, e.g. as each H_r has only a finite number of elements, by enumerating first the members of H_0, then of H_1, then of H_2, and so on.

Now consider (5.2) over the Herbrand Universe; it states that there are elements of H (x and z) such that $R(x, f(x), z, g(x, z))$ is true. Suppose we enumerate all possible pairs of elements of H, and since we can enumerate H this can easily be done. Suppose the S-th pair is

$$x_s, z_s$$

and let

$$U_s \text{ denote } R(x_s, f(x_s), z_s, g(x_s, z_s))$$

Stating that (5.2) is true in the Herbrand Universe means that for any assignment of the predicate functions occurring in R there is a pair x_n, z_n such that U_n is true, and different assignments to the predicate functions will produce different n. This suggests, and it can be proved, that (5.2) is a theorem if and only if

$$U_1 \vee U_2 \vee U_3 \ldots \vee U_s \ldots \tag{5.3}$$

is a theorem.

This is an infinite disjunction, but it is known that an infinite disjunction is a theorem if and only if some finite subset of it is, i.e. there is a N such that

$$U_1 \vee U_2 \vee U_3 \ldots \vee U_N \tag{5.4}$$

is a theorem.

Noting that each U_i is in fact a wff of the propositional calculus (the predicate functions only have particular constants from the Herbrand Universe as their arguments and so are statements of the propositional calculus), we have produced our desired sequence.

Before giving a concrete example let us note a simplification. Assuming that T, after step 1, is of the form

$$(Ex_1)(Ex_2) \ldots (Ex_n)R(x_1 \ldots x_n)$$

where $R(x_1 \ldots x_n)$ is a wff without quantifiers, then each U_s is obtained from $R(x_1 \ldots x_n)$ by a particular substitution of constants from H for the variables $x_1 \ldots x_n$. We say that U_s is an *instance* of $R(x_1 \ldots x_n)$. Now suppose we first transform $R(x_1 \ldots x_n)$ into *disjunctive normal form*. This may be defined as follows.

We define an *atomic formula* to be a predicate function (of any number of variables and possibly having functions compounded in any way as arguments) e.g. $F(x, f(g(a, y)), g(x, y))$. We then define a *literal* to be an atomic formula, or the negation of an atomic formula, and a *clause* to be the *conjunction* (i.e. the logical 'and') of any number of literals. Finally we get a disjunctive normal form by taking the *disjunction* (i.e. the logical "or") of any number of clauses. Any wff may be put into disjunctive normal form by first removing logical connectives $\rightarrow$ and $\leftrightarrow$ by use of their definitions; then driving negation signs as far 'inwards' as possible by such rules as replacing $\sim(A \wedge B)$ with $\sim A \vee \sim B$; then driving all conjunctions inwards by replacing $(A \vee B) \wedge C$ with $(A \wedge C) \vee (B \wedge C)$. An example is given below.

If this process is carried out so that $R(x_1, \ldots x_n)$ is in disjunctive normal form, say

$$R(x_1, \ldots x_n) = C_1 \vee C_2 \vee \ldots \vee C_k \tag{5.5}$$

where each C_i is the conjunction of literals, then each U_s of equation (5.3) will itself be a disjunction. Then in picking the finite subset of (5.3) that is a theorem we can select from the constituent clauses of each U_s rather than taking each complete U_s as in (5.4).

So, finally, we have our version of Herbrand's Theorem:—

Perform step 1 as indicated and write $R(x_1 \ldots x_n)$ in disjunctive normal form as equation (5.5). Then there is an N such that

$$Q_1 \vee Q_2 \vee Q_3 \ldots \vee Q_N \tag{5.6}$$

is a theorem, where each Q_i is obtained from some C_j by a particular substitution of constants from the Herbrand Universe for the variables occurring in C_j, i.e. Q_i is an instance of C_j.

Example

Let us take the fourth example of (2.2), a theorem of the predicate calculus.

$$(Ex)(Ey)(z)((F(x, y) \rightarrow (F(y, z) \wedge F(z, z))) \wedge ((F(x, y) \wedge G(x, y)) \rightarrow (G(x, z) \wedge G(z, z)))) \quad (5.7)$$

Replacing $\rightarrow$ by its definition and transforming to disjunctive normal form (it obviously does not matter whether the transformation to disjunctive normal form is done before or after step 1) we get

$$(Ex)(Ey)(z)[\sim F(x, y) \vee F(y, z)F(z, z)\sim G(x, y) \vee F(y, z)F(z, z)G(x, z)G(z, z)]$$

where we have used concatenation to denote conjunction, i.e. AB stands for $A \wedge B$ and we have also replaced $A \vee (A \wedge B)$ by its logical equivalent A.

To eliminate the universal quantifier in this example we have to replace z wherever it occurs by $f(x, y)$, thus obtaining

$$(Ex)(Ey)[\sim F(x, y) \vee F(y, f(x, y))F(f(x, y), f(x, y))\sim G(x, y) \vee F(y, f(x, y))F(f(x, y), f(x, y))G(x, f(x, y))G(f(x, y), f(x, y))] \quad (5.8)$$

In this case we have

C_1 is $\sim F(x, y)$

C_2 is $F(y, f(x, y))F(f(x, y), f(x, y))\sim G(x, y)$

C_3 is $F(y, f(x, y))F(f(x, y), f(x, y))G(x, f(x, y))G(f(x, y), f(x, y))$

and the successive levels of the Herbrand Universe are given by

H_0 contains just a (the single constant added as (5.8) has no free variable)

H_1 contains just $f(a, a)$

H_2 contains just $f(a, f(a, a))$; $f(f(a, a), a)$ and $f(f(a, a), f(a, a))$ and so on.

As (5.7) was a theorem we must be able to find a finite set of instances of C_1, C_2, C_3 (by substitution for x and y from H, the union of all H_i) such that the disjunction of these instances is a theorem of the propositional calculus. Let us abbreviate $f(a, a)$ to 1; $f(a, f(a, a))$ to 2 and $f(f(a, a), f(a, a))$ to 3, and use subscripts on F and G rather than arguments, so that for example

F_{12} stands for $F(f(a, a), f(a, f(a, a)))$.

Then a theorem of the propositional calculus is

$$\sim F_{11} \vee \sim F_{a1} \vee \sim F_{13} \vee \sim F_{12} \vee \sim F_{33} \vee \sim F_{22} \vee$$
$$\vee F_{13}F_{33}\sim G_{11} \vee F_{12}F_{22}\sim G_{a1} \vee F_{a1}F_{11}G_{a1}G_{11}$$

and each clause is an instance of one of C_1, C_2 or C_3.

Quine (1955) showed that Herbrand's Theorem, in the form we have stated it, could be used as a proof procedure. He also pointed out the advantages of not reducing the wff to prenex normal form, and showed how this could be done. He did not give any practical methods for finding the particular instances required. In the next section we shall show how this problem has been tackled.

6. THEOREM-PROVING PROGRAMS

Herbrand's Theorem tells us that there is a theorem of the propositional calculus of the form (5.4) or (5.6) corresponding to any theorem of the predicate calculus, but how do we find it? The obvious method is to use (5.4), that is to successively generate U_i and periodically test to see whether we have a tautology of the form (5.4).

Thus first test $U_1 \vee U_2 \ldots \vee U_N$ for a tautology; if it is not
then test $U_1 \vee U_2 \ldots \vee U_{2N}$ for a tautology; if it is not
then test $U_1 \vee U_2 \ldots \vee U_{3N}$ for a tautology, and so on, where N is some given integer, say 5 or 10.

Gilmore (1959, 1960) used this method. His method of testing (5.4) for a theorem was to transform (5.4) into conjunctive normal form (this is as a disjunctive normal form previously defined except that a wff in conjunctive normal form is the logical "and" of a number of clauses, each clause being the logical "or" of a number of literals). There is then a very simple test of a wff in conjunctive normal form to find out whether it is a theorem or not. It is a theorem if and only if every separate clause is, and each clause is a theorem if and only if it contains some atomic formula p and its negative $\sim p$.

Gilmore's method worked on simple examples, but failed to reach its conclusion for the example of the last section in 20 mins. running time (on an IBM 704). The difficulty is that the transformation to conjunctive normal form causes a very great increase in the number of clauses, unless the wff is of a special form such that there is a great deal of cancellation. The amount of work increases exponentially with N, the number of clauses in the disjunctive normal form.

Davis and Putnam (1960) pointed this out, and also the fact that the use of truth tables would be just as disastrous since it would involve the work increasing exponentially with the number of variables in the wff of the propositional calculus. They give an alternative method in which the work only

increases linearly with the number of variables. Their method consists of a fast technique for testing a wff of the propositional calculus which is in disjunctive normal form, to find out whether or not it is a theorem. (They took the viewpoint of proving $\sim T$ inconsistent rather than T a theorem and so the technique they actually gave is, in fact, the dual of this, i.e. a method for testing a wff in conjunctive normal form as to whether it is consistent or not.)

This testing of a wff in disjunctive normal form consisted of a series of eliminations of the variables. They give three rules for the transformation of a wff into another wff with one variable eliminated, this second wff being a theorem if and only if the first wff was a theorem (or with their viewpoint is consistent if and only if the first is). These three rules are: first, a rule for the elimination of any variable that occurs as the only literal of some clause; second, a rule for the elimination of any variable that either only occurs affirmatively or only occurs negatively; and third, a rule for eliminating any variable. By an affirmative occurrence of a variable we mean an occurrence not preceded by a negation sign and a negative occurrence is one that is so preceded.

This process terminates in at most $2(R - 1)$ steps where R is the number of variables. Since the successive generations of formulas obtained by using Herbrand's theorem naturally occur in disjunctive normal form the use of this method avoids the disastrous conversions to conjunctive normal form of Gilmore. The fast test of wffs in disjunctive normal form was discovered independently by Dunham, Fridsal and Sward (1959).

The method of Davis and Putnam works by hand simulation on the example of the last section. The method has been implemented as a computer program (see Davis, Logemann and Loveland 1962), where some results are mentioned and the program's failure on a simple example from group theory analysed.

Prawitz, Prawitz and Voghera (1960) describe a proof procedure which, although not produced directly from Herbrand's Theorem, is equivalent to Gilmore's and Davis and Putnam's procedures in the production of the successive quantifier free lines, but is inferior to the method of Davis and Putnam in its propositional calculus part. However it has the advantage of not first requiring the wff to be reduced to prenex normal form.

The method starts by assuming the given wff T is not a theorem, i.e. it is possible to assign the value "false" to it. The consequences of this are then followed by looking at the main connective of T; for example if T is of the form $F \rightarrow G$ we assign the value 'true' to F and "false" to G, as this is the only way $F \rightarrow G$ can be assigned the value "false". A list of fourteen actions are specified according to whether the assignment is to 'true" or "false" and according to whether the wff starts with a universal or existential

quantifier or, if it does not start with a quantifier, according to which of the five logical connectives is its main connective. Some of these actions cause a splitting to form a tree structure, for example, assigning "true" to $F \vee G$ we have either to assign "true" to F or "true" to G. The actions corresponding to the quantifiers involve the introduction of new constants, or the substitution of all constants previously introduced for variables in the wff, corresponding to generation of the successive quantifier free line of the last section. If all branches of the tree end with a contradiction (by making the opposite truth value assignment to some formula previously assigned a value) then the original assignment of "false" was impossible and so the given wff was a theorem. If the given wff was not a theorem usually the branches of the tree will expand indefinitely and the program not terminate.

Prawitz, Prawitz and Voghera give a complete description of their program and five simple examples of theorems proved by it.

J. A. Robinson (1963) has carried out an investigation into the rate of growth of the various levels of the Herbrand Universe. To show how rapid that can be he gives an example from group theory: the existence of a right identity element in any algebra closed under a binary associative operation having left and right solutions x and y for all equations $x.a = b$ and $a.y = b$, where a and b are in the algebra (a proof of this is normally written down in about four lines). He shows that substitutions from H_0 generate 6 lines, from H_1 31330 lines, from H_2 more than 5.10^{11} lines. The propositional calculus theorem is the disjunction of only four of these lines, but one of these lines is obtained by a substitution from H_3 and so any program which enumerates H by first enumerating H_0, then H_1 and so on (J. A. Robinson calls this a level saturation procedure) would have to generate at least 5.10^{11} lines and probably a very much larger number.

The particular constants of H substituted for variables in this example are just

$$a,\ h(a, a),\ k(h(a, a)),\ g(a, k(h(a, a))) \tag{6.1}$$

and he calls this a proof set. J. A. Robinson suggests that the spotting of this proof set is the creative part of theorem-proving and has produced a program which has to be supplied with a guess at the proof set. The program then only generates lines obtained by instantiation with members of this set, and if the guess is right the program produces a proof of the theorem. If the guess is wrong the program says so, and one can try another guess at the proof set. The members of the proof set correspond to substitutions that have to be made in the course of the proof, and so it is not unreasonable to expect the user to be able to guess at these. An example is given of a proof of the theorem "the square root of a prime number is irrational".

J. A. Robinson also points out that all the members of the proof set can be obtained as constituent parts of a single compound constant; in the example

of (6.1) the first three constants are parts of the fourth. In fact it is true that if a proof set contains a particular compound constant it must also contain all its constituents. He therefore suggests the following procedure. First, try H_0 as a proof set; if it is not take each member of H_1 and use it and all its constituents together with the remaining members of H_0 as a proof set. Then do the same with H_2, H_3, and so on. In this way we can hope to get deeper into the Herbrand Universe and yet not have too many distinct constants at one time. This implies that the N of (5.4) or (5.6) does not get large, the particular clauses used changing rather than keeping a particular Q_i for ever once it appears, but of course the number of members of H_i increases very rapidly with i and we still have to try each member of each level.

Prawitz (1960) describes a proof procedure in which the particular quantifier free lines needed in (5.4) to produce a theorem can be calculated directly from the given theorem, so that a disjunction with minimum N is produced. The proof procedures of Gilmore and of Davis and Putnam both produce a disjunction that is a theorem, but almost all the clauses of the disjunction are irrelevant; they can be omitted and still the disjunction is a theorem. The method of Prawitz is extremely attractive in that it produces directly just the necessary clauses, so that in the group theory example the four relevant clauses and only these are produced. However, Davis (1963) points out that except in very special cases the Prawitz procedure will not work because of the explosive nature of the various transformations from one normal form to another. J. A. Robinson (1963) states that the Prawitz procedure works on the group theory example but not on the example used in section 5.

The essential idea of the method is the use of dummies. We are looking for a disjunction (5.4) where each U_i is obtained by substitution of constants for variables. Instead of substituting actual constants (from the Herbrand Universe) for the variables, dummy symbols are used, and then an automatic procedure is invoked to find the values of these dummies, if any, to make a theorem. Thus dummies are placed for the variables in R forming U_1, and then the values are sought for them to make a theorem. If none are found then different dummy symbols are used to form U_2, and values of these and of the previous dummies are sought to make $U_1 \vee U_2$ a theorem. This procedure is continued until values of the dummies can be found to make a theorem, and we then have our disjunction of minimum length.

The method of finding the values of the dummies is roughly as follows. First, convert the disjunctive normal form to conjunctive normal form. As we have pointed out before the conditions for a wff of this form to be a theorem are easily stated, and these immediately lead to a set of relations among the dummies that have to be satisfied. It turns out that this set of relations that has to be satisfied is a wff of the propositional calculus (with the equality symbol) and is in conjunctive normal form. Now we have to

find values of the dummies to satisfy this set, i.e. the set must be consistent. There is a simple condition for consistency of a wff in disjunctive normal form (this is the dual of the test for a theorem of a wff in conjunctive normal form). So the set of relations is converted to disjunctive normal form, and a consistent set of values for the dummies can be seen immediately. It is, of course, these two conversions to normal form that make the method unsuitable in practice.

Prawitz does not use functions in his procedure and does not directly carry out the first step of our section 5. Instead, constant symbols are substituted directly for the universally quantified variables. This leads to a number of conditions on possible values for the dummy symbols. The possibility of two constants being equal must also be allowed for and this leads to further restrictions on the values of the dummy symbols. All these restrictions must be met in the values assigned to the dummies.

Kanger (1963) describes a method using dummies very similar to the method of Prawitz, but does not give any method for directly calculating the values of the dummies. He allows the inclusion of the equality sign in the predicate calculus and Prawitz (1960, page 123), quotes a similar proof method of Kanger as also including the addition and multiplication signs.

Davis (1963) gives an elementary and clear account of what is meant by a theorem and of Herbrand's theorem; he then summarizes Gilmore's, Davis and Putnam's, and Prawitz's programs. He ends by describing a new kind of procedure which, in his words, seeks to combine the virtues of the Prawitz procedure and those of the Davis-Putnam procedure.

Let us assume that in some wff of the propositional calculus which is in disjunctive normal form the statement p occurs but the combination $\sim p$ never occurs (or vice versa). Then it is easy to prove that the wff is a theorem if and only if the wff obtained from the given wff by deleting all clauses containing p (or $\sim p$) is a theorem. This is equivalent to the second elimination rule of the Davis and Putnam procedure previously mentioned. By repeating this process we must eventually arrive at a wff that is a selection of the clauses of the original wff, and which has the property that every statement p that occurs in it appears in some clause as p and in some other clause as $\sim p$. Such a disjunctive normal form is called a *linked disjunct.*

There must, therefore, be some subset of the $Q_1 \ldots Q_N$ of (5.6) that forms a linked disjunct and which is a theorem if and only if (5.6) is. Davis suggests as a means of not including large numbers of irrelevant clauses that we devise a procedure that only generates linked disjuncts. Every time some instantiation over the Herbrand Universe is made to produce a clause Q_i we look at the constituent literals of the clause. If there is one, $F(f(a), b)$, say, whose mate $\sim F(f(a), b)$ has not yet occurred in any clause so far produced, then we instantiate some other clause with some particular set of constants

so that this mate is produced. When we have a linked disjunct we test for a theorem. Davis gives some heuristic rules for carrying out this procedure; it is not clear how effective they will be. He ends by stating that the proof procedure is currently being programmed.

Dunham and North (1963) independently note the theorem on linked disjuncts and suggest it be used in two ways, first as a basis for looking ahead to select relevant clauses and second in order to trim down already generated expressions. They suggest practical methods of bookkeeping in computers for the testing of very large expressions (large enough to warrant magnetic tape storage) as generated by Herbrand expansions.

J. A. Robinson (1964) sets out a new formulation of the predicate calculus, one which is specially adapted to the design of theorem-proving programs. It is based on a single rule of inference, the resolution principle, which is considerably more complicated than those previously used in logic, but one which is specially suited for the development of mechanical proof finders and of useful principles for cutting down the search time for proofs. He briefly discusses some such principles in the final section of his paper, and promises a sequel in which the theoretical framework will be used as a basis for setting out search principles and for the design of theorem-proving programs.

Wang (1960a) gives various remarks on theorem-proving in general, and also reports on three actual computer programs. The first of these is a decision procedure for the propositional calculus, i.e. it is a program which tests whether a given wff of the propositional calculus is a theorem or not. Wang uses the notion of a sequent; this can be regarded as a generalisation of the logical connective "implies" ($\rightarrow$), a generalization that it is natural to make and one which enables the easier expression of "real" theorems.

If $\phi_1, \phi_2, \ldots, \phi_n$ and $\psi_1, \psi_2, \ldots, \psi_n$

are wffs (either of the propositional calculus or of the predicate calculus) then

$$\phi_1, \phi_2, \ldots, \phi_m \rightarrow \psi_1, \psi_2 \ldots, \psi_n \tag{6.2}$$

is a sequent which can be thought of as meaning that if all the wffs on the left are true at least one on the right is true, i.e. the sequent is true if and only if

$$(\phi_1 \wedge \phi_2 \ldots \wedge \phi_m) \rightarrow (\psi_1 \vee \psi_2 \ldots \vee \psi_n)$$

is a theorem of the propositional calculus.

Prawitz (1960) also works with sequents, although we did not explain his work in those terms.

Wang then gives rules for transforming a sequent of the propositional calculus into another sequent with one less logical connective, for example, in (6.2) if ϕ_1 was of the form $A \wedge B$ the ϕ_1 could be replaced by A, B. Some of the rules produce a splitting, two sequents being generated from one and

we thus have a tree structure; at the ends of all branches of the tree are sequents without logical connectives If all these sequents are theorems then the original sequent is a theorem, and these sequents may very easily be tested as they are without logical connectives. Such a sequent is a theorem if, and only if, at least one of the wffs on the left is the same as one of the wffs on the right. The method (expressed without sequents) is precisely a method of reduction to conjunctive normal form which, as we have seen before, can easily be tested for a theorem.

Wang goes on to specify additional rules for extending the program to include the equality sign in the propositional calculus. There is no change to the sequent transformation rules but the test of the final sequents must allow for substitution of equal terms. He did not produce an actual program for this, but the method was included as a part of his third program.

Wang's second program was an attempt to mechanically produce "interesting" theorems of the propositional calculus. The program generated sequents which were theorems, each sequent containing at most two formulas and each formula containing at most six symbols (i.e. connectives and statements) of which at most three were statements. Rules were introduced for rejecting "trivial" theorems, but these proved to be not selective enough. There were still far too many theorems left to consider that the program has "discovered" any interesting theorems.

Wang then went on to consider the predicate calculus, including the equals sign but excluding functions. He specified three procedures for the AE case, and a wff of the predicate calculus comes under this subcase if it can be converted to an equivalent wff in prenex normal form in which all universal quantifiers precede all existential quantifiers. This, as we have mentioned before, is a decidable subcase of the predicate calculus. The Herbrand Universe in this case is finite and in the methods of Gilmore, or Davis and Putnam, only a finite number of quantifier free lines can be generated. The Wang procedures for the AE case differ in the particular transformations made and in the order various substitutions are made, but all at some time make substitutions for the variables that effectively generate these lines.

One of the procedures was coded as a computer program; this program was able to prove almost all the theorems of Principia Mathematica that were in the restricted predicate calculus with equality, and it could have proved them all with some minor additions at the front. It could only do this, of course, because of the absence of any theorems not within the AE subcase.

Wang finally considers the full predicate calculus and suggests methods that are equivalent to generating all the quantifier free lines and periodically testing their disjunction for a theorem. He comments that this method can soon become cumbersome and suggests that strategies are necessary for choosing the right lines to form the disjunctions. This, he states, is in part

related to making use of previously proved theorems, which it seems likely will be needed when we try to prove theorems in more advanced disciplines.

Wang (1960b) describes in outline a general method of theorem-proving which he calls pattern recognition. Wang (1961) gives the detailed theoretical considerations, whilst as an illustration of the general method the first paper gives a solution of the E_1A case (i.e. the wff is in prenex normal form with one existential followed by an indefinite number of universal quantifiers). In addition this paper contains details of a program which could form a basis for developing the method of pattern recognition. This program is an extension of the third program of Wang (1960a) and tries to split the given wff into a number of simpler wffs, some of which could well fall into a decidable subclass of the predicate calculus. The program will decide most wffs of the AE form (simple additions would enable it to decide them all), and it can also solve some problems not of the AE form. The program does not include any pattern recognition techniques, but is intended as a systematic preparation for the use of such techniques.

Rather than give the details of Wang's various pattern recognition methods we describe a similar method of Friedman.

Friedman (1963a) describes a semi-decision procedure for the predicate calculus. For most known decidable subcases of the predicate calculus the method is a decision procedure, i.e. it always terminates with a "yes" or "no" answer; for all other cases it is a semi-decision procedure, i.e. it will terminate and may give an answer or may not.

The procedure assumes the wff is in Skolem normal form and has no free variables. This means that all quantifiers must be first, i.e. the wff must be in prenex normal form, but further all existential quantifiers must precede all universal quantifiers. There are algorithms for converting any wff into a wff in Skolem normal form, such that both are theorems or both not. In general, however, there will be an increase in the number of quantifiers and in the number of predicate functions. Thus a wff in Skolem normal form must have the form

$$(Ey_1)(Ey_2)\ldots(Ey_m)(z_1)(z_2)\ldots(z_n)M(y_1\ldots y_m, z_1\ldots z_n)$$

where $M(y_1\ldots y_m, z_1\ldots z_n)$ is quantifier free and is called the matrix. The matrix will consist of predicate functions joined by logical connectives; the actual predicate functions with arguments that occur in the matrix Friedman calls its elementary parts. Thus the fourth example of (2.2) is already in Skolem normal form, the elementary parts being $F(x, y)$, $F(y, z)$, $F(z, z)$, $G(x, y)$ and $G(x, z)$.

If the wff is in Skolem normal form there is a simple method of allocating distinct integers to the distinct members of the Herbrand Universe and of generating the quantifier free lines. The general method can be seen by

looking at the case of two existentials preceding three universals. In this case we write down a series of lines as follows:

y_1	y_2	z_1	z_2	z_3
0	0	1	2	3
0	1	4	5	6
1	0	7	8	9
1	1	10	11	12
0	2	13	14	15
1	2	16	17	18
2	0	19	20	21
2	1	22	23	24

In general we head each column with a variable, and under the m existentially qualified variables we write all possible m-tuples of the natural numbers (commencing with zero). Under the n universally quantified variables we write the natural numbers in order starting with 1, proceeding across the first line, then the next, and so on. The numbers 0, 1, 2, ... correspond to the elements of the Herbrand Universe as defined in section 5, the s-th row of the table indicating what substitutions have to be made in the matrix for the variables at the head of the columns to obtain the s-th quantifier free line. This correspondence can be seen as follows. In step 1 of section 5 the variables z_1, z_2 and z_3 will be replaced by $f(x, y)$, $g(x, y)$ and $h(x, y)$ say. Now 0 will correspond to the one free variable introduced, and then obviously 1 corresponds to $f(0, 0)$, 2 to $g(0, 0)$, ..., 8 to $g(1, 0)$ which is $g(f(0, 0), 0)$, and so on.

In addition to columns for the variables we can provide columns for the elementary parts occurring in the matrix. Thus returning to the fourth example of (2.2), which has x and y as existential variables and z as a universal variable, we have a table, in which F_{xy} stands for $F(x, y)$, given by

x	y	z	F_{xy}	F_{yz}	F_{zz}	G_{xy}	G_{xz}	G_{zz}
0	0	1		A				
0	1	2	B					
1	0	3						
1	1	4						

(6.3)

...

Such a table Friedman calls a decision table, Wang a sequential table. Under the elementary parts truth values may be filled in. Thus at A we can place the truth value of F_{01}. Because of the construction of the table the same truth value must be placed at B and there are obviously many such restrictions if the infinite table is to be legitimately filled in with truth values.

Herbrand's theorem states, loosely speaking, that the original wff is a

theorem if and only if the infinite disjunction of quantifier free lines is a theorem. This implies that the original wff is not a theorem if and only if the infinite disjunction is not a theorem, and if a disjunction is not a theorem it must be possible, with some particular assignment of truth values to the members of the Herbrand Universe, to make all clauses in the disjunction false. Thus if the wff is not a theorem it is possible to legitimately fill in the above decision table in such a way that each row gives a value "false" to the matrix; conversely, if the table cannot be so filled the original wff was a theorem.

Note, however, that there are only a finite set of assignments of truth values to the elementary parts that make the matrix false. Thus in our example where the matrix is

$$(F_{xy} \rightarrow (F_{yz} \wedge F_{zz})) \wedge ((F_{xy} \wedge G_{xy}) \rightarrow (G_{xz} \wedge G_{zz}))$$

only a set of assignments can make the matrix false which are taken from a row of the table

F_{xy}	F_{yz}	F_{zz}	G_{xy}	G_{xz}	G_{zz}
true	false				
true		false			
true			true	false	
true			true		false

(6.4)

A blank indicates that any assignment may be made; thus the first line corresponds to 16 different sets of assignments, the second line to 8 new sets of assignments not included in the first 16, the third line to another 2 and the last line to 1 more, making 27 possible sets of assignments to the elementary parts. Such a set is called a falsifying set.

If the original wff was not a theorem then it must be possible to fill the infinite table (6.3), each row having one of the 27 falsifying sets (6.4) in such a way that the constraints of the table are satisfied. But it is easy to see that this is impossible, and so the wff must be a theorem. For suppose one of the 16 sets of the first line of (6.4) is used in some arbitrary row of table (6.3), say the one with $x = \alpha$, $y = \beta$ and $z = \gamma$. Then $F_{\beta\gamma}$ is false. But somewhere in (6.3) is a row with $x = \beta$ and $y = \gamma$ as x, y run all over pairs of integers. In this row, under F_{xy} which will be $F_{\beta\gamma}$, we must then put "false" for consistency. But no set in (6.4) has F_{xy} false and so it will be impossible to fill this row. Therefore none of the 16 sets of the first line of (6.4) can possibly be used in (6.3), since α, β, γ were arbitrary. Similar reasoning shows that none of the 27 sets of (6.4) can be used at all, and so it is impossible to fill the table (6.3). We have therefore shown that our example was a theorem.

Definite rules may be written down for eliminating falsifying sets from being usable to fill the decision table. Thus Wang gives a single rule for the case of only one enistential quantifier; Friedman gives three rules for the

case of two existential quantifiers and four rules for the general case. Some of these rules are of the form: if one of the falsifying sets has certain properties, and if there are not other sets with certain other properties, then the given set can be deleted as unusable in the decision table. Another form of rule causes a splitting into subproblems, for example a rule which shows that two particular sets may not be used together in the decision table leads to two subproblems, one without one of the sets and the other without the other.

These rules may be programmed; if as a result of continued applications of these rules all sets are eliminated from all subproblems then we can conclude that the original wff was a theorem. However the program may also terminate because no more rules are applicable, and then in general no conclusion may be drawn. Just because our particular rules are no longer applicable we cannot conclude that it is possible to fill in the decision table. In certain cases, however, this conclusion can be drawn. If the wff was of some special form (for example the cases just mentioned having only one or two existential quantifiers) then the rules for eliminating sets given by Wang or Friedman are exhaustive, in the sense that if there are any sets left then the original wff was not a theorem. The proof of this proceeds by showing how the infinite table may, in fact, be filled in if we have sets to which none of the rules are applicable.

Of course, in any particular case, by special reasoning we may be able to prove the impossibility of filling the decision table with the given falsifying sets. This reasoning will depend on the pattern of constraints in the decision table and on the falsifying sets, which is why such methods are called pattern recognition techniques.

Friedman (1963b) describes a computer program for the special case of her semi-decision procedure in which the wff is already in Skolem normal form, has only two existential quantifiers and all predicate functions have only two arguments. This is a case where the elimination rules are exhaustive, and so the program always terminates with a conclusion, theorem or not (given enough time of course). She states that some seventy examples have been run and gives examples of some of the more difficult ones, and ends by concluding that this method, where it is applicable, is much faster than previous theorem-proving programs, the increase of speed being intrinsic to the method.

7. REFERENCES

ACKERMANN, W. (1962) *Solvable Cases of the Decision Problem*, North-Holland Publishing Company, Amsterdam.

COOPER, W. S. (1964) Fact retrieval and deductive question-answering information retrieval systems, *J. ACM.* **11**, 117.

DAVIS, M. (1963) Eliminating the irrelevant from mechanical proofs, *Proc. Symposia in Applied Maths*,Vol. XV, A.M.S. (Ed. Metropolis *et al.*), p. 15.

DAVIS, M., LOGEMANN, G. and LOVELAND, D. (1962) A machine program for theorem proving, *Comm. ACM.* **5,** 394.

DAVIS, M. and PUTNAM, H. (1960) A computing procedure for quantification theory, *J. ACM.* **7,** 201.

DUNHAM, B., FRIDSAL, R. and SWARD, G. L. (1959) A non-heuristic program for proving elementary logical theorems, *Proc. of International Conference on Information Processing*, Paris, UNESCO, p. 282.

DUNHAM, B. and NORTH, J. H. (1963) Theorem testing by computer, *Proc. of Symposium on Mathematical Theory of Automata. Microwave Research Inst.*, Symposia Series, Vol. XII, Polytechnic Press, p. 173.

FEIGENBAUM, E. A. and FELDMAN, J. (Eds.) (1963) *Computers and Thought*, McGraw-Hill. This book includes the articles Newell, Shaw and Simon (1957), Newell and Simon (1961), Gelernter (1959), Gelernter, Hansen and Loveland (1960), Slagle (1961) and Minsky (1961).)

FRIEDMAN, J. (1963a) A semi-decision procedure for the functional calculus, *J. ACM.* **10,** 1.

FRIEDMAN, J. (1963b) A computer program for a solvable case of the decision problem, *J. ACM.* **10,** 348.

GELERNTER, H. (1959) Realization of a geometry-theorem proving machine, *Proc. of International Conference on Information Processing*, Paris, UNESCO, p. 273.

GELERNTER, H., HANSEN, J. R. and LOVELAND, D. W. (1960) Empirical explorations of the geometry-theorem proving machine, *Proc. of the Western Joint Computer Conference*, **17,** p. 143.

GELERNTER, H. and ROCHESTER, N. (1958) Intelligent behaviour in problem-solving machines, *I.B.M. Journal of Res. and Dev.* **2,** 336.

GILMORE, P. C. (1959) A program for the production from axioms of proofs for theorems derivable within the first order predicate calculus, *Proc. of International Conference on Information Processing*, Paris, UNESCO, p. 265

GILMORE, P. C. (1960) A proof method for quantification theory: its justification and realization, *I.B.M. Journal of Res. and Dev.* **4,** 28.

HERBRAND, J. (1930) Recherches sur la theorie de la demonstration, *Traveaux de la Société des Sciences de Varsovie*, No. 33.

KANGER, S. (1963) A simplified proof method for elementary logic. *Computer programming and formal systems* (Ed. Braffort, P. and Hirschberg, D.), North-Holland, p. 87.

MCCARTHY, J. (1962) Computer programs for checking mathematical proofs, *Proc. Symposia in Pure Mathematics*, Vol. V, *A.M.S.*, p. 219.

MINSKY, M. L. (1961) Steps towards artificial intelligence, *Proc. IRE* **49,** 8.

NEWELL, A., SHAW, J. C. and SIMON, H. A. (1957) Empirical explorations of the logic theory machine, a case study in Heuristic, *Proc. Western Joint Computer Conference*, 15, p. 218.

NEWELL, A. and SIMON, H. A. (1961) *GPS*, a Program that Simulates Human Thought, *Lernende Automaten Oldenbourg*, Munich.

PRAWITZ, D. (1960) An improved proof procedure, *Theoria* **26,** 102.

PRAWITZ, D., PRAWITZ, H. and VOGHERA, N. (1960) A mechanical proof procedure and its realization in an electronic computer, *J. ACM.* **7,** 102.

QUINE, W. V. O. (1955) A proof procedure for quantification theory, *J. of Symbolic Logic*, **20,** 141.

RAPHAEL, B. (1964) *SIR:* A Computer Program for Semantic Information Retrieval, *MAC-TR-2*, Massachusetts Institute of Technology; part fulfilment of Ph.D. thesis.

ROBINSON, A. (1960) On the mechanization of the theory of equations, *Bull. Res. Council of Israel*, **9F,** 47.

ROBINSON, A. (1963) A basis for the mechanization of the theory of equations, *Computer Programming and Formal Systems* (Ed. Braffort, P. and Hirschberg, D.), North-Holland, p. 95.

ROBINSON, J. A. (1963) Theorem-proving on the computer, *J. ACM.* **10,** 163.

ROBINSON, J. A. (1964) A machine oriented logic based on the resolution principle (to appear in *J. ACM*).

SLAGLE, J. R. (1961) A computer program for solving problems in freshman calculus, doctoral dissertation M.I.T., and *J. ACM*, **10**, 507.

STEFFERUD, E. (1963) The Logic Theory Machine: a Model Heuristic Program, *The Rand Corp. memorandum RM-3731-CC.*

WANG, H. (1960a) Toward mechanical mathematics, *I.B.M. Journal of Res. and Dev.* **4**, 2.

WANG, H. (1960b) Proving theorems by pattern recognition I, *Comm. ACM.* **3**, 220.

WANG, H. (1961) Proving theorems by pattern recognition II, *Bell System Technical Journal*, **40**, 1.

WANG, H. (1963a) The mechanisation of mathematical arguments, *Proc. Symposia in Applied Maths*, Vol. XV, *A.M.S.* (Ed. Metropolis *et al.*), p. 31.

WANG, H. (1963b) Mechanical mathematics and inferential analysis, *Computer Programming and Formal Systems* (Ed., Braffort, P. and Hirschberg, D.), North Holland, p. 1.

CHAPTER 8

GAME-PLAYING AND GAME-LEARNING AUTOMATA

D. MICHIE

1. INTRODUCTION

Mechanized game-playing is studied partly for fun and partly for its value as a model for decision-taking in real life, for example in control problems of chemical engineering, or in medical diagnosis, or in animal behaviour. In the last case the two players are the animal, whose moves are called "responses", and its environment, whose moves are called "stimuli" (see Michie, 1962).

A game is a sequence of choices, each choice being made from a number of discrete alternatives. Every sequence terminates in an outcome, and every outcome has a value. It is conventional to assign outcome values from the point of view of the opening player, so that in Chess, for example, there are three outcome values, which we may denote $+1$, 0, and -1 (a win for White, a draw, and a win for Black respectively). Chess belongs to the class of two-person games with perfect information (i.e. both sides are allowed full view of the board) without chance moves. Other examples are Draughts (Checkers), Noughts and Crosses (Tic-tac-toe), Go and Nim. The last two are two-valued games, since drawn positions do not occur. Many-valued games also exist.

2. A GAME AS A GRAPH

A game of the type which we have defined can be represented by a directed graph, as has been done in Fig. 1 for a simple version of Nim. Terminal nodes have been labelled with the corresponding outcome values, on the presumption that a description of the position itself (including which player's turn it is to play) is all we need to know in order to assign a value to it. Although this is true of Nim it is not necessarily the case in general. The value of a position, even a terminal position, could be dependent on the route followed in arriving at it. If there were a rule in Chess that making more than 100 moves automatically loses the game, it would be fruitless for a player to point out, after making his hundredth move, that he could force mate in one.

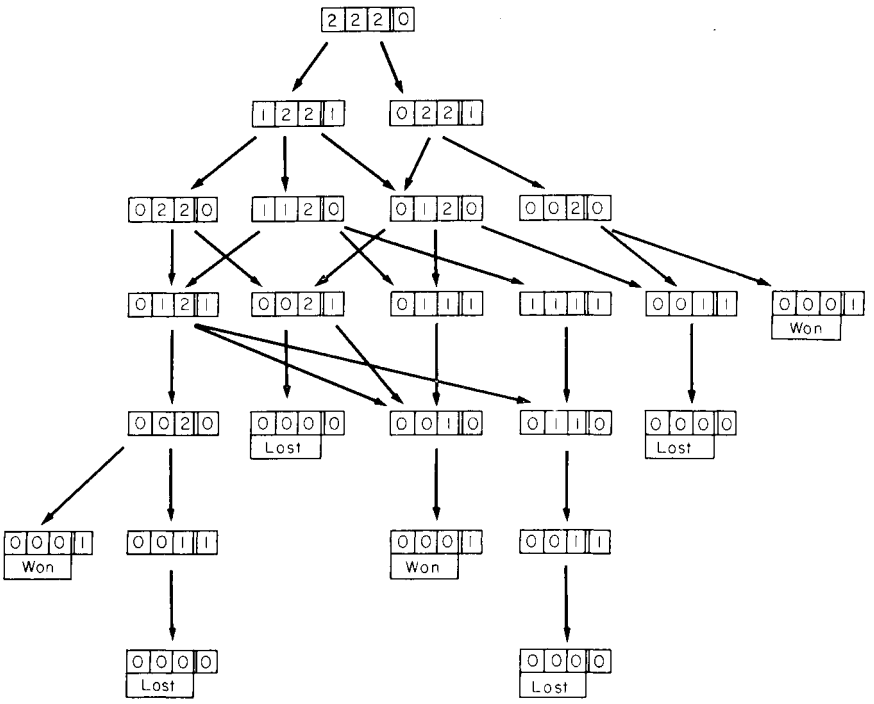


FIG. 1. Representation of an ultra-simple version of the game of Nim by means of a directed graph. The initial state here consists of three heaps of two coins each. The fourth digit in the numerical coding of positions is 0 or 1 according to whether it is the first or second player's turn to play. Moves are made alternately, and consist in the removal of one or more coins from any one heap. The object is to be the player who removes the last coin.

In order to handle this complication it is customary to depict a game not by a generalized graph, in which the nodes stand for positions, but by the special kind of graph known as a *tree*. The nodes are now made to stand not for positions but for *partial plays*. A partial play embodies the total past history from the start up to the given stage and hence is not simply a static photograph of the current position. A graph of positions can be converted in a straightforward manner into a tree of partial plays, as has been done in Fig. 2 for the Nim example.

3. THE 'FOREGONE CONCLUSION' THEOREM

The tree representation permits the demonstration of the fundamental theorem that every game is, in a certain easily definable sense, a foregone

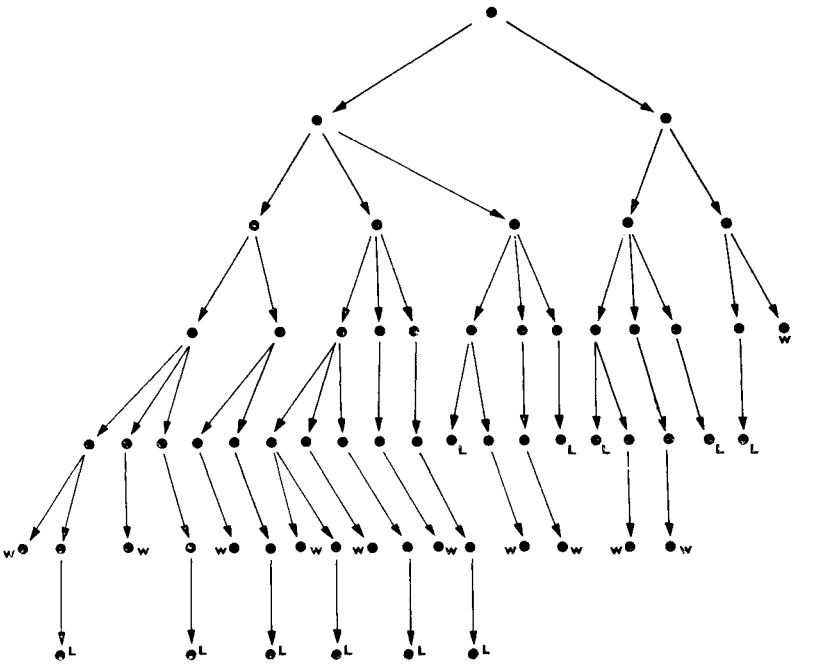


FIG. 2. The graph of the game shown in Fig. 1 expanded into a tree of partial plays. The terminal nodes have been labelled with outcome values: W for a won game (for the first player) and L for a lost game.

conclusion. The theorem applies just as powerfully to "deep" games such as Chess or Go as it does to Noughts and Crosses, where the truth of the theorem is fairly obvious to most adults. It states that, if we assume that both sides follow an optimal strategy, values can be assigned not only to the terminal nodes—these are already assigned by the rules of the game—but to every other node, including the starting node. This initial value is the *value of the game*, and shows what the outcome would be, provided that neither side were to make a mistake.

The method of "backing up" the terminal values to higher and higher levels of the tree is important to grasp, since it foreshadows the "minimaxing" procedure employed in mechanized systems. The terminal spots of the complete tree of the game can be labelled with the values of the corresponding outcomes (win, draw, lose). The pre-terminal nodes can then be assigned values by the following rule: if the node represents our side's choice, then label it with the maximum of the values of the family of terminal nodes dependent from it. If it represents opponent's choice, then label it with the minimum. The assumption is, of course, that our side will play so as to

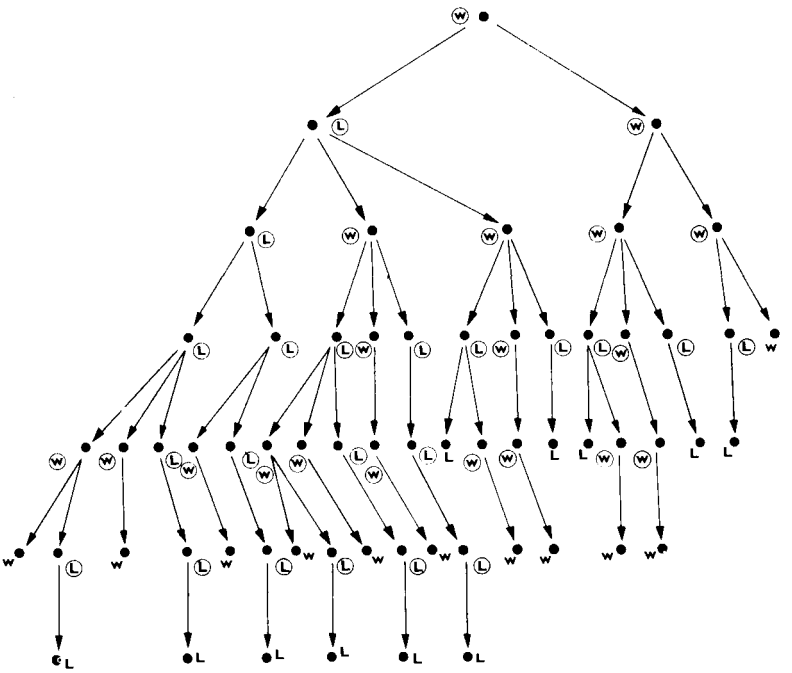


FIG. 3. The tree of partial plays shown in Fig. 2, with all nodes labelled back to the origin by application of the minimax rule. Values obtained by the "backing up" process are ringed. The value of the game is thus established as a win for the first player.

maximize the value of the outcome and conversely for the opponent. The process can obviously be repeated, thus "backing-up" the assigned values to higher and higher values of the tree. Ultimately the initial node will be labelled. This represents the *value* of the game. The process is illustrated in Fig. 3.

4. TWO FALLACIES

It is possible to be misled by the foregone conclusion theorem into thinking that the mechanization of game-playing presents no particular problem. But it is one thing to assert a general theorem about games of this structure and quite another thing to be able to say what the value of any particular game is. Nobody knows whether Chess is a forced win for White, or for Black, or is a draw, nor is anyone ever likely to know: the number of partial plays is certainly greater than the number of elementary particles in the observable universe, so that the task of constructing the required tree is

unthinkable. In the case of a relatively trivial game such as Noughts and Crosses, or the 3 + 3 + 3 version of Nim, total enumeration is feasible, so that an optimal strategy could take the form of an exhaustive list of positions, with a recommended move entered against each.

This is a convenient point at which to consider another misconception of a more subtle kind. We have seen that it is only practicable exhaustively to specify an optimal strategy for a game of relatively trivial dimensions. "Optimal" is here used in a technical sense, as describing any strategy which invariably selects the highest-valued alternative, in the case of the opening player, or the lowest-valued in the case of the second player. Such a strategy guarantees that the outcome cannot be worse than the value of the game. Is it optimal in a different and more practical sense, in ensuring that the expected outcome is as favourable as possible?

The answer is No, for a reason familiar to game-players in connexion with "complicating the position" and the setting of "traps". Against a weaker player it can sometimes pay to make an inferior move with the idea of tempting, bluffing, or bewildering him into making mistakes. More generally, the object of a game is to secure the best result possible against the actual opponent confronting us across the board, rather than the hypothetical opponent defined by the minimax rule. It will be essential to keep this distinction in mind when we later discuss the learning, as opposed to the mere implementation, of effective strategies.

5. COMPRESSED STRATEGIES

The idea of an exhaustive enumeration is useful as a base-line for attempts at compressing the information into a manageable compass. The game of Nim affords a famous example of an elegant compression, namely the rule of play based on conserving the parity of a binary representation of the current position. Compression of a strategy is analogous to the mental processes known as abstraction and generalization, and it is at this point that the challenge to the mechanizer becomes interesting. A compressed strategy is typically based on the evaluation of intermediate positions by means of abstracted features, a process first explicitly propounded in the present context by A. M. Turing, who was responsible for the first detailed thinking about the design of game-playing machines (see Turing 1953).

6. TURING'S IDEAS

It is sometimes thought that Turing's interest in mechanized game playing was the spare time frivolity of a man who reserved his serious thoughts for

worthier topics. This was not the case. He had the conviction that the development of high-speed digital computing equipment would make possible the mechanization of human thought processes of every kind, and that games constituted an ideal model system in which studies of machine intelligence could first be developed. This view is commonplace today although at the time it was regarded askance by many. I shall not stay longer in the pre-history of the subject but will list the basic ideas which Turing contributed to mechanized game-playing.

(1) *Minimax evaluation.* This is an extension of the method referred to in our earlier description of the use of the "tree" representation to assign a value to every choice-point and so establish the 'foregone conclusion' theorem.

Turing pointed out that the method of "backing-up" could be applied to sub-terminal evaluations made on some general strategic grounds without

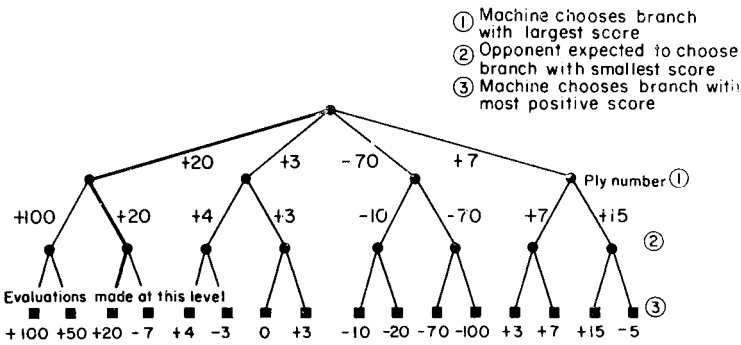


FIG. 4. Simplified diagram showing how the evaluations are backed up through the "tree" of possible moves to arrive at the best next move. The evaluation process starts at (3). (Reproduced from Samuel, 1960, p. 167)

necessitating complete backward tracing from an astronomical number of terminal spots. The crudest of such evaluations would be those provided by the value of own and enemy pieces, rating, for example, in the case of chess, Q = 10, R = 5, B = $3\frac{1}{2}$, Kt = 3, P = 1. Other components of a strategic score might be numerical expressions of such features as mobility, control of central squares, advancement of pawns, etc. Given some means of attaching a score to any position we have a simple scheme for a game-playing machine: look forward along the tree to a certain depth, evaluate all the positions at this depth, and then "back-up" the evaluation by the minimaxing method described earlier. The process is illustrated in Fig. 4.

(2) "*Dead positions*" *and variable lookahead.* Turing defined a "dead position" as one which not only lay beyond the given depth of lookahead,

but could not be reached from within this depth by a capture-recapture sequence. The point of the "dead position" category was to introduce, although in a crude form, the idea of a variable depth of lookahead. The notion was to analyse ahead to a fixed depth, for example 2, and then follow all possible capture-recapture sequences until "dead positions" were reached, before finally making evaluations. Subtler variations on this theme have been subsequently embodied into the program of A. L. Samuel (1959) for Draughts (Checkers). The quality of play which resulted from Turing's application of ideas (1) and (2) was poor, being in part limited by the lack of adequate high-speed computing equipment. The earliest published description of an automaton designed according to these principles seems to be that of Turing (1953), where a complete record of a specimen game is given. This was not, however, Turing's earliest essay in the field. As early as 1944 he was discussing with various friends and colleagues the idea of making computers play chess. In 1948, as a *jeu d'esprit*, he and D. G. Champernowne devised a one-move analyser, the TUROCHAMP. In the same spirit, S. Wylie and D. Michie designed a rival, the MACHIAVELLI.

A tournament between the two machines was attempted at that time, but never completed. Many years later, the MACHIAVELLI was tested in combat with Smith's One-Move Analyser, (the SOMA) designed by J. Maynard Smith (see Maynard Smith and Michie 1961 for the full game record).

In order to give the reader a picture of the kind of board evaluations used in these early automata, the specifications of a SOMA-MACHIAVELLI hybrid are reproduced in the Appendix, contributed by J. Maynard Smith, at the end of this chapter. This machine, when allowed a lookahead of 2, has a standard of play equal to that of a mediocre human player (Maynard Smith and Michie 1961). The design of the TUROCHAMP is no longer extant.

Turing attempted to program both the TUROCHAMP and the MACHIAVELLI on the Manchester Ferranti Mark I in order to play them off against each other automatically, but he never completed it. Shannon (1950) made proposals, not worked out in detail, for the design of a chess-playing program, and this stands as the earliest published treatment of the topic.

It would, however, be unsatisfactory to leave this discussion of chess without exhibiting some specimen of reasonably competent machine play. Unfortunately this is not possible, since there seem to be no survivors of the various chess programs known to have been written (see Newell, Shaw and Simon (1958) and Samuel (1960)) which are capable of really competent chess. On this criterion it appears that no significant advance has been made since the early days, in spite of a widespread and strongly-held belief to the contrary.

(3) *Self-improving behaviour obtained through parameter-adjustment.* We are today familiar in many and varied contexts with the idea of a program which improves its performance in the light of the results obtained, by varying the coefficients or weights attached to the various terms of some linear evaluation function. In bare outline Turing proposed this system also, but its full development had to await Samuel's work on the mechanization of Draughts (Checkers), an outline account of which will now be given.

7. SAMUEL'S WORK WITH THE GAME OF DRAUGHTS (CHECKERS)

During the early 1950's the game of Draughts was programmed by C. Strachey—a considerable *tour de force* at that time—but the work was not taken much beyond the solution of the purely programming problems involved.

It was, however, the first working computer program to combine position-evaluation with a minimaxing lookahead. Evaluation was crude, consisting only of a count of own and enemy pieces. But the lookahead went to a minimum depth (or "ply" in A. L. Samuel's terminology) of 6 , and contained an important means of extending this range. In the forward analysis captures were not counted as moves, so that forcible variations involving exchanges permitted an extended lookahead. This is reminiscent of Turing's suggestion that lookahead might be continued until a "dead position", i.e. one not offering any capture opportunities, was reached. Both schemes aim at directing as high a proportion of the search as possible to the less luxuriantly ramified parts of the tree. This is precisely what expert human players learn to do, exploring only forcible variations to any great depth and otherwise relying largely on their powers of static evaluation. The difference between the Chess or Checkers master and the strong club player does not lie, as people often suppose, in the ability to explore a forward tree of many thousands of nodes and keep track of them somehow in the short-term memory, but rather in a superior instinct of how to restrict the amount of searching that has to be done. The story is told of the chess master who, when asked how many moves ahead he looked, replied "One, the right one"!

Building upon Strachey's foundation, A. L. Samuel (1959) has subsequently engaged on the most thorough and illuminating exercise which is on record. In 1959 Samuel's program was defeating its author with fair regularity. It has subsequently improved its standard with practice, and in 1962 it defeated a former checkers champion of Connecticut, R. W. Nealey. Five further games have since been played, of which four were drawn and one won by Mr. Nealey.

The record of the first game is reproduced below. The annotations were made by Dr. Samuel. Mr. Nealey's comments, as quoted by the *IBM Research News*, are as follows:

> Our game . . . did have its points. Up to the 31st move, all of our play had been previously published except where I evaded "the book" several times in a vain effort to throw the computer's timing off. At the 30–26 loser and onwards, all the play is original with us, so far as I have been able to find. It is very interesting to me to note that the computer had to make several star moves in order to get the win, and that I had several opportunities to draw otherwise. That is why I kept the game going. The machine, therefore, played a perfect ending without one misstep. In the matter of the end game, I have not had such competition from any human being since 1954, when I lost my last game.
>
> Nealey (WHITE) vs. Samuel Checker Program (BLACK), July 12, 1962, at Yorktown, New York. Mr. Nealey was given the option and chose to defend. The Old Fourteenth opening was followed.

11	15		
23	19		
8	11		
22	17		
4	8		
17	13		25–22 would restrict Black's variety of play a little more.
15	18		
24	20		Lee's Old Fourteenth, Var. 9. 11–15 is the trunk move.
9	14		
26	23		Doran's Var. 100 listed as an even game.
10	15		
19	10		
6	15		
28	24		Doran lists 23–19 as giving an easier game for White.
15	19		An aggressive move for Black.
24	15		
5	9		
13	6		
1	19	26	
31	22	15	
11	18		Still in Lee's Var. 9 and Doran's Var. 100
30	26		This is probably a poor move on Mr. Nealey's part.
8	11		A good reply maintaining control of the center.
25	22		
18	25		
29	22		
11	15		
27	23		
15	19		
23	16		

Let us now consider the new features added by Samuel to Turing's original framework of ideas. Of these, the first is what Samuel calls *rote-learning*. At its simplest level this consists in no more than saving the score for future reference whenever a position is evaluated. If the same position is encountered in subsequent play, time which would otherwise be spent on

re-evaluating the position is saved, and thus is available for alternative computations such as extending the lookahead distance.

Rote-learning, however, as so defined, leads immediately to a most important means of exploiting past evaluations. The point is at the same time subtle and simple. Many of the evaluations which are saved have themselves been obtained by "backing-up" from spots at a lower level in the game tree, as was described earlier. In a subsequent play of the game such a position might again be encountered, this time in the lookahead from the current position on the board. The effect of having saved its value is then that the *effective* lookahead has been extended. The point is illustrated in Fig. 5.

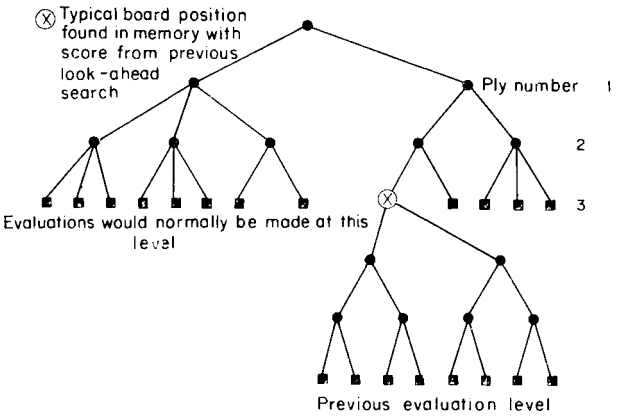


FIG. 5. Simplified representation of the rote-learning process, in which information saved from a previous game is used to increase the effective ply of the backed-up score. (Reproduced from Samuel, 1960, p. 183.)

A further important feature of rote-learning as implemented by Samuel causes the program to tend to delaying tactics when losing, and the opposite when winning. I shall not here go into the details of this, nor of many other ingenious devices which Samuel employed in creating the full operating system, but will proceed directly to the learning-by-generalization system which Samuel ultimately combined with the rote-learning routine. As mentioned before, this was based on a linear scoring function consisting of a number of terms each intended to express some strategic feature of the game, together with weighting coefficients. During the learning procedure the program caused two phantom players, Alpha and Beta, to play against each other, and adjustments to be made in the weighting coefficients according to the relative success of the two sides. Fig. 6 shows the early phase of a learning process of this kind. Terms whose coefficients gravitated to zero or near-zero values were dropped from the list and replaced by new terms selected from a

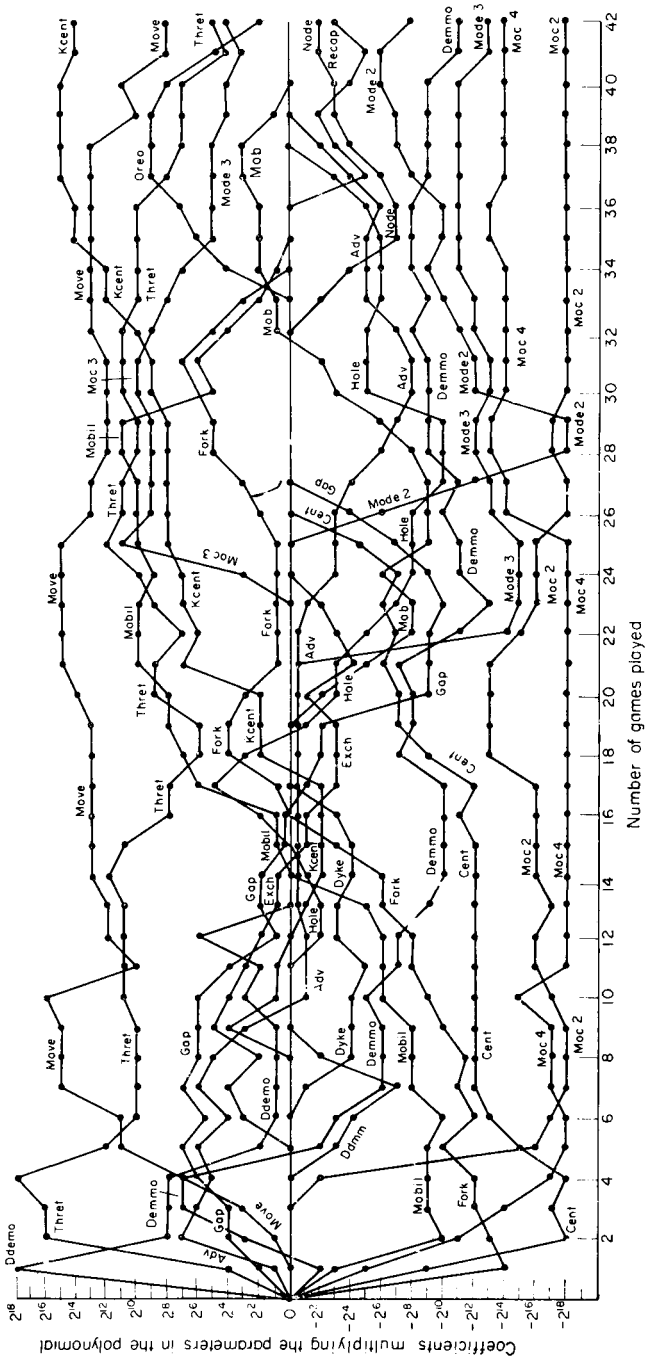


FIG. 6. Second series of learning-by-generalization tests. Coefficients assigned by the program to the more significant parameters of the evaluation polynomial plotted as a function of the number of games played. Two regions of special interest migh: be noted: (1) the situation after 13 or 14 games, when the program found that the initial signs of many of the terms had been set incorrectly, and (2) the conditions of relative stability which are beginning to show up after 31 or 33 games. (Reproduced from Samuel, 1959.)

reserve list. Each term stands for some strategic feature of the game. Thus ADV = "advancement", KCENT = "king centre control" etc. The method used to modify the coefficients attached to the various terms was based on the idea of bringing about as good agreement as possible between the values calculated for different positions connected by the same minimax chain. For details of this and other features of the program, the reader is referred to Samuel's published descriptions.

The fact that Samuel's list of terms was man-made and that no means was devised whereby the program could generate new terms for itself is significant, for it is precisely at this point that current research on "artificial intelligence" is encountering great difficulty. Deductive processes are in principle easy to mechanise. But the intellectual processes involved in induction, with their aura of "creativity", "originality", "concept-formation", etc., are difficult to capture within a formal framework.

It would be unprofitable to pursue these questions here. Rather, we shall now turn from games of such complexity that schemes of classification and evaluation are positively forced upon the would-be mechaniser, and consider the following question. In simple games for which individual storage of all past board positions is feasible, is any optimal learning algorithm known?

8. THE "TWO-ARMED BANDIT" PROBLEM

The surprising answer to the last question is "No". In other words, it is beyond our present understanding to design an automaton guaranteed to make the most efficient possible job of learning even games more trivial than Noughts and Crosses. The difficulty lies in costing the acquisition of information for future use at the expense of present expected gain. A means of expressing the value of the former in terms of the latter would lead directly to the required algorithm. We can briefly illustrate the point by scaling the problem down to the simplest task definable as a two-person game. Such a game will have two outcomes (win, lose), two stages (one choice for each player) and two alternative moves at each choice-point. The structure of the game is shown in Fig. 7. The learning problem is that of the opening player, whose task is to gain the greatest possible number of wins in N successive plays of the game on the basis of no prior knowledge of his opponent's behaviour.

In the special case where the opponent responds to M_1, with losing and winning moves in the proportions p_1, $(1 - p_1)$ and to M_2 with losing and winning moves in the proportions p_2, $(1 - p_2)$, where p_1 and p_2 are stochastic variables of fixed, but initially unknown, values, the problem is identical with a well-known problem as the "two-armed bandit" problem. Here, in place of the opponent with his two modes of response we have two gambling

machines characterized by the stochastic variables p_1 and p_2 representing mutually independent probabilities of pay-off. Given some assumption about the prior probability distributions of p_1 and p_2 (e.g. that the prior distributions are uniform), the problem is to devise a policy for choosing between the machines on successive trials in such a way as to maximize the expected monetary gain over N trials, or alternatively over an infinite succession of trials during which the value of the pay-off is suffering a steady exponential decay. At first sight the problem looks easy, since there is no particular

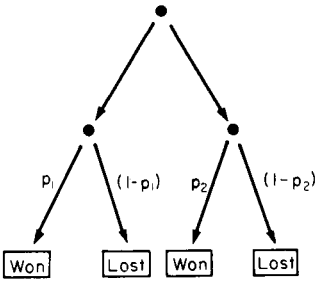


FIG. 7. The formal structure of the simplest possible two-person game, in which each player is allowed one move, each move being a choice of two alternatives. The second player, if he were a non-imbecilic human, would infallibly win so trivial a game. Imagine, however, a fallible player, whose replies to the two possible openings are made with probabilities p_1, p_2, initially unknown to the first player. Then the first player's task of learning to play the game is equivalent to the two-armed bandit problem (see text).

difficulty of forming *estimates* of p_1 and p_2 from the results of past trials, which we may denote by $\hat{p}_1$ and $\hat{p}_2$. It now seems common sense to choose M_1 or M_2 according as $\hat{p}_1 > \hat{p}_2$ or $\hat{p}_1 < \hat{p}_2$. On reflection this can be seen to be the right answer to the wrong question, as though we had been asked for a policy which would always maximize the expected gain on the *next* trial, instead of the expected gain accumulated over all remaining trials. It can be seen, even on intuitive grounds, that in some such case as

M_1: 2 trials, 1 success,

M_2: 99 trials, 50 successes,

M_1 is the better choice for the next trial rather than M_2 because of the much greater information increment, from which cash benefit can be extracted in future choices.

The two-armed bandit problem is still unsolved, and *a fortiori* so is the problem of optimizing the design of a game-learning automaton for even the simplest of games.

9. CONCLUDING REMARKS

Given that the key problems of mechanizing the learning both of easy and of difficult games seem still a long way from solution, one may ask whether the further experimental pursuit of the subject is truly worth the effort, when so many more pressing applications of computer science claim attention. Any answer to this question must be sought in the special challenge offered by games to develop and perfect new technique, and to isolate in pure form the logical structure underlying real-life systems. In view of the prevalence of multi-stage decision processes in practical contexts, the use of games as testing-grounds for automatic methods seems assured of continuance.

10. REFERENCES

MAYNARD SMITH, J. and MICHIE, D. (1961) Machines that play games, *New Scientist*, **12**, 367–9.

MICHIE, D. (1962) Puzzle-learning versus game-learning in studies of behaviour, pp. 90–100 of *The Scientist Speculates* (I. J. Good, ed.) Heinemann, London.

NEWELL, A., SHAW, J. C. and SIMON, H. (1958) Chess-playing programs and the problem of complexity, *IBM Journal of Research and Development*, **2**, 320–35.

SAMUEL, A. L. (1959) Some studies in machine learning using the game of checkers, *IBM Journal of Research and Development*, **3**, 211–29.

SAMUEL, A. L. (1960) Programming computers to play games, pp. 165–92 of *Advances in Computers*, vol. 1 (F. L. Alt, ed.) Academic Press, London and New York.

SHANNON (1950) Programming a digital computer for playing chess, *Phil. Mag.* **41**, 356–75.

TURING, A. M. (1953), pp. 288–95 of *Faster than Thought* (B. V. Bowden, ed.), Pitman, London.

APPENDIX

JOHN MAYNARD SMITH

Rules of SOMAC*

(SOMA with features taken from the MACHIAVELLI)

1. GENERAL (A)

It is supposed that SOMAC is playing the White pieces. If there are n legal moves, calculate the values $V_1, V_2, \ldots, V_n$ of White's position after each of them, and make the move giving the maximum value. If two or more moves give equal values, choose at random. In practice it is easier to calculate $V_1 - V_0$, etc., where V_0 is value of White's position before moving.

* Numbered notes (1) (2), etc., explain how moves are calculated. Lettered notes (A), (B), etc., comment on effects of rules, and are at the end.

The value of White's position is equal to:

	Note
Value of White's pieces*—Value of Black's pieces	(1)
+Value of squares attacked by White pieces	(2)
—"Exchange Value" of White pieces	(3)
+"Exchange Value" of Black pieces	(3)
−5 for every White piece (*not* pawns) which, although not attacked by a Black pawn, can be so attacked by a single Black move (i.e. by an advance or capture by a black pawn) (B)	(4)
$+5/x$ if Black is in check, where x is the number of legal moves available to Black (C)	
+5 if O — O and/or O — O — O is a legal move; +16 if O — O has been played; +14 if O — O — O has been played (D)	(5)

Note (1) *Value of pieces*

$$P = 10;\ N = B = 30;\ R = 50;\ Q = 90 \text{ or } 100;\ K = 1000$$

The value of a Queen is 90 if it is not on the back two ranks and the player has not castled; otherwise Q = 100. Thus if White, before castling, moves Q from the back two ranks, this reduces the value of his position by 10: if he then castles this increases the value of his position by 10. (E)

Note (2) *Value of squares attacked*

+1 for every square, occupied or not, attacked by a White piece (2i)
+1 extra for every attack on one of the four central squares
+2 extra for every attack on a square adjacent to the Black King (2ii)

Note that the squares attacked by each White piece are enumerated separately and added, so that if a square is attacked by r White Pieces, this adds r to the value of his position.

Note (2i)

A piece attacks a square (occupied by a Black or White piece or unoccupied) if it could capture a piece of opposite colour on that square: but this rule is modified by "*pins*" and "*transparencies*" as follows: (F).

Pins. A White piece "X" does not attack a square if, should it move to that square, *either*, a Black piece attacks a White piece (other than X) of Value higher than itself, which it did not attack before "X" was moved, *or* a Black piece attacks a White piece (*not* a pawn; *not* X) which it did not attack before X was moved, and which occupies a square not itself attacked by any other White piece.

* Unless otherwise stated "Pieces" includes pawns and King.

Transparencies. In calculating squares "attacked", certain pieces are assumed, contrary to the rules of the game, to be allowed to move through squares occupied by other pieces of the same colour, which are then said to be transparent, as follows:

R is transparent to R,
R is transparent to Q on files and ranks only,
B is transparent to Q on diagonals only,
Q is transparent to R and B,
P is transparent to B and Q, along diagonals, but only for one square beyond pawn, and only if B or Q are moving forwards along the diagonal.

Note (2ii)
A square is "adjacent" to the King if it is attacked by the King.

Note (3) *Calculation of exchange values* (*G*)

(In what follows, a piece is said to be attacked by another piece if it occupies a square attacked by that piece: squares attacked by Black pieces are calculated, *mutatis mutandis*, in the same way as squares attacked by White pieces.)
Step A.
Calculate "Swap-off Value" S for all pieces, of either colour which are *en prise* i.e. attacked by a piece of opposite colour.

Swap-off value of a White piece (*H*)

(reverse this procedure for a Black piece)

A White piece of Value v_0 is attacked by n White pieces of value v_1, $v_2, \ldots, v_n$ in ascending order of value and by N Black pieces of values u_1, $u_2, \ldots, u_N$ in ascending order of value (3i).
Calculate:

$$w_1 = v_0 \qquad b_1 = v_0 - u_1$$

$$w_2 = v_0 - u_1 + v_1 \qquad b_2 = v_0 - u_1 + v_1 - u_2$$

and so on, until the series $w_1, b_1, w_2, b_2, w_3 \ldots$ terminates (it cannot terminate before v_0).

(i) if $n \geqslant N$ (series ends at b_N),
S = largest value of b, or, if smaller, the smallest value of w preceding this.

(ii) if $n < N$ (series ends at w_{n+1}),
S = smallest value of w, or, if larger, the largest value of b preceding this.

Note (3i). If a piece X only attacks the square by virtue of the assumed transparency of another piece Y, then even if $v_x < v_y$ (i.e. Value of X < Value of Y) v_x cannot precede v_y in this series.

Step B

Calculation of exchange value of White pieces = V_{EW} (*I*)

Ignore negative and zero values of S.

V_{EW} = Largest positive value of S on any White piece +5 for each additional White piece with a positive S. (I)

Step C

Calculation of exchange value of Black pieces = V_{EB} (*J*)

If only one Black piece has a positive value of S, $V_{EB} = +5$.

If r Black pieces have positive values of S, $V_{EB} = 5(r - 1)$ + Second highest Value of S. (J)

In calculating r, any White piece which is *en prise* with a zero or positive value of S is deemed to attack only one Black piece, which is taken to be that piece which maximises r.

If $r > 1$, but would have been 1 or 0 had it not been that a White piece with a negative S value $= -S'$ attacked two or more Black pieces, then V_{EB} is taken as *either* the value as calculated above, *or* as S', whichever is less.

Note (4). Only one value of −5 is scored for a single White piece, even if it could be attacked by two or more Black pawn moves.

Note (5). If a player has castled, castling is no longer a legal move: so the net gains for O − O and O − O − O are +11 and +9 respectively.

Comments on SOMAC

A. *General* The "currency" is based on a value of 10 for a pawn, of 1 for an attack on a square by a piece, and of 5 for having the short-term "initiative". These relative values are about right, judging by the way SOMAC plays.

B. White scores −5 if his opponent can gain the initiative (i.e. force White to move a particular piece) by a pawn move. A several-move analyser could probably dispense with this rather arbitrary addition.

C. This is a rule from MACHIAVELLI. It ensures that SOMAC will play a mating move if one is available ($x = 0$; $5/x = \infty$). If SOMAC himself is in check, he will move out of check if he can, because in check he has an "exchange value" of $\simeq -1000$. There is no built-in rule to say "I am checkmated".

D. A one-move analyser cannot be persuaded to castle without a special inducement.

E. This, though rather inelegant, is to prevent the Q moving out too soon. Without it, White would, for example, after 1:P–K4, P–K4, play 2:Q–B3?

instead of 2:Kt–KB3. It might be simpler and perhaps more effective, but equally inelegant, to impose a penalty for moving Q before, say, 2 minor pieces have been moved.

F. It would not be worth allowing for pins and transparencies if it were only a matter of calculating the value of the squares attacked; but it is vital in calculating the exchange values. The rule on pins works quite well. The rule on transparencies has snags. The main one is that in which White has castled, and has a K, Q and 2R's only on the back rank. In this case, since the rooks are transparent to the Queen, there is little to encourage the Q to move off the back rank. But this is perhaps a minor snag.

G. This is really the main part of the machine as far as combinational Chess is concerned. It could almost certainly be greatly improved.

H. The Swap-off value of a piece is in effect what a player would lose if his opponent captured it.

I. This is simple. It assumes Black will capture a White piece if he gains material. The +5 is for initiative.

J. This assumes that Black will move or defend the piece with the largest *S*. The 5 is for initiative. The difficulties arise because of "inter-locking" between pieces, with one piece defending two others, etc.

CHAPTER 9

INFORMATION RETRIEVAL AND SOME COGNATE COMPUTING PROBLEMS

R. M. NEEDHAM

1. INTRODUCTION

The first point we have to consider is what we mean by information retrieval. In Chapter 6 it was described as obtaining information from a collection given a very definite key as a search guide: an example is getting information about a chemical compound from a list of compounds and their properties, given a precise description of the chemical structure of the compound as a key. I maintain that this is really a data-processing problem, and is not typical of information retrieval. In my view, and I think that most people would agree with this, information retrieval consists of obtaining the required document or documents from a collection, given a request specifying a subject or topic, such as "I want anything on Russian inflections", or "What have you on Smithson's process for preparing oxalic acid by distilling lemon juice?". The essential object of information retrieval, that is, is to give the person making a request what he wants without his having to check through the whole document collection.

How do we set about doing this? We can diagram the information process as a whole as shown in Fig. 1.

To the left, at A, we have a pile of documents; to the right, the person wanting information. The whole object of the enterprise is to avoid bringing the person into direct contact with the documents; instead, we have some representation of the content of the documents, and some encodement of his request, and some procedure for matching the latter against the former so that documents with the appropriate or relevant content are selected. At B, therefore, we have some form of hardware containing, to put it as loosely as possible, bundles of notations representing documents; at D we have a bundle of notation for the request; and at C we have some matching algorithm.

The history of information retrieval research consists of attempts to mechanize an increasing amount of the process. The first point of attack was

the algorithm, which could be carried out on a computer or on various forms of punched card. The problem is to find the most efficient matching procedure. The simplest approach is to look for an identity match of request and document specifications. This means, however, that the request has to represent the whole document, which is clearly unsatisfactory. The next simplest approach is to look for set inclusion of the request specification in the document specification: if we have a request represented by properties $P_1, \ldots, P_k$, a document with $P_1, \ldots, P_n$ will be relevant. This is in fact a good approach if one is concerned with saving real money in a real library.

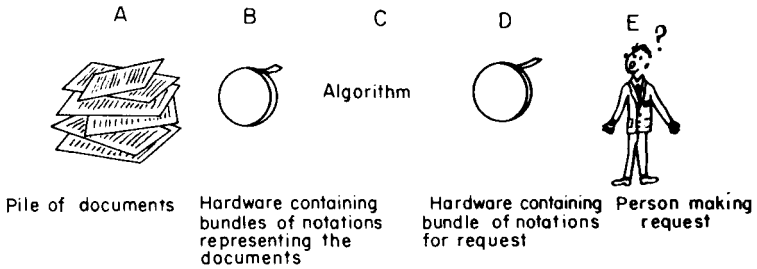


FIG. 1.

Finding a suitable matching algorithm does not, however, mean that we can set up our system; we still have to deal with the basic problem of finding our properties. How much work, then, is involved in turning documents and requests into bundles of notations? The important point is that the algorithm and the encodement are complementary: the easier one is, the more difficult the other.

2. UNITERMS AND DESCRIPTORS

The easiest way of encoding documents is to pick out words, or keywords, from the text; a document will thus be represented by a word list. This can be done either by mechanized statistical processes, or by a human being underlining words he thinks important. This approach was described by Taube *et al.* (1952) who called the selected words *uniterms*. It has the advantage that the extraction process is very simple, but the disadvantage that the specification is restricted to words which actually occur in the text: this means that if a document is specified by, say, "calculating machine" and a request contains "computer", the two will not match although one would say that a document about calculating machines is relevant to a request for documents about computers.

To deal with this problem we find that we are obliged to construct an elaborate linguistic apparatus, that is, a thesaurus, which we use with our

algorithm. This linguistic structure bears the same relation to the algorithm as the submerged part of an iceberg does to its visible part; more specifically, a thesaurus of any size turns out to be difficult to compile, to store, to access, and to keep up to date.

To avoid these difficulties we may try the alternative method of indexing our documents by *descriptors*. The distinction between uniterms and descriptors is an extremely important one: a uniterm is a word; a descriptor is represented by a word, but is not a word: it is a stylized notion. A good example is the descriptor "micro" used in a retrieval scheme at the Royal Radar Establishment: this descriptor is imputed to any document dealing with anything smaller than normal, whether they are cars, computer components or people. In using descriptors we thus avoid the characteristic defect of uniterms; but we may, on the other hand, get involved in problems of assignment which cannot arise with uniterms: we have to have a rule for each descriptor which tells us how to use it. The rule for "micro", therefore, is that it applies to anything unusually small, or miniature. The rule can simply be a list of the words which can appear in a document which suggest that the descriptor is appropriate; we have, that is, a bit of thesaurus.

The main advantage of descriptors is that we can have a very small number of them; the R.R.E. scheme uses 70, and a system in use in the Meteorology Laboratory of Cambridge has about 160. If we use uniterms, we will get as many as there are keywords in our documents, and this can run into thousands for quite a small collection. Descriptors can also be very reliable if they are properly set up, which means that we do not have the submerged part of the iceberg. In retrieval we have of course to turn the request into a set of descriptors; we can then apply our algorithm—set inclusion of the request set in a document set.

It should be noticed that we have not strictly abolished the submerged berg, but merely shifted the effort to the points where we replace documents and requests by descriptors. This does not, however, matter too much as experience shows that people can provide descriptor notations quite easily provided the descriptors and rules they have to use are properly presented.

Many descriptor systems are currently in use for collections of up to 20,000 to 30,000 documents; they are implemented on all kinds of hardware, from magnetic tape files to optical incidence cards. In card systems there is usually a card for each descriptor, with the numbers of the documents having the descriptor punched on it, and retrieval simply consists of a peek-a-boo operation. Cards have the advantage that they are cheap, not very bulky, and easily used. They can be quite big: cards or card-like things are available with over 20,000 positions. They have, however, some disadvantages: they cannot be really large, and the holes may be inconveniently small; a more important point is that once information has been put on this kind of

card it is, as it were, embalmed: there are no processing machines available for most types of card, so that they cannot, for example, be copied mechanically when they begin to wear out; they cannot be input as they stand to a computer, which means that the information they contain cannot be transferred to a machine store; and they are not a standard output medium either, so that there is no automatic way, for example, of producing a new set of cards each time one's retrieval system is updated.*

These are, however, practical problems of implementation which have nothing to do with the efficiency of descriptors as a method of indexing documents. Now there is no doubt that descriptor systems are quite adequate in many cases; why, therefore, should one go any further?

The fact is, one can get into difficulties with descriptors (that is, with the actual descriptors, and not the matching algorithm). Firstly, one may have a bad descriptor set. With a large set of documents, one can set up an apparently efficient set of descriptors, and then find they do not work. This is where we have to consider the process of generating descriptors. It usually represents the interaction between a particular set of people and a particular set of documents: one may set about finding descriptors for a particular library by asking one of the users "Why should this document not be thrown away"? If there is too much material this is impossible; instead, the person providing the descriptors sits down and thinks up what the descriptors ought to be; he approaches the problem in an *a priori* way, rather like a taxonomist. The result tends to be that the descriptors do not work.

It may, however, be that the problem is a more serious one; it may be that descriptors in themselves are inappropriate where large collections are concerned.

3. DESCRIPTORS AND SYNTAX

When we use descriptors we lose what we can describe as the syntax of our documents: we cannot distinguish between a document about exports from Britain to Russia, and one about exports from Russia to Britain as we simply use the three descriptors "exports", "Britain" and "Russia". Here we are up against the inadequacy of descriptors in general, rather than that of any particular descriptors. With a small library the ambiguity will usually not matter: but with a large one it will. If we want to deal with this problem, therefore, we have to encode the document syntax in some way. We may do this by combining descriptors with some rather general syntactic relations,

* These remarks may soon be obsolete. Work is going on at the All-Union Institute of Scientific and Technical Information, Moscow, on photographic preparation of Peek-a-boo cards from computer-produced punched cards. There may well be similar efforts elsewhere.

such as "effects", or "causes", or "is associated with"; a document will then be represented by a structure, for instance

$$D_1 \underline{\ R_1\ } D_2 \underline{\ R_2\ } D_3.$$

This approach has been studied by Cros *et al.* (1964), who have put forward a complete system called Le SYNTOL; this is an artificial language for describing documents which contains a large number of descriptors and four syntactic relations.

This kind of approach means that the bundles of notations for documents, which have to be provided by human beings, are much more complicated—they may have a bracket structure—so that there is a problem in storing them. The request has of course to be put in the same form.

This brings up the question of the retrieval algorithm. With descriptors and uniterms an implicit or explicit data structure is required, but the algorithm is simple. With a very detailed structured notation errors and variations in encoding are much more likely, so that the matching process becomes much more complicated. For instance, given the three structures

$$(A \underline{\ R_1\ } B) \underline{\ R_2\ } C \underline{\ R_3\ } D$$
$$(A \underline{\ R_1\ } B) \underline{\ R_3\ } D$$
$$(A \underline{\ R_1\ } B) \underline{\ R_2\ } D$$

are they, for matching purposes, to be regarded as the same or not?

Gardin's is the most sophisticated indexing attempt to date; and it is worth noting that in tests in 1962, 60% of the retrieval was by descriptor match only: that is the relations had to be abandoned before any selection from the collection could be effected. (It is hardly necessary to point out that this is no longer true.)

There is nevertheless general agreement that some syntax is required; the only question is how much: the more there is, the more likely it is to be wrong, and the more work has to be done if unpicking is required in matching.

4. AUTOMATIC LANGUAGE PROCESSING

If we want to be sure that the structure at least is right, we have to mechanize the encodement of document and request. We have, that is, to go in for full scale text-processing, if not of the whole document, of some or all of the title, abstract and references; and we have to process the request in the same way. Now it seems fairly obvious that the actual method of processing should be the same for document and request. This means that statistical processes are not as suitable as they might appear to be: we can apply statistical procedures like word-counting to the text, but we cannot apply them to the request for the simple reason that it is not long enough.

We cannot, therefore, hope to get anything from what can be described as a partial automatic analysis of the document: the only approach open to us is to attempt a full-scale analysis of the text, both syntactic and semantic. At this point we are concerned with problems which are common to research on machine translation and automatic abstracting, as well as information retrieval, and I shall therefore go on to consider these general problems of automatic language processing, without specific reference to information retrieval.

To bring out the connection between information retrieval and language data processing, we can consider the following approach to retrieval. Suppose that we want to analyse an abstract; we can parse the sentences in

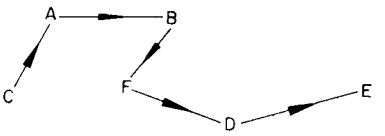


FIG. 2.

the abstract with a phrase structure grammar; we refer to a thesaurus to turn the words into descriptors. We can then obtain a graph representing the structure of the abstract, as shown in Fig. 2. Here the nodes represent descriptors and links of one or more sorts and represent what can be described as distilled syntax. We carry out the same analysis of the request, and then match structures. This is where we get into difficulties, because we will need an extremely sophisticated program to tell us whether two graphs are isomorphic, or one is a sub-graph, or partial graph, or partial sub-graph of the other. Work on these lines is being carried out by Salton (1961). Its importance in this context lies in the fact that the connection with machine translation research is very close: the syntactic analysis which Salton carries out is very much the same as the kind of analysis which has to be carried out on a translation input text.

We have, then, to consider what are the implications of embarking on natural language text processing.

Machine translation was one of the earliest large-scale non-numerical computer applications: programs for the IBM 704 containing 30,000 hand-coded machine orders existed as early as 1957. They did not in fact give acceptable translations. This was because people thought that the whole problem was a technological one: there were numerous dictionaries and grammars available, and it was thought that machine translation merely involved writing programs for encoding and looking up these dictionaries and for encoding and applying the grammar-book rules. This was done on a large scale, mostly on an *ad hoc* basis: no attempt was made to separate the

grammar of the language from the parsing program. The approach was carried to its logical conclusion by some workers at Georgetown University, who used a dictionary which gave, for awkward French words, an English word and a special-purpose subroutine for dealing with it.

The method, however, suffered from two defects; it is an inefficient way of programming, and, more importantly, the information used was inadequate and unsuitable; grammar books are generally not specific enough, as they are intended for human beings who can interpret and supplement them from their own knowledge. Machines cannot do this.

Subsequent work has therefore been concerned both with assembling better linguistic data and with constructing better programs for using it. The kind of programming effort involved could be illustrated by a discussion of methods for dictionary lookup: all translation programs make use of a lookup at some stage, which means that this is a procedure which has been investigated in some detail; and the fact that it has been programmed by a number of research groups means that several different approaches can be compared.

5. AUTOMATIC SYNTAX ANALYSIS

Most of the research effort in natural language processing has gone into syntax. It is worth considering in some detail as syntax in natural language can usefully be compared with syntax in formal languages, like programming languages.

The first attempts to carry out automatic syntax analysis were based, as we saw, on programming grammar books. Syntax studies are now much more formal: everyone works within the framework of a context-free phrase structure grammar, that is, in a Chomsky system of Type 2. These grammars, as illustrated in Chapter 4, contain rules such as

$$A \rightarrow aBc$$

where A and B are categories, a and c are individuals.

The formal framework is thus the same for natural language and artificial language grammars. The distinction between the two (syntactically speaking) lies in the amount and character of the ambiguity they can exhibit. In an artificial language there is no ambiguity, or at most very little: a given individual unit, that is, can normally only belong to one syntactic class or type. Artificial languages are in fact intended to be unambiguous, and if we find that such a language is ambiguous we change it. A natural language like English is ambiguous, and we cannot change it: we have to accept the ambiguity. The real distinction, however, given that an artificial language

may be ambiguous, is that between local and non-local ambiguity. Suppose that we have the string

a b c d e f g

and *a* can be a member of either A or B. In an artificial language this will be resolved very quickly, by a reference to the next item *b*, or at most by *c*; the immediately surrounding context will make it clear which is appropriate, so that the analysis can be comparatively straightforward. In a natural language, in contrast, we may not resolve the ambiguity until we have proceeded much further with our analysis; we may well find that we can only tell whether A or B is appropriate when we have reached the last item in the sentence. This means that we have to record all the alternatives until we can resolve them, which may well be a long time. We may indeed get alternatives for a whole sentence, and even if we do not get this we may have ambiguities which last almost until we have reached the end of the sentence. To avoid this we may either produce all the alternative analyses in parallel, or set up a sequence by making use of some heuristic preference order of alternatives. The only difficulty in the second case is getting the heuristics right. This is, however, such a problem that a parallel method is more usual. The difficulties of this approach are well exemplified by some early experience at Harvard, where it was not uncommon to find a dozen or more alternative analyses for a sentence.

Even if the formal structure of the syntax is the same for the two kinds of language, therefore, the problems of using the analyser are quite different. This can be brought out by considering natural language procedures in more detail.

6. PROCEDURES FOR NATURAL LANGUAGES

We say that an individual item, say *a*, is a *minimal constitute*, and that (*a b c*) is a *constitute*. We have a table or algorithm available for looking up any pair of constitutes, to find whether a constitute could exist of which the two given items are immediate constituents. This can be done binarily, although to do so is not always the most efficient course. The problem of parsing strategies is to discover the precise order in which we refer to the table the very numerous pairs which are available. There are two basic approaches to the problem, sometimes called *morsel* and *backtrack*; a crude morsel procedure would look something like the following.

We initially have a sequence, say

a b c$^{\downarrow}$ *d e*

and a pointer, initially as shown. Our policy is always to examine the two items to the right of the pointer, if there are two items there. Suppose that

d and *e* will combine to make *a* constitute *f*; then we make a new copy of the sequence

$$a \quad b \quad c^{\downarrow} \quad f$$

There are now no longer two items to the right of the pointer, so it is moved one place left, giving the *two* sequences

$$a \quad b^{\downarrow} \quad c \quad d \quad e$$
$$a \quad b^{\downarrow} \quad c \quad f$$

Now there are two pairs to examine, namely *cd* and *cf*, and if both qualify our list will then appear as

$$a \quad b^{\downarrow} \quad c \quad d \quad e$$
$$a \quad b^{\downarrow} \quad c \quad f$$
$$a \quad b^{\downarrow} \quad g \quad e$$
$$a \quad b^{\downarrow} \quad h$$

and there is still the pair *ge* to examine. It is easy to verify that if this procedure is carried out until the pointer is at the far left of the sequence, then, in the doubtless very long list of partially parsed sequences, any member which consists of a single symbol will indicate a complete parsing; all such complete parsings are different, and there cannot be any more.

The alternative is to start at the beginning of the sequence. We ask: can *a* be the first item in a pair which are the immediate constituents of a sentence; if not, can it be the first item in a pair which is the first item in a pair which are the immediate constituents of the sentence, and so on. If we find that our potential structure is wrong we have to back track and unpick it. This may take more time than the first method, but uses less space. It can also be added to heuristically, as we can manipulate the order of the possibilities we consider; this means that we have a reasonable chance of getting the right answer quite quickly, though if we do not we may be in a worse position than we would have been if we had accumulated information in the other way.

Most existing programs are in fact of the first type, most probably because no-one has obtained any useful information for setting up the heuristics required for the second. They do not work very well, and one wonders why. They do suffer from space and time problems: it has been known to take 10 minutes on a 7090 to sort out one sentence. One wants to ask whether this is because the sets of categories used are wrong, that is whether our rules of the form

$$\mathrm{A} \rightarrow b \quad c$$

are incorrect or inappropriate. There is indeed a wide choice of categories available; the only trouble is that there are no rules for choosing the right ones.

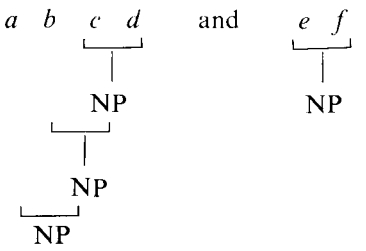
7. SEMANTICS

We can attempt to deal with the problem of alternatives by what can be crudely described as cramming semantics into syntax, that is, by multiplying rules in the following way: if we consider nouns, the simplest categorization is simply to use a class labelled "Noun"; if this is inadequate we construct a class "Noun or Verb"; and if this is inadequate we try a class "Animate Noun". I maintain that animateness verges on being a semantic notion; and there is no limit on this extension of the classification. The whole system becomes immensely complicated and, if not impossible, very difficult to handle. Enlarging one's classification in this way is particularly dangerous if it is undertaken in the middle of one's research: suppose, for instance, that one updates one's grammar after parsing 10,000 sentences. This is a very natural thing to do, but one is immediately faced with the problem of discovering whether one's updated grammar still works for the sentences already treated; and checking back on 10,000 sentences can be very tedious.

As it may turn out, therefore, that there is no very distinct division between syntax and semantics, we should go on to consider semantics specifically.

It was generally believed, in the early days of machine translation research, that there are no semantic problems in translation: if there are semantic ambiguities, these can be resolved by simple context markers: a simple label "Finance", say, would deal with the ambiguity of "bank". It turned out, however, that the semantic classification had to be more elaborate than was first thought if it was to be at all effective; and current semantic research is largely concerned with the problem of constructing an appropriate classification.

Semantic analysis is also related to syntax in another way: we may use a semantic classification to tidy up the output of our syntax program, or as an aid in parsing. This is well illustrated if we consider the problems involved in discovering the scope of conjunctions: syntactically we can only say that the constitutes conjoined must have the same syntactic form; we can, however, also require that they have the same semantic form, in some sense. For instance, suppose that we have the noun phrase structure



We select the correct noun phrase on the left of the conjunction, that is, the partner of the one on the right, by looking at the semantic categories concerned: the words in the correct noun phrase should be members of the same semantic category or categories as those on the right hand phrase.

It may be that this can be achieved by a comparatively crude semantic classification, which would clearly be an advantage. The hope is that we may use the same classification for resolving semantic and syntactic ambiguity; that is to say, it might be the case that a semantic structure which is powerful enough to resolve multiple meaning will resolve syntactic ambiguities as well.

8. TIMETABLING

I shall at this point go on to discuss an apparently quite separate problem, namely timetabling (in the school as opposed to railway sense). What I hope to show is that common types of problem arise in such different fields as information retrieval and machine translation on the one hand and timetabling on the other, and moreover that techniques useful in them are also relevant where other non-numerical applications of computers are concerned.

School timetabling is symptomatic of a large class of problems, and I shall consider it in a little detail.

When we set about constructing a timetable we have to take a number of absolute constraints into account, like not having two classes in the same room, and not having one master for two classes in two different places. There are also what can be described as optional constraints, such as obtaining a fairly even distribution of each master's free periods. We then have to decide, given these restraints, how we should set up a timetable which satisfies them. One approach is to consider all the possible distributions. This is, however, a theoretical rather than practical line of attack, as it can easily result in a program containing 10^{500} operations. An alternative is to make random assignments; there would then be an expectation of 10^{500} operations, but a chance that one might get the right answer first time. A more intelligent method than either of these is to assemble the data in such a way that one can work out what will clash with what first; one then uses this information to set up a skeleton program covering the items with the most constraints attached, like double periods; the lessons are then fitted round this framework in a way which preserves as much freedom as possible.

We take a specific timetabling problem of a very similar kind as an example, that of obtaining an examination timetable for Part III of the Mathematical Tripos in Cambridge. I should say that this has been worked on more than once, but a satisfactory solution has not been found.

There are six papers, some forty optional subjects, and some forty candidates. Each candidate takes about six options, and each option gives rise to

about three questions. The data consists of a list of candidates and their options. The problem is to distribute questions among the six papers in such a way that each of the candidates gets a fair deal, i.e. finds an even distribution of questions on the options he has offered. (Some small variation is allowed for any candidate on any one paper, but this can only be $+1$ or -1 from the norm.) The first program tried out made an arbitrary initial assignment; an "unsatisfaction" function was computed based on the number of clashes, and attempts were made to descend this by making interchanges. We found however, that the procedure for finding the shortest descent presented too many problems, and therefore abandoned the attempt to program the timetabling this way.

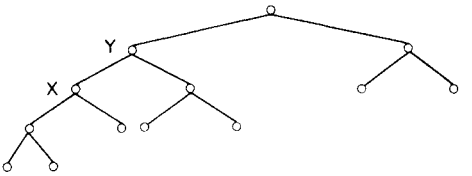


FIG. 3.

An alternative would be to use a choice tree method,* in such a way that if we find that we are going down a blind alley we cut off the branch concerned and never choose a route which would lead to it. In this technique there will be a node for each option, and the number of branches will be the number of ways of assigning the option's questions—usually $\binom{6}{3}$.

We assign the first option: the most obvious assignment is to the first three papers; we take the next option and look at the best way of assigning it, which will probably be to the first three papers too. We go on to consider each option in turn and try to find the best assignment for it. As we will soon get clashes, we will have to consider other combinations of options. A clash represents a block in a particular branch in the tree, from which we have to backtrack; each time we backtrack we abandon the part of the tree below the point to which we backtrack. If we backtrack from X to Y in the tree of Fig. 3, for example, we abandon the branch from X downwards.

The trouble with this procedure is that large quantities of red tape are involved because we have to record our route in case we need to backtrack. This means that we have to give some thought to an efficient and economical method of storing the information concerned. Also we need some heuristic to guide the assignment when more than one possibility is consistent with the rules.

* This method was used by J. H. Matthewman in a dissertation for the Diploma in Numerical Analysis and Automatic Computing.

Now it is clear that the effort involved in this procedure will be affected by the order in which the options are presented, and we have therefore to choose a suitable order. We may, for example, take the ones with the least number of candidates attached, or those with the most candidates. In the experiments we carried out we assigned the ones with the most clashes first, ran the computer for fifteen hours and gave up.

What in fact was wrong with the approach? It looks all right. If we use a tentative assignment and descent procedure, we may hit a local minimum; but as we give ourselves a chance to make an interchange at any stage we can continue for some time from a given starting point. In this procedure, however, we will never be held up by a local minimum but we may be on the wrong track because we made a wrong assignment at a very early stage: we will nevertheless wander about the lower branches of the tree for a long time before finding that the route is a mistaken one and go right back to the beginning and start with a new route from the top of the tree.

Now the candidates for Part III tend to fall into two clusters, representing pure and applied mathematicians, though there are candidates who combine the two subjects. We tried to make use of this fact by catering for the clusters first, and then dealing with the others, but the outcome was no better. This result does, however, suggest a method of attacking this kind of problem: if we can identify strongly interconnected clusters of our "objects", we should treat them first; if we find that we cannot deal with them, we will almost certainly not be able to deal with the set as a whole.

9. MACHINE DESIGN

As I said earlier, timetable construction is a typical member of a class of computer problems; we can now consider another problem in the light of the previous discussion. This is the assignment of machine parts in computer design.

The specific problem is as follows. We have a number of packages, which are plugged into an end board, where wiring connections between them are set up. We can represent this, with the packages seen endways, as shown in Fig. 4.

The question is whether this logical structure can be set up by programs. We may not be able to do all of it, but we would like to do as much as possible.

The wiring structure, which was formerly specified by listing wires, is now specified by listing the pins to be visited by each signal. This means that the actual wiring has to be fixed separately, the only constraint being to use the least possible wire. The problem, therefore, is to position the packages: we have to arrange them so as to obtain a desirable wiring pattern, where a

desirable pattern is one with as many short wires as possible, providing they are not too dense at any point and that there are not too many parallel wires, and so on. The essential problem is thus the same as the previous ones, namely that of optimizing over a large number of permutations, given that we cannot examine them all exhaustively.

We have, therefore, to solve a minimization problem (of the length of wires) for any positioning of the packages in order to assess the merit of that positioning. It is thus necessary to find the minimum over all positionings of the minimum of wire-lengths for each position.

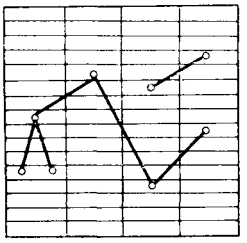


FIG. 4.

One way out of this difficulty could be to find clusters of logical elements which have strong mutual interconnections and should therefore be in the same area. We can start with a list of sets of interconnected elements, for example

47, 231, 63
48, 230
49, 227, 85, 92
.
.
.

and then find clusters of items which are connected most to each other and least to outside items. There is no doubt that if we can do this, it will make the actual wiring assignment easier.

10. CLUSTERS

Now we might, as we saw, try to find clusters in constructing timetables. The same problem of finding clusters also comes up in information retrieval. In information retrieval we want to combine the advantages of words or uniterms, which are easily picked out of the documents, and descriptors, which are more efficient as a means of specifying documents. Suppose, therefore, that we obtain measures of the extent to which pairs of uniterms

co-occur over our set of documents, and then find clusters of terms to which we can give names and use as descriptors.

The first stage in this classification procedure is thus to define the co-occurrence, or similarity, of a pair of terms. One very simple coefficient which can be used is

$$\frac{\text{number of times two items co-occur}}{\text{total number of different occurrences}}.$$

In Boolean terms this is the number of members of a set intersection divided by the number of members of a set union.

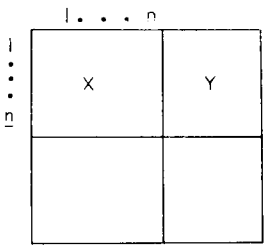


FIG. 5.

The next stage is to find clusters. It is clear that this can be treated as a general problem, independent of any particular application. In the same way, the co-occurrence coefficient just given can be used as a measure of the similarity or co-occurrence or connection of any objects which we have described by a set of properties. In my view there is a great deal to be gained from having a general-purpose classification program which can be applied to any data which can be presented in the form of an object-property incidence array. I shall therefore go on to consider classification in general, assuming that we have set up a matrix giving the similarities or connections between our objects.

What we want is a definition of a cluster, and a procedure for finding clusters in a given set of objects. Intuitively, all we can say about a cluster is that the elements should be more connected to each other than to outsiders. This gives us something to go on, but in no sense constitutes a definition. The real difficulty in fact is to find a programmable definition of a cluster, that is one which both corresponds to our intuitive interpretation and also leads to a computable algorithm. These two requirements are virtually incompatible: the most obvious definitions turn out to be very unsatisfactory algorithmically.

Let us consider some possible definitions. I shall discuss them in terms of the matrix, assuming that we want to decide whether the objects represented by the rows $1 - n$ in the matrix of Fig. 5 form a cluster (as the matrix is symmetrical we need only consider the two areas X and Y).

(a) As our first definition we will say that all the items in a cluster must be connected to each other equally, for instance by saying that they must all have a similarity to each other above a threshold θ. In our matrix this will mean that the area marked X must be all 1's. Clusters on this definition are easily found, but the definition is nevertheless unsatisfactory: if there is a single 0 in X the objects concerned will not form a cluster, even if Y consists only of 0's. The definition is such, that is, that a set of objects will not form a cluster, even if they have no outside connections, if they are not maximally connected to each other. It is, however, very likely that aberrant 0's will

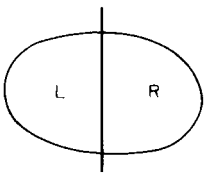


FIG. 6.

occur in genuine empirical material, if only because mistakes were made in collecting or encoding it. The point is that we cannot reasonably expect that there will be a maximum interconnection between objects like computer wires. Why should there be?

(b) An obvious definition to try is that which requires that Y shall be all 0's, but does not require that X be all 1's. But if there is an odd 1 in Y we will not get X as a cluster, and this is just as likely to happen as an odd 0 in X. What we want is a definition which really works in real applications where the data is full of irregularities.

(c) I shall now describe a definition which I have tried in a number of fields, and have so far found to be satisfactory where the two previous definitions were unsatisfactory. To refer back to the matrix, the definition requires that there be more 1's than 0's in the left half of each row in the upper part of the matrix, that is, in X, and more 0's than 1's in the right half of each row, that is, in Y. In the light of the previous discussion this definition has an obvious rationale; it is also the case that we can set up algorithms for finding these clusters.

Suppose that we imagine that our universe of objects is partitioned into L and R as shown in Fig. 6.

If the objects on one side of the partition, say L, are to form a cluster, the partition represents a local minimum of the links crossing the boundary. We can define the quantity to be minimized as the cohesion C in the form

$$C = \frac{C_{LR}}{C_{LL} + C_{RR}}$$

The obvious way to set about clump-finding is to take a partition and decrease the cohesion across it by throwing items from one side to the other. It is unfortunately the case that a partition in which one side is null has a local minimum, but the discovery procedure can be arranged so that we do not get this result. We could also avoid this situation by defining C as

$$C = \frac{C_{LR}}{(C_{LL}C_{RR})^{\frac{1}{2}}}$$

This discussion has necessarily been brief, and I have omitted a number of relevant details. I hope, however, that I have given some idea of how one might go about finding clusters in practice. In order to fill in the outlines a little I shall now consider an alternative approach to the one described, which I am currently investigating.

We can regard our similarity matrix as the matrix of a connected graph (if it falls apart, we can treat the pieces separately), or more strictly, as the matrix of a not very connected graph. What we want to find are, so to speak, lumps of graph which are only weakly connected to other lumps.

This is a useful way of looking at our problem, because it leads to a possible technique for finding clusters: there are algorithms available for discovering the shortest route between two points in a graph, which we can use as follows. Suppose that our graph in fact contains two clusters of about equal size. Now suppose that we take an arbitrary set of point pairs and find the shortest routes between them. We can expect that 50% of the point pairs will be divided between our two lumps, A and B, so that the routes between them will pass from A to B. We can keep a record of the routes we follow, and if we find a link which comes very often this will be a connecting link between A and B. We can, moreover, improve on this basic procedure: we can, for instance, start not with an arbitrary point pair, but with a random point, find the point which is furthest from it, and then take the shortest route from the one to the other. This route will more probably pass through the 'isthmus' between the two clusters than the routes between point pairs.

It might be thought that the work on clustering that I have described suffers from being too *ad hoc*, and lacks a sufficiently developed theoretical basis. In answer I should say that this is the kind of field where research is bound to be rather untidy; I also believe that the work on classification should concentrate on practical experiments rather than theory, because this is the only way we can learn anything really useful about classification in the cases where we have to use computers, namely those in which we have a large number of objects which do not show any very obvious grouping patterns. I have in any case tried to construct a general classification procedure: the method described has been applied not only to information retrieval and logical design, but also to some thesaurus-type semantic material, American

Indian tribes, medical patients and archaeological pots. Research in non-numerical applications of computers is still at a comparatively early stage, and I think that the real line of progress lies in applying general-purpose procedures in many different fields, in order to gain experience.

11. REFERENCES

CROS, R. C., GARDIN, J. C. and LÉVY, F. (1964) *L'Automatisation des Recherches Documentaires. Un Modèle Général: Le SYNTOL*, Paris: Gauthier-Villars.

SALTON, G. (1961–5) Scientific Reports on *Information Storage and Retrieval*, The Computation Laboratory, Harvard University.

TAUBE, M., GULL, C. D. and WACHTEL, I. S. (1952) "Unit Terms in Coordinate Indexing", *American Documentation*, **3**, 213.