Automatic computing: its problems and prizes

By Stanley Gill*

This is the text of a lecture delivered at Imperial College, London, on 26 January 1965, to mark the inauguration of the author as Professor of Computing Science. It illustrates the kind of thinking that is involved in the development of automatic computing as a new technology, and discusses its significance in the wider social context. Computing is compared with printing and with telecommunications as one of the forces that shape society; it poses economic problems as well as some very complex technical ones.

The speed of computers

My subject is automatic computing, which, as you all know, we can now do rather fast. This is perhaps a trite observation, but I can find no other starting point for my lecture. Speed of operation is the one basic achievement on which all the great developments of the last two decades in automatic computing have rested. We can now multiply two long numbers, of as many as twelve digits each, in the time taken by a rifle bullet to travel about a tenth of an inch. This speed in itself may not be very exciting, but whenever you get such an immense change in a capability you must look for the possibility of some qualitative effects. Take travel, for instance. Over a century and a half we have progressed from horseback and horse-drawn carriages to railways, cars and aeroplanes; a speed increase of perhaps fifty times. This as you know has had a certain qualitative effect on people's lives. But in computing we are dealing with a factor, not of fifty, but of a million.

Let us look at two other fields where similar increases have occurred: printing and communication. **Fig. 1** shows an early printing press which, I would guess, was capable of printing about 10,000 words per hour. **Fig. 2** shows its modern equivalent, the machine which prints *The Times*, and which is capable of printing something of the order of 10^{10} words per hour, and which is therefore about a million times faster than its predecessor.

Fig. 3 shows an early electrical telegraph operator who could transmit perhaps 200 words per hour. **Fig. 4** shows the control room at Goonhilly Down, where the satellite communication channel could handle over 10^{10} words per hour if it were all used for telegraphy. This therefore represents a speed increase of the order of 100 million. As we all know, printing and telecommunications have both had a tremendous effect on our society.

Fig. 5 shows a desk calculator of the kind that was in common use in the 1930's and with which one could perform two or three hundred arithmetical operations per hour. **Fig. 6** shows an electronic computer of a kind that is capable of doing several hundred million operations per hour and is therefore a million times faster than the earlier machines. The computer shown is the one that is now installed here in the College. In the centre can be seen the operator's control desk, and on the left

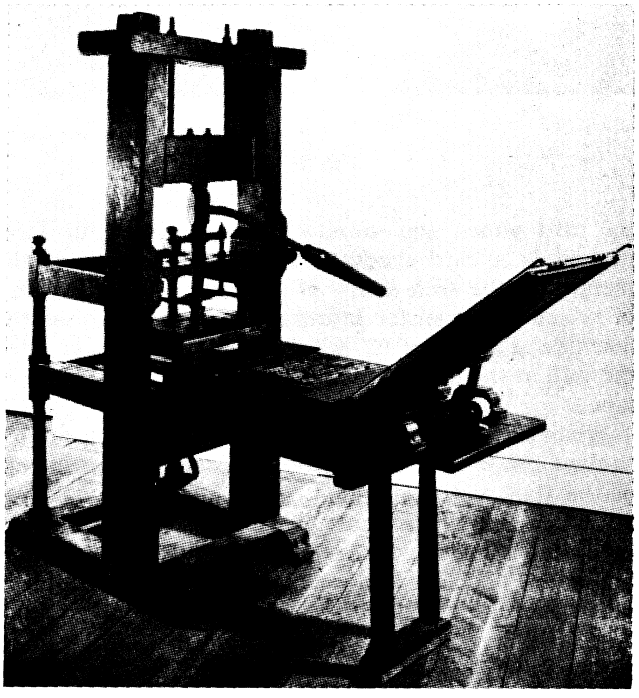


Fig. 1

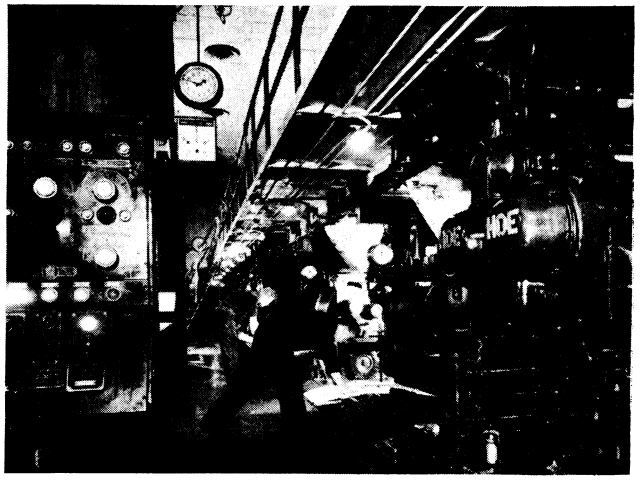


Fig. 2 *By courtesy of 'The Times'*

* *Professor of Computing Science, Dept. of Electrical Engineering, Imperial College, London, S.W.7.*

By courtesy of H.M. Postmaster-General

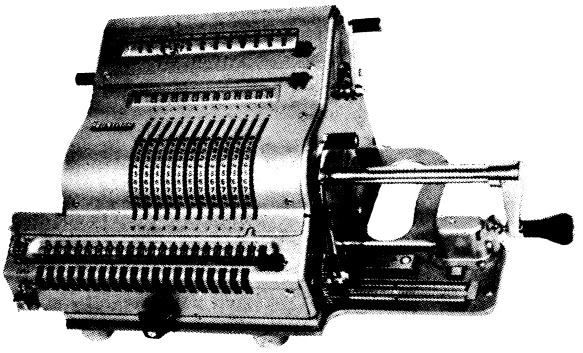
Fig. 3

the card punch and monitoring printer. In the foreground is a card reader and across the back of the picture can be seen a row of magnetic-tape units, each of which can transfer information into or out of the machine at a rate of 62,000 characters per second. On the wall at the back is racking holding reels of magnetic tape.

When this computer is in full use, if the time available to the College were shared equally between all members of the staff and postgraduate students, they would get rather more than one minute per week each. In that time each man could have several million arithmetical operations done for him each week. The cost of this is rather more than £1 per week per head, although most

By courtesy of H.M. Postmaster-General

Fig. 4



By courtesy of Olympia Werke A.G., Wilhelmshaven

Fig. 5

of this cost is that of the equipment itself, which is on free loan to the College from IBM, and we are indeed grateful for their generosity. The cost to the College works out at around 1 micro-shilling per arithmetical operation.

The mind tends to boggle at the idea of millions of arithmetical operations. However, one of our troubles is that some people don't boggle very well, and it doesn't really take much imagination to think of a calculation involving a billion steps, if you don't care how useless it is. This is an eternal problem in all computing centres. We who staff the centre cannot easily assess the soundness of every job that comes our way, and we must rely largely on the cooperation of all Departments in ensuring that jobs are worthwhile and well planned. This means that we would like to see, in all Departments, people who are familiar with computing as well as with their own subjects.

The nature of computing

Electronic computers, then, are revolutionary machines, but what is the nature of this revolution? It has something in common with the examples which I showed earlier, of printing and communications. They all deal with symbols or with language, with the very things, in fact, to which we owe our intellectual superiority over

Fig. 6

other species. Printing and communications enable us to copy symbols or to reproduce messages, either in a permanent form on paper, or rapidly in another place many miles away. Computing is concerned not merely with copying but with transforming, rearranging and replacing symbols according to given rules. The point is that if the language is systematic and the rules are right, you can to some extent replace intuitive thought by routine operations on symbols. If, for example, you ask a young child what number when added to three gives eight, he will think for a moment and perhaps conduct a few mental experiments. If he obtains the right answer, it will be by an intuitive process leading him to the right experiment. If you ask an older child this same question, he will perhaps first write down the equation $3 + x = 8$, and then, following a rule he learnt at school, but without stopping to think of the justification, he will write the equation as $x = 8 - 3$ and then as $x = 5$, and so obtain the answer. The steps from the first equation to the second and from the second to the third are examples of computational processes. Any application of rules to deduce new facts from given ones, without the guidance of intuition, is computation. It is clearly a process of fundamental importance. (Intuition may of course be required in framing the rules, but not in their application.)

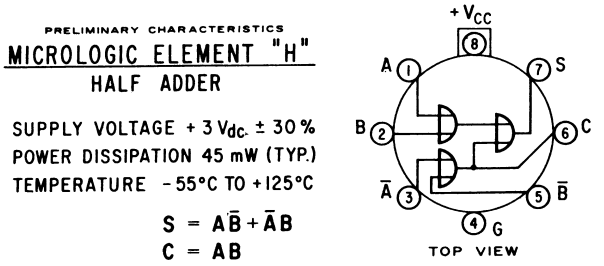
In a few years we have achieved a millionfold increase in the speed of computing, or what is perhaps more significant, a five-thousandfold decrease in the cost. (There are of course some situations in which the speed is valuable in itself as, for example, in the analysis of census returns or in the automatic control of industrial processes.) Given such a technical advance, what would we expect to see happen?

First, we would expect to see the immediate application of the new machines to jobs that are already formalized and for which computing procedures are known. Such applications are mostly found in scientific computing and in engineering design calculations. Secondly, we would expect people to work out explicit rules for doing calculations where the rules were formerly untidy and ambiguous, such as in business data processing. Both these developments have in fact occurred, and I cannot possibly list here the many applications which have already been made of electronic computers; the list is almost endless.

Thirdly, one would expect people to look for jobs where computation might provide a good alternative to present methods of inspired guesswork, such as in business planning. This is now happening, and business planning is becoming one of the most profitable areas for the application of computers.

Fourthly, one would expect computers to be used as essential parts of systems designed to do things that simply could not have been done without computers. It can hardly be said that such projects are yet being undertaken, but no doubt they will come in the near future.

There has in fact been a very rapid spread in the use



By courtesy of SGS–Fairchild Ltd.

Fig. 7

of computers, starting in about 1950. For several years the number in use doubled every year. The number is no longer doubling annually but it is still growing very fast indeed. There are now something like fifty thousand of them in the world, effectively equivalent to between 10^8 and 10^{10} people. Almost certainly more computing is now done by machines than by men (and, incidentally, between 80% and 90% of these machines are in the USA).

What is automatic computing like? How does one achieve it? First, one begins with a large number of elementary electronic circuits, each of which performs a very simple operation on one or two input signals, to give an output signal. Right away I must make a distinction between the two types of circuit that exist, one of which handles continuously changing signals and the other of which handles discrete signals that have only a limited number of meaningful states. It is not easy to mix these two types of circuit. The former can be used to make so-called analogue computers and the latter to make digital computers, which are more expensive, but enormously more versatile. Without wishing to offend those of my friends who work with analogue machines, I am going to talk here only about digital machines.

Fig. 7 shows the specification of a typical digital circuit element. Like practically all digital elements, it works on binary or two-state signals. This circuit element takes two input signals and their inverses, each in binary form, and produces two separate output signals. The relation of the output signals to the input signals is indicated in the specification by a Boolean expression.

Starting with such elements as this, we need to finish up with a system that can accept the data for some specific problem and do a lengthy calculation on it. **Fig. 8** is an excerpt from the specification of a large calculation showing how the data should be prepared for it.

NOTES ON INPUT

1. Regions must be identified as follows:

MODERATOR	
FUEL	
CONTROL	(for control rod cross sections)
SIDE	(for side reflector cross sections)
END	(for end reflector cross sections)

   Only one type of the above regions may be given per case (e.g., two 'MODERATOR' regions may not be specified); thus the maximum number of regions per case is five. Regions may be entered in any order with the first region entered on page 38. There are additional pages, 39, available for entering more than one region. A 'MODERATOR' region must be specified.
2. The same temperature must be given for the 'MODERATOR' region and the 'FUEL' region (if both regions are specified for one case).
3. Materials must be identified by 'designation' or 'formula'. Please see an up-to-date printout of the nuclear data tape for designations and formulae of materials. Materials may be listed in any order and may be repeated in the same region. No more than 12 materials may be given in one region.
4. To compute W for a given material, give a nonzero value for N (or f) and specify $W=0$. Correspondingly, to compute N (or f), give N (or f) $=0$ and give a nonzero value for W.
5. To compute g factors by program I_2, give $g=0$ for all materials in the 'FUEL' region. (If a value of $g\neq 0$ is given for any material in the 'FUEL' region, program I_2 will not be used.) For regions other than 'FUEL', $g=1$ should be given. Program I_2 requires a three-zone cell ($D_3>D_2>D_1>0$). When g factors are to be computed and if $D_3=D_2$ or $D_1=0$, it will be necessary to specify a false, thin moderator zone ($D_3=D_2+10^{-5}$ or $D_1=10^{-5}$ should be satisfactory).
6. If all g factors are unity (as in a homogeneous composition), only a 'MODERATOR' region is needed, and all geometrical input except H, D, S_r, and S_z should be zero.

By courtesy of General Electric

Fig. 8

The ability to accept such data as these, and to deal with them, seems far removed from the elementary operations performed by the individual circuit elements. This gap is bridged in stages.

First, the circuit elements are used to build up various major parts of computers, such as registers for holding single numbers; arithmetic circuits, for deriving new numbers from given ones according to the rules of arithmetic; stores, for holding many numbers and reproducing any of them on demand (this calls for special forms of hardware); control units, for directing the operations of other units; and so on. Secondly, these parts are assembled according to a suitable overall design, to form a computer.

But this is still only part of the story. A computer, as produced in the factory, is not immediately capable of accepting data and performing calculations like that illustrated in Fig. 8. All that a computer does is to obey very simple instructions, each defining, for example, one arithmetical operation. In fact, the specification of a computer consists largely of a list of the various instructions that it can obey. This is called its "order code" or "instruction code." **Fig. 9** shows part of the order code of a computer. (The instructions appearing here are actually concerned not with arithmetic but with choosing the sequence of steps to follow; I will come back to this point later.) There are other things about a computer that have to be specified too, so that people will know how to use it; for example, the manual controls which allow the operator to steer the computer in a general way and to cope with various kinds of failure: see for example **Fig. 10.**

LOGICAL COMPARE AND MODIFY INSTRUCTIONS

Mnemonic	Mode	Code	Effect
TRN	—	600	None
	E	602	If E $\wedge$ $C(AC)_R = 0$, then skip.
	A	604	Always skips.
	N	606	If E $\wedge$ $C(AC)_R \neq 0$, then skip.
TLN	—	601	None
	E	603	If E $\wedge$ $C(AC)_L = 0$, then skip.
	A	605	Always skips.
	N	607	If E $\wedge$ $C(AC)_L \neq 0$, then skip.
TDZ	—	630	Clear selected bits. Do not skip.
	E	632	If C(E) $\wedge$ C(AC) = 0, then clear and skip; otherwise, clear and don't skip.
	A	634	Clear selected bits and skip.
	N	636	If C(E) $\wedge$ C(AC) $\neq$ 0, then clear and skip; otherwise, clear and don't skip.
TSZ	—	631	Clear and don't skip.
	E	633	If $C(E)_S \wedge C(AC) = 0$, then clear and skip; otherwise, clear and don't skip.
	A	635	Clear and skip.
	N	637	If $C(E)_S \wedge C(AC) \neq 0$, then clear and skip; otherwise, don't skip.

By courtesy of Digital Equipment Corporation

Fig. 9

	by a factor of ten up to 340 milliseconds. The right knob is a continuously variable fine control.
POWER	Turns on power to the processor and all equipment connected to it.
MEMORY DISABLE	This switch causes the pause mode operation of memory to change to a separate read and write mode. When in single step, the memory module is free for use with multiple processors.
DATA	Up to 36 bits of information can be inserted for transfer to memory when the DEPOSIT switch is pressed. (May be examined under program control. See I/O Programming for Type 166 Processor.)
ADDRESS	There are 18 address switches which are used with various keys and switches.
Indicator Lights	**Function**
INSTRUCTION	Displays bits 0-8 of the instruction register.
AC	Displays bits 9-12 of the instruction register.
I	Displays bit 13 of the instruction register.
INDEX	Displays bits 14-17 of the instruction register.
MEMORY INDICATION	Displays the contents of the specified memory register (see EXAMINE lever switch).
PROGRAM COUNTER	Displays the contents of the program counter. When the processor stops, the program counter will be pointing to the next instruction.

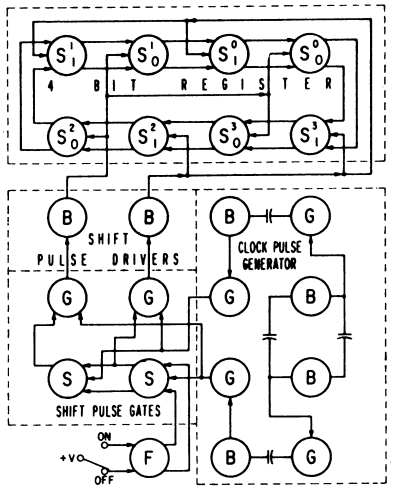
By courtesy of Digital Equipment Corporation

Fig. 10

The major part of the task of matching the computer to an actual computing job falls to the programmer, who must use the permitted instructions to compose a *program* for the job.

Thus the logical task, of making a given set of circuits perform an actual computation, falls into two main phases. In the first, the computer designer uses the circuits to make a computer. In the second, the programmer prepares a program to make the computer do the calculation. When this has been done, the data can be prepared according to the requirements of the program, and the calculation can then be done. In each phase, the work rests on specifications supplied by the previous phase and results in specifications to be used in the next stage of the work.

What does this work look like? The work of the computer designer is illustrated in **Fig. 11,** which shows part of a "logical" diagram giving the interconnections between some elementary circuits. Programming is illustrated by **Fig. 12,** which shows part of a program forming part of an information retrieval project.



By courtesy of SGS–Fairchild Ltd

Fig. 11

Programming

As you can imagine, it is not at all easy to get a big program right. One complication is that the programmer makes frequent use of the fact that the sequence in which the instructions are executed need not be the same as that in which they appear in the program. During execution, the machine can "jump" from place to place in the program, at the behest of instructions like those illustrated in Fig. 9, whose purpose this is. Obeying a program, in fact, is like playing a game of snakes and ladders.

This is illustrated by **Fig. 13,** which shows in snakes-and-ladders form a program for finding the highest common factor of two numbers, p and q. The rules for playing this game are a little different from those for ordinary snakes and ladders. Specifically:

(1) there are no dice; you move only one square at a time;
(2) you must obey the instruction in each square as you come to it;
(3) the snakes and the ladders may be conditional; i.e. they are not necessarily used every time they are encountered, but only if the instructions say so; and
(4) as normally obeyed by a computer, this is a game for one player only; the competitive element is lacking (though it is worth noting that we are now beginning to make use of computing systems in which the same program may be in process of execution in several different contexts at the same time).

This program actually expresses a simplified form of Euclid's algorithm for solving this problem (an algorithm being any strictly defined computing procedure). Let us run through this program for the case where $p = 18$ and $q = 12$. The instructions are obeyed in the following sequence.

04155	10	0	00000		ENA	O
	20	0	04343		STA	NEEDENG
04156	75	0	04153		SLJ	BACK
	50	0	00000			
04157	75	0	00000	READDL AB	SLJ	**
	74	7	00031		EXF	31B , 7
04160	74	0	32022		EXF	32022B
	74	7	32000	—	EXF	32000B , 7
04161	10	0	04536		ENA	DLABEL + 4
	20	0	00003		STA	3
04162	74	3	04532		EXF	DLABEL , 3
	75	0	04157		SLJ	READDL AB
04163	75	0	00000	READEL AB	SLJ	**
	74	7	00051	—	EXF	51B , 7
04164	74	0	52032		EXF	52032B
	74	7	52000	—	EXF	52000B , 7
04165	10	0	04532		ENA	LABEL + 4
	20	0	00005		STA	5
04166	74	5	04526		EXF	LABEL , 5
	75	0	04163		SLJ	READEL AB
04167	75	0	00000	BCDTEST3	SLJ	**
	74	7	32000	—	EXF	32000B , 7
04170	74	7	32003	+	EXF	32003B , 7
	75	4	04611		RTJ	FIXUP3
04171	74	7	32005	+	EXF	32005B , 7
	75	4	04611		RTJ	FIXUP3
04172	75	0	04167	+	SLJ	BCDTEST3
	50	0	00000			

By courtesy of the Office of Naval Research, American Embassy

Fig. 12

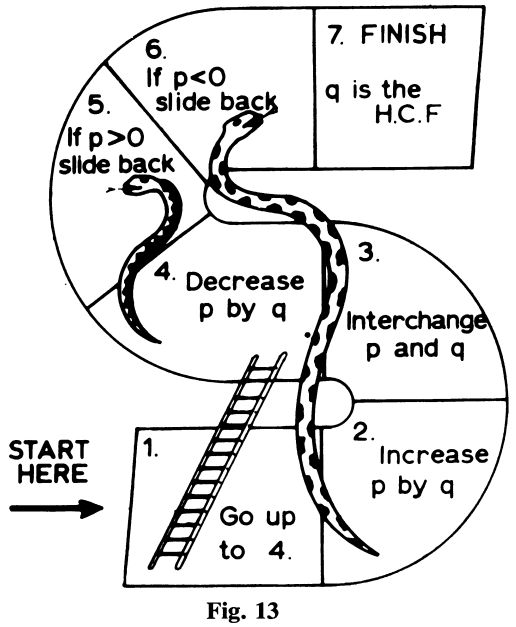


Fig. 13

INSTRUCTION NUMBER	RESULTING VALUE OF p	RESULTING VALUE OF q	COMMENTS
1	18	12	Go up ladder
4	6	12	p decreased by 12
5	6	12	p positive; slide back
4	−6	12	p decreased by 12
5	−6	12	p negative; ignore snake
6	−6	12	p negative; back to 2

2	6	12	p increased by 12
3	12	6	p and q interchanged
4	6	6	p decreased by 6
5	6	6	p positive; slide back
4	0	6	p decreased by 6
5	0	6	p not positive; go on
6	0	6	p not negative; go on
7	0	6	Finish: result $= q = 6$

This example illustrates clearly the rather tedious way in which a computer may have to proceed through even a simple calculation. The above manoeuvring looks positively ungainly; a human could obviously short-cut it quite a bit. But this procedure has the great virtue that it is completely automatic, assuming that one can mechanize each individual step. And the speed of modern computers is such that *all* the above steps can be performed in less than 0·0001 second. With such speed provided, one can afford to put up with a rather pedantic approach to problems.

I said just now, "assuming that one can mechanize each individual step." Of course the repertoire of instructions in a computer is limited. Most computers would not be able, for example, to interchange p and q in a single step. It would then be necessary instead to use three instructions, and to introduce a third variable, r say, that would act as a kind of "parking space" for one of the values during the interchange. The three steps could then be as follows:

$$\text{Set } r = p$$
$$\text{Set } p = q$$
$$\text{Set } q = r$$

This is an example of programming detail. Programming also has its architecture, and **Fig. 14** shows the "flow chart" of a program. This indicates in general terms the various groups of instructions that comprise a program; the arrows show how the attention of the computer passes from one section to another.

The idea of a flow chart can also be useful on a much smaller scale. **Fig. 15** is part of a flow chart drawn by a computer designer to show how the machine proceeds during the execution of a single instruction.

How are we to characterize this work? At the basic level, it is clearly part of electronic engineering. Yet even at this level it is often not specifically electronic in nature. The emphasis throughout is less on the properties of circuit elements than on the logic of their interconnection. Nevertheless it is still engineering; we are using scientific knowledge and methods to meet practical needs.

In the same sense, programming is also a kind of engineering, although the scientific knowledge involved is purely logical rather than material. It is, if you like, "mathematical engineering."

Let us look at some of the techniques of this activity. The most basic technique in programming is the use of *subroutines*. A subroutine is simply a group of instruc-

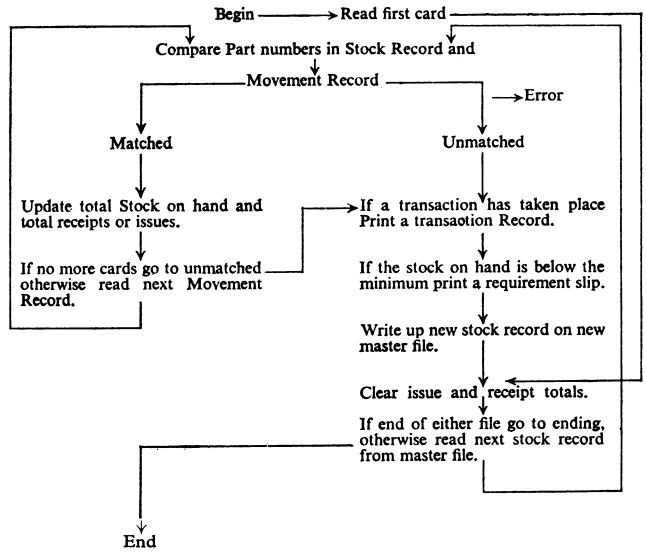
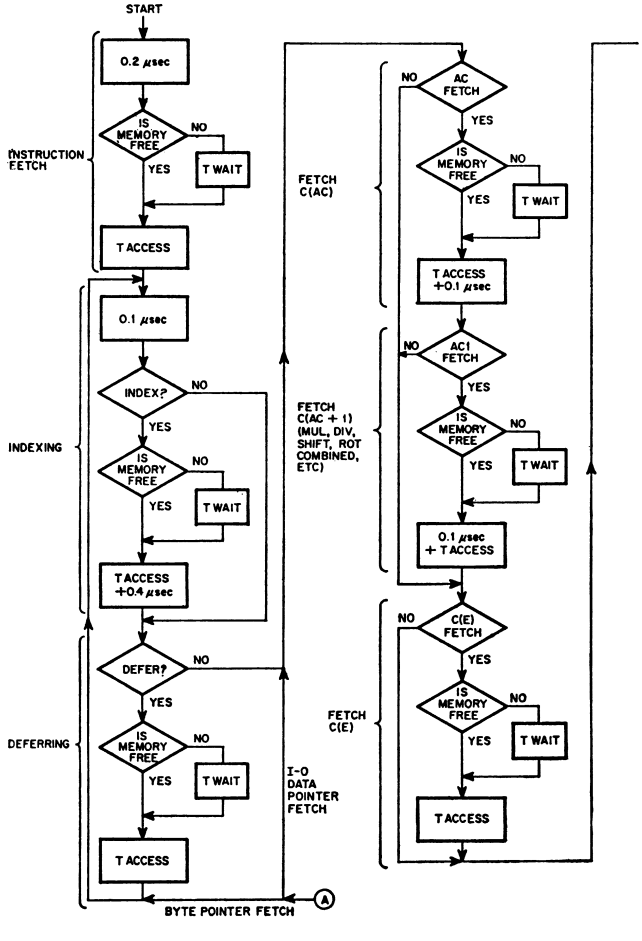


Fig. 14



By courtesy of Digital Equipment Corporation

Fig. 15

tions telling the machine how to perform some clearly defined action; the subroutine can be filed, and used whenever this action is required. For example:

Subroutine: *Interchange x and y*

Set $z = x$
Set $x = y$
Set $y = z$.

The idea is simple enough, but in practice there are many complexities. We shall look at just one of these, the problem of keeping "links."

Often the same subroutine is required to be used at several different points in a program. If it is a big subroutine, it is wasteful to occupy space in the store of the computer with several copies of the subroutine. One is enough, provided it can somehow be arranged that the machine's attention is switched to the subroutine when required, and is switched back again to the right point in the program when the subroutine has been executed, as shown schematically in **Fig. 16**. But this means that, while the subroutine is being executed, a record must be held, in some standard place, showing the point that has been reached in the execution of the main part of the program, so that the machine will "know" where it must return after executing the subroutine. This record is known as the "link." The box representing it in Fig. 16 is actually labelled "Link etc," because in general it may be necessary to store some other information as well. Most computers have a few registers that have special properties, being particularly readily accessible or playing special roles in arithmetic. These registers may be needed by the subroutine, yet they may already contain some information that is vital to the calculation and that will be needed there afterwards. This information must therefore also be stored, along with the link, so that it can be restored when the subroutine has done its work.

Now, suppose that the subroutine itself uses another subroutine. The second link cannot be kept in the same place as the first, which is not yet finished with. There must therefore be room for two links, as shown in **Fig. 17**. In general, in a complicated program, several links may need to be stored simultaneously, and the system must allow for this.

There is, however, one special case in which entering a "sub-subroutine" does *not* demand an extra link space. This is where the call to execute the "sub-subroutine" is the final act of the subroutine (see **Fig. 18**). In this case there is no need to return to the subroutine, whose work is finished, and the sub-subroutine can instead (when it has completed its job) hand control of the machine straight back to the main program. Hence the sub-subroutine can be given the subroutine's link as its own, and there is no need for two link spaces.

This may seem a trivial technicality to include in a lecture such as this. I have picked it because, in addition to showing the kind of thing that a programmer has to worry about, it also illustrates a rather interesting

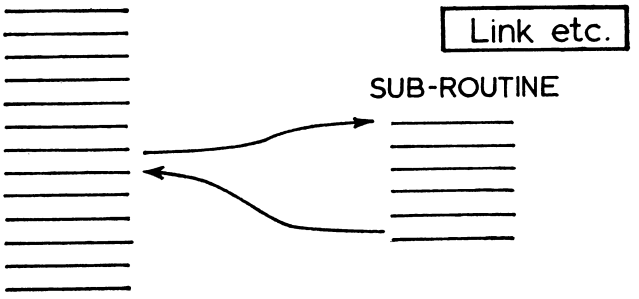


Fig. 16

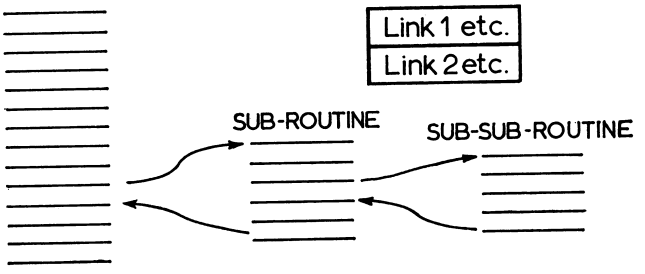


Fig. 17

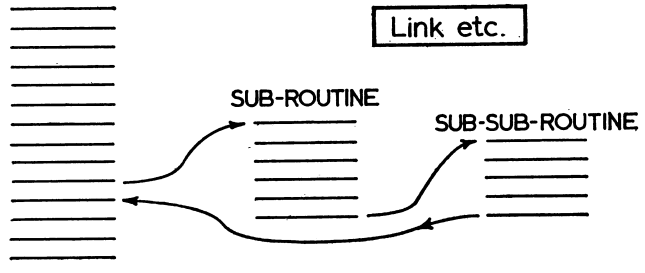


Fig. 18

analogy. This business of links, and of rescuing work-in-progress from special registers to allow a subroutine to use them, is really quite fundamental. It occurs in many forms of computing system, and also in the human brain (although we cannot say where, physically, the links are stored). The following example of an English sentence will make this apparent; for in unravelling a sentence, the brain follows a procedure which can be compared with a computer program.

Consider this sentence: "This is the cat that chased the rat that ate the malt that lay in the house that Jack built." Many people have demonstrated that this sentence can be readily understood by the average child of four.

Now consider the following: "The house in which the malt that the rat that this cat chased ate lay was built by Jack." This sentence conveys exactly the same meaning

as the one quoted earlier. It is a little more concise, containing fewer words, and also fewer subordinate clauses. Yet it is harder to follow. Why?

Let us examine its structure. In **Fig. 19** it is laid out with each clause on a different line, using arrows to show the path followed as the sentence is uttered. We can think of the brain as entering a subroutine in order to understand each clause; as it does so, it must store a link, etc., summarizing the stage it has reached so far, to be picked up again when the clause has been taken in. Owing to the deep "nesting" of clauses, one within another, three links are required.

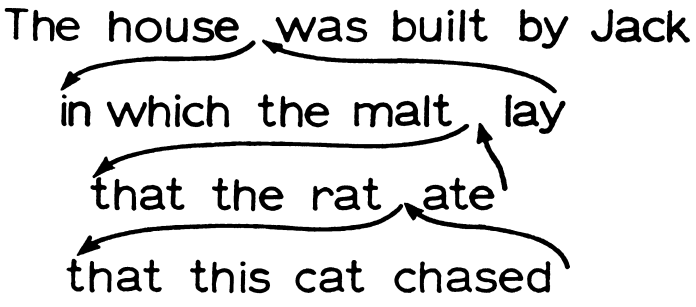


Fig. 19

Fig. 20 shows the original, more familiar sentence laid out in a similar way. Here there is an even deeper nesting of clauses, and at first sight it seems that the brain must hold four links simultaneously. A comparison with Fig. 18, however, will show that this is a case where the short-cut can be employed, because each clause forms the last component of the one above it. Hence only one link space is needed. The remarkable thing, perhaps, is that the brain learns this short-cut at the age of four!

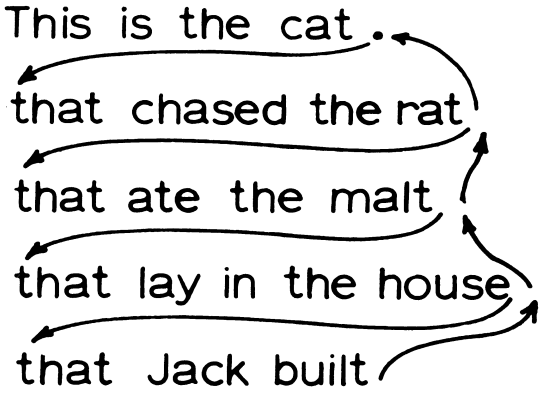


Fig. 20

The use of subroutines is only one elementary way of easing the task of programming. A much more powerful way is to write your program in a more convenient language than that of the machine itself, and then to get it translated into machine instructions. If the language is precisely defined—as it must be (unlike English) if it is to express algorithms unambiguously—then the translation can be done by following rules. Thus it too becomes a computation, carried out according to an algorithm, which can be expressed as a program; so the computer can translate its own programs. (A program used to translate other programs is called a *compiler*.)

The work of programming thus becomes split into two parts. The first is compiler writing, which bridges the gap between the instruction code of the computer itself, and the more convenient language in which programs are actually written; and the second is the writing of programs for actual jobs. The compiler writer takes as his starting point the given instruction code of the computer; using this, he aims to write a compiler that meets a target specification which lays down the language that is to be accepted and translated by the compiler. **Fig. 21** shows part of a specification of such a programming language.

This specification is in turn the starting point for other programmers, who use the language to write programs for actual jobs. **Fig. 22** shows part of a program written in an international standard programming language called "ALGOL." It may look cryptic to the novice, but an equivalent program written in the machine's own instruction code would look even more cryptic, and a great deal longer.

A good compiler will do more than translate programs; it will also help the user in finding his mistakes. **Fig. 23** is another excerpt from the same specification illustrated in Fig. 21, showing some of the diagnostic information that the compiler provides.

The arithmetical instructions

The basic form of the instructions for computing variables may be illustrated by the following example:

$$y = 2mn\, a_{(m+1)} + a_m n + ma_n + 0{\cdot}01m + 0{\cdot}01n$$

which gives the new value of the variable to be altered (in this case y) in terms of other quantities.

In general the right hand side may involve any number of products which may each have any number of factors either variables, indices, or constants. As already mentioned it is necessary to distinguish in the 'one-dimensional' form

$$y = 2\,mna(m+1) + amn + man + 0{\cdot}01m + 0{\cdot}01n$$

between am meaning a_m and $a \times m$. The convention adopted is that an index immediately following a variable letter is treated as a suffix so that the above expression is interpreted as

$$y = 2 \times m \times n \times a_{(m+1)} + (a_m \times n) + (m \times a_n) + (0{\cdot}01 \times m) + (0{\cdot}01 \times n)$$

As a consequence of these rules, numerical factors will usually be placed at the beginning of a product.

Further examples of instructions in this general class are:—

$$a = 0 \quad x_k = x_k + 1 \quad x_n = x_o + nh \quad x = i$$

Products can also be divided by a single quantity

thus $u = x/a + y/b + z/c$

and $v = 2\pi u/n$

are possible instructions.

By courtesy of ICT Ltd.

Fig. 21

```
begin
    real bigajj;
    integer i, j, k;
    real array p, q[1: n];
    Boolean array r[1: n];
    for i:=1 step 1 until n do r[i]:=true;
grand loop:
    for i:=1 step 1 until n do
    begin
search for pivot:
        bigajj:=0;
        for j:=1 step 1 until n do
        begin
          if r[j] ∧ abs(a[j,j]) > bigajj then
          begin
            bigajj:=abs(a[j,j]);
            k:=j
          end;
        end;
        if bigajj=0 then go to fail;
```

*By courtesy of the Association for Computing Machinery, New York**

Fig. 22

IIIA. **Faults encountered during INPUT**

1. The machine may come to a 99 stop (PF = all zeros).
   CAUSE: NO matrix tape in machine OR incorrect form of instruction.
2. The machine may stop and record ACCUMULATOR OVERFLOW (the 2 most significant digits of YA are different).
   CAUSE: Incorrect form of instruction.
3. The parity light may come *ON* with PF = 0010100 . . . but NO drum selection light ON.
   CAUSE: Incorrect form of instruction OR ERROR on tape.
4. The machine may come to a *LOOP STOP* (PF = 0000001000) *IN THIS CASE, LOOK IN B7* which will give the FAULT NUMBER.
   Certain versions of AUTOCODE do not stop on encountering an input fault, but continue to translate and list all the faults discovered during the input attempt.

IIIB. **Faults encountered during OPERATION**

1. The machine may come to a loop stop (as Note 4 in IIIA). LOOK IN B7 for FAULT NUMBERS 8, 32, 33 to 35. Fault 32 can be surmounted after correcting the tape, by resetting control to 15.0.
2. The machine may record ACCUMULATOR OVERFLOW. The numbers have exceeded capacity, i.e. > 10^{77}.

IV. **Action required when the hooter sounds**

By courtesy of ICT Ltd.

Fig. 23

Manipulating symbols

It will be clear from what I have said that computing nowadays involves many things besides operations on numbers. Compilers, for example, have little to do with numbers, although the programs which they translate are usually destined to operate on numbers. In fact, many applications involve other kinds of information. In business data processing, computers must handle names and addresses, catalogue codes, and various kinds of categorization such as sex, marital status, etc. Those whose interest centres on the computing systems themselves are particularly interested in operations on strings of symbols, because most of the information entering a computer is in this form. Let us take as an example the following string.

$$(3a + \sqrt{b})/(3a + 1)$$

Now most computers work in terms of binary digits, which have only two possible values, 0 and 1. A printed symbol is coded as a set of binary digits; usually either 5, 6, 7 or 8 of them. So, using a suitable code (with say 6 binary digits per symbol), the above symbol string might appear as a long string of binary digits thus:

```
001000010011100001011101111 . . .
  (     3     a     +
```

Inside a computer, these binary digits are held in groups of a particular length, usually between 20 and 50, called *words*. It is therefore possible to pack several coded symbols side by side in a single word. Thus if there are 24 binary digits per word, we could arrange our symbols 4 per word as follows:

Word 1:	(	3	a	+
Word 2:	√	b	)	/
Word 3:	(	3	a	+
Word 4:	1	)	end	

The serial number, by which each word is identified, is called its *address*. Most computers can store several thousand words, each identified by a different address.

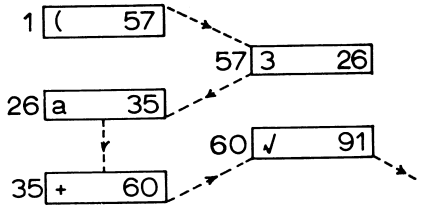
The above arrangement, with 4 symbols per word, is clearly economical in storage space. However, if we wanted to make the computer work on this symbol string, it would probably be more convenient to use a separate word for each symbol, so that individual symbols could be more quickly picked out or changed. Of course, much of each word would be left unused. The first few words would look like this:

Word 1:	(
Word 2:	3
Word 3:	a
Word 4:	+
Word 5:	√
Word 6:	b

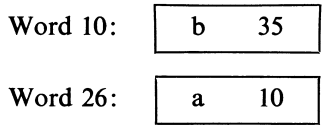
etc.

* *Part of* Algorithm 150, by H. Rutishauser, *Communications of the ACM*, Vol.6, No. 2, p. 67 (February 1963).

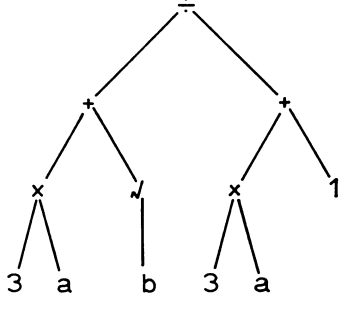
This form is convenient for many purposes, but is still not ideal if it is necessary frequently to remove or insert symbols in the middle of the string. For this, the technique of *chaining* is used. In the last example above, the symbols occupy consecutive words, and the position of a symbol in the string is defined by its position in the store. When chaining is used, the words may be scattered anywhere in the store; the proper sequence is defined by quoting, in each word, the address of its successor. Retrieving the string is like following a treasure hunt:



With this method of coding we can arrange to insert, say, a symbol "b" immediately after the "a," by taking any free word, say Word 10, and setting this and Word 26 thus:



So far I have not made any reference to any possible meaning of this symbol string. In practice, this string is obviously intended to be interpreted as an algebraic expression, and we are likely to want to make the computer do operations on it based on its structure as an algebraic expression. This structure is best depicted as an inverted tree, with the principal operator at the top:



There are ways of representing this structure fairly directly inside a computer, but before I go on to this I will make a few remarks about the relationship between the tree form of the expression and the serial string form. Most languages of any power express concepts having a fairly elaborate structure; English certainly does. Yet, for most purposes, it is necessary to express messages in the form of a string of some kind. Speech, for example, requires us to utter sentences consisting of a simply ordered set of words. In fact, transmission through any kind of channel of limited capacity forces information to be treated in the same way as meat is treated by a sausage machine. If the structure is to be reconstituted, there must be conventions (i.e. a grammar) by which the structure can be deduced from an examination of the string. The conventions are largely arbitrary; for example English and French use different word orders to express the same underlying structure. In fact in natural languages the conventions are often imperfect, so that the structure cannot always be recovered unambiguously.

It is worth noting that in biology the genetic information is also held in the form of a string, although it represents something with a very complex structure indeed. Perhaps this is because the copying processes, to which this information is repeatedly subjected, impose a bottleneck through which the information can only pass as a long string.

Returning to algebra, again there must be conventions to indicate the structure of an expression, but again these are arbitrary. Simple arithmetic operators like $+$, $-$, $\times$ are usually written between their operands; thus we write $a + b$, $5 - 2$, $a \times 3$ etc. For general mathematical functions, however, we usually put the function name in front, thus: f(x,y). We can do this for all the operators, in which case we have what is usually called the "Polish" notation, having been invented by the Polish mathematician Jan Łukasiewicz in the 1920's. In Poland it is called the "parenthesis-free" notation, because it has the property that, provided each operator has a known fixed number of operands, it is not necessary to use brackets to avoid ambiguity. Thus the foregoing example would appear in Polish notation as follows:

$$\div + \times 3a\sqrt{b} + \times 3a1$$

It is said that Łukasiewicz happened to possess a typewriter on which the brackets were in upper case and all the other symbols in lower case, and that he used this notation so that he could type expressions faster.

It is also possible to use a "reversed Polish" notation, in which every operator comes after its operands; again, brackets are unnecessary.

$$3a \times b\sqrt{} + 3a \times 1 + \div$$

This reversed Polish notation has become of very great interest to computer programmers, on account of one special property. This is that it can be used directly as a program, for a particular type of machine, to evaluate the expression represented. The machine must have what is known as a "push-down" or "nesting" accumulator. This is a register that can hold indefinitely many numbers; as each new number arrives it is placed on top of those already there, which become inaccessible until the top number is removed again. When evaluating an expression, any number that is encountered is simply placed on the top of the accumulator. If a letter is encountered the number that it represents is placed on top of the accumulator. When an operator is encount-

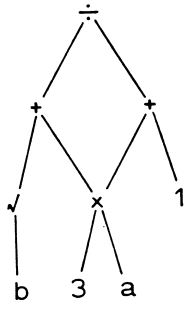
ered, the requisite number of operands is taken from the top of the accumulator, the operation is performed, and the result is put on top of the accumulator. At the end of the evaluation the top number in the accumulator will be the value of the whole expression; any initial contents which the accumulator may have had will remain unchanged, underneath.

The evaluation of the above expression, in the case where a = 2 and b = 64, would proceed as follows:

Program step:	3	a	×	b	√	+	3	a	×	1	+	÷
Contents of accumulator:	3	2 3	6	64 6	8 6	14	3 14	2 3 14	6 14	1 6 14	7 14	2

Since the reversed Polish notation gives particularly compact programs for evaluating expressions, it has been used as a basis for the instruction codes of some computers.

There is, however, one thing that cannot conveniently be done with any serial representation of an expression, but that can easily be done with the tree form. This is to avoid the need for duplicating a sub-expression that happens to be used more than once in the expression. Thus the sub-expression 3 × a occurs twice in the above example, and it has to appear twice in each of the serial forms. One result of this is that, when the expression is evaluated using the push-down accumulator, the same multiplication is done twice. The tree notation, however, avoids this duplication in a perfectly natural way:



A fairly direct representation of this tree form can be devised for use within a computer, by an extension of the "chaining" technique mentioned earlier. Each simple expression or sub-expression appears as a chain of words, each containing in its left half an element of the expression, and in its right half the address of the next word in the chain. (The last word contains here some terminating symbol, which I will write as Ø.) I will assume that the elements of a simple expression appear in this chain in Polish order, i.e. operator first, followed by its operands. If an operand is a single symbol or number, then it is recorded directly in its half-word. If, however, it is a sub-expression, then the half-word contains an address, namely the address at which a representation of the sub-expression may be found. Starting at this address, the sub-expression is stored in exactly the same way as the expression of which it forms a part. Thus either half of a word may contain an address "pointing to" some other word; it is convenient to depict this relationship by an arrow leading from the half-word containing this address to the word whose address it is. In this way the whole expression as stored in a computer may be pictured as in **Fig. 24**. This kind of arrangement is known as a "list structure," and several computer programs have been written for performing miscellaneous operations on list structures. Most of them allow for common sub-structures (like the bottom row of words in Fig. 24); another possibility, which is usually ruled out on account of the complications that can arise, is that the structure may be recursive, i.e. there may be a closed loop of arrows.

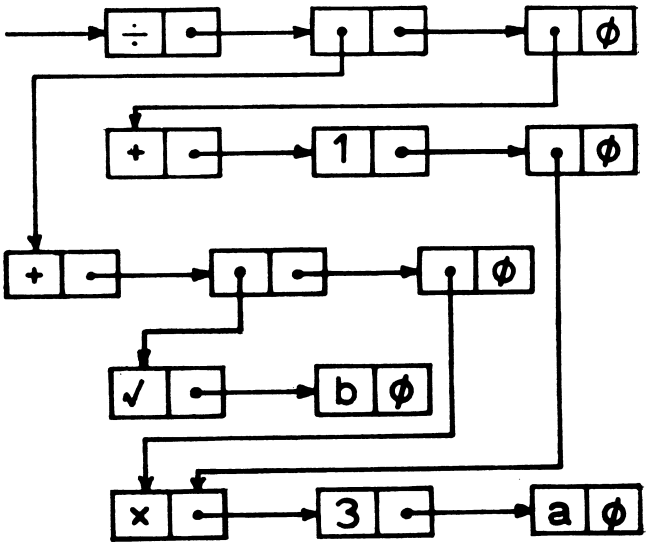


Fig. 24

As an example of the kind of processing that might be carried out by a computer on list structures, consider the formal differentiation of algebraic expressions. Let us denote the result of differentiating expression E with respect to variable x by D(E,x). E will consist of an operator, which we will denote by F[E], and one or more operands, which may be written $A_1[E]$, $A_2[E]$, . . . etc. These operands may themselves be expressions with operators and operands, or they may be "atoms" (i.e. single variables or constants). The process of differentiation may be defined in the following way:

$$D(E,x) = \begin{cases} \text{if } E = \text{"x"}: & 1 \\ \text{otherwise if E is an atom}: & 0 \\ \text{otherwise if } F[E] = \text{"+"}: & \\ \qquad D(A_1[E],x) \times D(A_2[E],x) & \\ \text{otherwise if } F[E] = \text{"−"}: & \\ \qquad D(A_1[E],x) - D(A_2[E],x) & \\ \text{otherwise if } F[E] = \text{"×"}: & \\ \qquad A_1[E] \times D(A_2[E],x) + & \\ \qquad\quad A_2[E] \times D(A_1[E],x) & \\ \ldots \text{ otherwise } \ldots\ldots \text{ etc.} & \end{cases}$$

It is possible to write an ordinary computer program to follow these rules and thus to perform differentiations. (The result may, however, bear simplification, a more subtle process which I will not go into here.) We could, however, go further and note that the above definition of the differentiation process is itself tree-like in structure, and can be considered as an algebraic expression (although its operands are formal, not numerical). By adopting suitable conventions it too can be stored in a computer as a list structure. One can then consider writing a computer program that will apply, or "evaluate," any such definition in any given context. Several such programs have been written, the most noteworthy being one known as "LISP," developed by McCarthy at M.I.T. The fact that a definition may, like the one shown above, be highly recursive, does not upset LISP, which takes recursion in its stride.

I hope that, in years to come, we shall be able to leave all our tedious algebra to computers. There will be difficulties, of course, in steering a computer through a long series of manipulations; often it is not easy to lay down in advance just what kinds of manipulation are to be performed. This problem is being tackled on two fronts. First, computers are being developed that can act immediately in response to requests made through a directly connected keyboard, and can then, without serious loss of efficiency, wait for the user to examine the result before making a further request. Efficiency of utilization of the computer is maintained by allowing several users to carry on such "conversations" with the computer at the same time through different keyboards, and by filling in any remaining gaps in the computer's time with a steady load of low-priority work.

Secondly, algorithms are being devised by which a computer can obtain on its own, results that previously required the intervention of human intelligence. This is happening, in particular, in the field of theorem-proving, which can be regarded as manipulations in the algebra of symbolic logic. To find, by random searching, a series of manipulations that will lead to a proof of a given theorem is impractical owing to the enormous number of possible steps that present themselves. Recent work has shown ways of conducting the search so as to reduce considerably the number of steps required. Clearly this is only the beginning of a very big subject, but results so far are promising, and there are good prospects of mathematicians getting practical assistance from computers sometime in the future.

At present, the main practical application of the programming techniques, like those exemplified in LISP, that involve a high degree of abstraction, is in writing compilers (the programs that translate programs into computer instructions). It is now common practice to define the syntax of the language being translated in a formal way; for example, the following is an excerpt from the formal definition of the syntax of ALGOL:

[adding operator] :: = + | −
[multiplying operator] :: = × | / | ÷
[primary] :: = [unsigned number] | [variable] |
  [function designator] | [arithmetic expression])
[factor] :: = [primary] | [factor] ↑ [primary]
[term] :: = [factor] | [term] [multiplying operator]
  [factor]
[simple arithmetic expression] :: = [term] |
  [adding operator] [term] | [simple arithmetic
  expression]
  [adding operator] [term]
[if clause] :: = **if** [Boolean expression] **then**
[arithmetic expression] :: = [simple arithmetic
  expression] |
  [if clause] [simple arithmetic expression] **else**
  [arithmetic expression]

Each name enclosed in square brackets is the name of a particular type of phrase that can occur in ALGOL. The phrase-type named on the left of each line is defined on the right by exhibiting the alternative constructions that a phrase of this type can have, separated by vertical lines. It is now a common practice to write the part of a compiler that handles the syntactic analysis in such a way that the syntax itself is embodied in statements like those above, to which the compiler refers in order to interpret a given text.

There has only been time in this lecture to dabble in a few of the tricks of modern computing. I hope, however, that I have said enough to show that computers are not only incredibly powerful tools, but also a tremendous intellectual stimulus. In striving to make computers do our reasoning for us, we are getting an entirely new insight into what "reasoning" is.

Before I end my talk, and as a change from the applications that I have been considering so far, I would like to show you a short film made at the M.I.T. Lincoln Laboratory, showing the processing of graphical information by computer. The shapes that you will see in this film are coded as sets of elements, as combinations of lines, circular arcs, etc., and stored as list structures in the computer. This is not yet a commercially viable application, nor does this film show all the tricks that are possible. I am sure that, as you watch this film, a lot of ideas will occur to you for extending the work that is shown here.

[*Professor Gill then showed the film "Sketchpad."*]

Conclusion

The title of my lecture is "Automatic computing: its problems and prizes." I hope that I have said enough about the prizes, and I have mentioned one or two of the problems. Let me finish with a few more words about the problems.

There are really two big problems in computing today. The first is a problem of economics; the problem of getting the right investment in the right thing at the right time. Free enterprise does not operate very well in the computer field, because the profit does not always accrue to the person who makes the investment. The result, especially in this country, has been very unfortunate.

There is no doubt, in my mind, that computer technology is more important for the future than aviation, atomic power, or space travel. It calls for smaller investments than these, investments measured in tens of millions rather than hundreds of millions, yet whereas we have put thousands of millions of pounds into these other kinds of research, we have probably put less than ten millions altogether into computer research. We now have a lot of leeway to make up.

The other big problem is the design problem. I am referring not only to the design of the computers themselves, the "hardware," but also to the design of the "software," the collection of standard programs—the compilers and so forth, amounting perhaps to over 100,000 instructions—needed to make computers do the kind of tricks that I have been talking about. I am referring, in fact, to the whole great edifice of "mathematical engineering" that comprises a complete computing system. This is now so vast, rambling and expensive, that it demands much more study than it has had so far. One obvious need is for more standardization at all levels. But there are so many people involved, and the repercussions of a decision so complicated, that standardization is extraordinarily difficult.

I myself would like to see more theoretical work done on this subject, and more attempts to extract general principles by studying mathematical models of certain aspects of computers. This is a subject that uses many of the methods of mathematical logic, but requires an appreciation of the practical problems of computing. There are, unfortunately, very few people looking into it in this country at present. If my arrival can do anything at all to promote the study of this subject in this College, then I shall be very pleased.

Reference

McCarthy, J. *et al.* (1962). *LISP* 1.5 *Programmers Manual*, M.I.T. Press.

Book Reviews

Time-dependent Results in Storage Theory, by N. U. Prabhu, 1965; 48 pages. (London: *Methuen & Co. Ltd.*, 8s. 6d.)

This is the first of a series, each member of which will consist of a separate off-print of a review paper published in the *Journal of Applied Probability*. The intention is to keep the standard of exposition as simple as possible but to provide up-to-date accounts of research done in particular fields. This article by Mr. Prabhu has a fairly extensive list of references and an index, and should be useful to anyone interested in storage theory.

Storage models are a particular case of what is variously described as queuing theory or inventory theory. Basically they comprise a reservoir with inputs and outputs which may be either random or controlled variables. Mr. Prabhu, working with Moran and Gani in Australia, has been particularly interested in dams. Early work in this field was concerned to extend the theory of queues, in which arrivals, service and departures are usually in discrete quantities, to the case where input at least is continuous; e.g. by rainfall. Later developments have considered extensions to inputs which are correlated over time, and to various types of controlled release from the system. Interest has also developed in various transient features, and in particular to the distribution of volume in the reservoir and of the so-called "wet periods," i.e. times during which there is anything in store available for release.

Mr. Prabhu knows his field thoroughly and has made some notable contributions himself to its development. This article effectively covers work done in the last seven or eight years. From the nature of the case, it is not a text-book and has at times to quote results without proof or to condense arguments. But it gives a very useful review of a rapidly developing subject and will be very valuable to all those interested in storage problems.

M. G. Kendall

Journal of the Institute of Mathematics and its Applications: Volume 1, *Numbers* 1, 2, edited by F. A. Goldsworthy, 1965. (London: *Academic Press*, 120s. per volume).

The *Institute of Mathematics and its Applications* was formed in 1964; of its two publications the *Journal* (which is quarterly), is devoted to research papers while the *Bulletin* (also quarterly), is principally for news. Two issues of the *Journal* have now appeared; the quality of production is very high and the printing and format pleasing. The quality of the articles fully matches this standard; the series begins appropriately with an outstanding survey/expository article on group velocity by the Institute's first President, Professor M. J. Lighthill, F.R.S.

Of the twelve papers (average length 16 pages), in the first two numbers, six are on fluid mechanics (including MHD), three on elasticity, one on electromagnetic diffraction, and two only on statistical topics (stochastic processes and curve-fitting). The concentration on classical applied mathematics seems somewhat at variance with the declared policy of treating "all areas of the application of mathematics," but it would clearly be unfair to judge from two issues and no doubt the newer applications in mathematical economics, bio-mathematics, information theory etc., will in time receive their due share of attention.

One hopes also that the editorial statement "Especially welcome will be papers which develop mathematical techniques applicable to more than one field," may give scope for articles on applicable pure mathematics or applied mathematics in the Continental sense, in which this country seems sadly backward and in which members of the *British Computer Society* might be greatly interested.

F. M. Arscott