AUTOMATIC COMPUTING

by

G.W. Hill
C.S.I.R.O., Division of Mathematical Statistics
C.S.I.R.A.C., Computation Laboratory
University of Melbourne, Victoria

and

J.G. Sanderson
Mathematical Services Group
Weapons Research Establishment
Salisbury, South Australia

PART I SIMPLE COMPILER TECHNIQUES OF C.S.I.R.A.C.

by

G.W. Hill

1. INTRODUCTION

The task of programming involves a considerable amount of tedious bookkeeping, together with a dash of ingenuity. The tedious bookkeeping may be done by the computer, so that is constructs its own program from simplified input codes, which represent a concentrated form of the user's ingenuity. Such a process of using the computer to program itself is known as "automatic programming."

Use of the computer to handle the bookkeeping reduces the number of minor errors and enables the program to be designed more rapidly. If the bookkeeping extends to the stage of remembering standard programming "grammar", the programmer may use a suitable "shorthand" language for describing the problem to the computer. Automatic programming procedures for which the "shorthand" symbols have the form similar to computer orders, may be called "compiler techniques".

The task of the user may be simplified even further if the "shorthand" is as close as practicable to conventional mathematical notation. This requires some form of translation from symbols into the form of orders, either during input or in the course of operation by use of interpretive procedures. A "symbolic translator" system will be described in Part II of this paper.

Broadly speaking, standard programming is rather tedious but capable of use in any problem, compiler techniques reduce errors and design time but are limited by the scope of the library, while symbolic translators require very little training of the user but are restricted to the symbolic system catered for. Automatic programming does not render programmers obsolete, but it does free them from tedious work for many problems and allows them to concentrate on extension of program techniques.

In this part of the paper we consider the features of initial orders and compiler routines, which, although designed for use of CSIRAC, will illustrate the factors involved in use of compiler techniques with almost any computer.

2. FEATURES OF INITIAL ORDERS

Initial orders are inserted in an empty computer in a special manner and then used to assemble the problem program from input codes recorded on punched cards, punched tape or magnetic tape. The sequence of input codes must be regrouped, decoded or otherwise modified to yield the digit train representing the desired order, which is then stored in an appropriate location in store.

It is generally convenient to provide certain "luxuries" by way of forming a directory of program parameters, which are subsequently incorporated in orders associated with special input codes. Other "luxuries" include provision for starting storage of orders from a chosen location and for performing an order instead of storing it. These features may be listed as follows:-

Essential primary operations;

1. Encoding machine orders from input code,
2. Storage of assembled orders in serial locations,

"Luxury" control operations;

3. Addition of parameter from directory.
4. Starting storage from selected location,
5. Performing the assembled order instead of storing it.

Compiler techniques may be regarded as an extension of the "luxury" control operations.

3. HUMAN COMPILATION

The human programmer transforms the problem specification into a sequence of operations, each of which can be performed by computer orders or by standard routines already designed. This sequence becomes the "master" routine in which each reference to a standard routine is replaced by a group of orders. These orders transfer the data from locations required by the master to "registers" required by the routine, organise transfer of control to the routine and back to the master, when the routine's operation is completed, and finally transfer the result from the register used by the routine to the location required by the master.

The standard routines are copied from previously tested cards or tape and the master routine is prepared on the same input medium. Control operations provided by the initial orders are used to simplify reference to locations the routines will occupy and to incorporate any program parameters required.

4. AUTOMATIC COMPILATION

To analogue most of the processes of human compilation, a compiler routine must extend the facilities of the initial orders to include:-

6. Incorporation of codes for registers involved by standard routines into partial orders supplied by the user to indicate the locations of data as implied by the master routine.
7. Forming the orders which link the routine to the master.
8. Selecting and transferring library routines from the backing store to the problem program.

9. Incorporating parameters in library orders in the process of transferring library routines.

This may be achieved by two compiler routines. The first completes the skeleton data transfer orders supplied by the user by adding the register codes stored preceding the routine in the library. It also lists the locations of orders which will transfer control to the library routines. When the master program is stored, the second compiler routine works through the list, using a library directory to select the required routine and stores the orders following the master routine. When the computer "knows" where the routine is in the problem program, it forms and stores the appropriate order in the master to shift control to the transferred routine.

5. COMPLETION OF DATA-TRANSFER ORDERS

For the CSIRAC system the programmer supplies the following input codes to cater for a "three-address" operation of the form:-

$z' = f(x,y)$ e.g. $z = x/y$

(x)	--- 0	Transfer data from location x to register ?.
(y)	--- 0	Transfer data from location y to register ?.
(S)	--- D_0	Store current order address in "link" register.
	m E.	Use the m-th library routine.
(0)	--- z	Transfer result from register ? to location z.

"E" is an input code, detected by the initial orders, which calls in the first compiler routine. This uses "m" to look up the m-th routine in the library directory, which lists the locations of the first order of each routine. From the location preceding the first order is extracted library "data", which contains the codes of the registers required by the routine. These are added to the skeleton outlined above to yield:-

(x)	--- R_x	Library data is stored in the form
(y)	--- R_y	
(S)	--- D_L	L, R_y, R_z, R_x.
	m	The address of this location is placed at the end of the transfer directory.
(R_z)	--- z	

In the case of operations which require fewer registers, the corresponding codes are zero so that they do not affect orders to which they are added. Thus the programmer omits the order (x) --- 0 in the case of the function sine $y = z'$, and the code R_x is zero and thus does not change the order to which it is added.

This may be extended in a fairly obvious way to cater for "single-address" computers in which two orders are required to transfer one datum through the Accumulator to the "register" of the routine.

6. TRANSFER OF LIBRARY ROUTINES

When the master routine is completed in the manner outlined above, the second compiler routine proceeds to work through the transfer directory list of locations of the various "m"s in the master routine. It extracts each

"m" and uses the library directory to locate the routine and proceeds to transfer it, order by order, to locations following the master routine. The address, "h", of the location, in which the first order of the routine is stored, is now used to construct the order "h --- S" for shifting control to the routine and this order is inserted in the master routine to replace the corresponding "m".

At the same time, the address "h", suitably tagged, replaces the m-th entry in the library directory. Should the master routine refer to the m-th routine again, the tag is detected and instead of transferring the m-th routine again, the "h" stored in the library directory is used to form the sequence shift "h --- S", which then replaces the corresponding "m" in the master routine.

7. INCORPORATION OF PARAMETERS DURING TRANSFER

The "grammar" of routines is such that all orders requiring incorporation of parameters have the form of a transfer of numbers to and from locations (as distinct from registers) or a transfer to control inside the routine. These may be detected during transfer from the library and given special treatment.

Such orders are "tagged" in the upper three digits of the address to indicate which of eight parameters is to be added. Every such order tagged with 0 will have "h" added to it, so that the resultant order refers correctly to locations in the routine in its new position in the compiled program. The user inserts other parameters immediately following the input code, "m E". These parameters are stored in the transfer directory, whence they are added to orders tagged with 1, 2, 3, etc.

8. OTHER FEATURES OF CSIRAC COMPILERS

The CSIRAC system caters for library routines, which use other library routines as subordinates. During transfer of routines from the library, any reference to a subordinate routine is detected by its "grammatical" form:-

$$(S) \underset{m}{---} D_0$$

Store address of next order!
Compiler code for m-th routine.

The address of the location in which "m" is stored is included in the transfer directory, so that the subordinate routine will be transferred just as though it were subordinate to the master routine.

Compiler techniques have been applied to the CSIRAC system of hyper-programmes, which use interpretive routines to provide elaborate operations corresponding to each hyper-order. The main differences arise from the different "grammar" appropriate to single-address hyper-orders and from the different process for linking subordinate hyper-routines to the master hyper-routine.

9. EXTENSIONS TO OBTAIN EFFICIENT PROGRAMMES

The CSIRAC system will be extended to improve the speed of compiled programmes.

The task of summing the contents of locations 100, 101, 102, ... 199, may be performed by a routine of 101 orders or by a repeated loop routine only 8 orders long. The second process is five times slower than the first. If this process occurred in the innermost loop of a looped programme, it would be considerably economical to use the 101 order routine, provided it would fit in the store. It is proposed to store routines for repetitive operations, such as polynomial evaluation, in such a form that the compiler "unrolls" them in the process of transferring them into the compiled program.

For similar reasons it is proposed to form a separate transfer directory corresponding to those routines referred to in the innermost loop of the master routine. After the routines corresponding to the normal directory have been transferred, the compiler will scan the other directory to determine how much store would be free if the most compact routines were selected from the first section of the library. It will then proceed to allot the remaining store space by choosing speedier and bulkier routines from the second section of the library.

Hyper-programmes are roughly twice as expensive in operating time and one third as expensive in store space as computer-coded programmes. A proposed extension of the hyper-programme compiler will translate each hyper-order of the innermost loop of the hyper-program into computer coded programs to achieve an estimated 10% saving in time by expanding the store space devoted to the loop by a factor of about 3.

Use of symbolic addresses for data will be catered for by establishing a further directory in which the compiler will establish its own coding for symbols. Symbols will be allotted serial locations in the order in which they are input. This will simplify program design and at the same time economise on the use of store for data.

10. PERSPECTIVE

Users of compiler techniques are required to learn the conventional form of orders used on the computer and to be capable of designing the master routine. The compiler routines merely relieve him of considerable detail, thereby minimising the opportunity for program errors and speeding the process of design and fault-finding.

Symbolic translator routines, as exemplified in the second part of this paper, go considerably further towards helping the user by enabling him to design the program in his customary language.

Automatic programming obtains efficiency in preparation of programs at the cost of inefficiency in operation, as compared with human programming. A combination of symbolic translation with compiler techniques will increase efficiency of preparation and decrease inefficiency of operation of computer constructed programs.

PART II THE WREDAC AUTOMATIC PROGRAMMING ROUTINE

by

J.G. Sanderson

1. STATEMENT OF THE PROBLEM

In the first part of this paper it was shown that if automatic programming is to be pushed to its logical conclusion the problem will be presented to the computer in normal mathematical notation, or at least in as close an approximation to it as possible. How close the approximation will be is determined largely by the form of input to the computer. In the case of WREDAC the input is standard five-hole Creed teletype equipment which provides us with the following symbols in addition to capital letters and decimal digits: . * = ' . , % : (- ? / +) £ @. We have here at least the essential mathematical signs, + - . / ( ) but on the other hand we are faced with certain restrictions. Since there are only twenty-six letters available we will have to use subscripts in many problems. There is no provision for raising or lowering characters to represent indices or subscripts so that the latter will be indicated by the use of the asterisk, e.g. *X3 for X_3. There is no sign for square root and division can be represented only by the stroke, "/". Despite these restrictions we may expect a reasonable approximation to mathematical notation. For example, the evaluation of $y = e^{x^2+x}$ for values of x input from tape could be presented to the machine in the following form:

```
@1.  READ (X)                    ("@" indicates that the number 1 is not
     EXP (X2 + X) = Y.            part of an instruction or equation.)
     PRINT (X, Y)
     GO (1)     (i.e. return to step 1)
```

Once we have decided on the form of input shown in this example it becomes clear that the automatic programming routine must consist of two distinct parts. The first part, the **translation routine** takes the **coded statement** of the problem, namely a tape punched to give a series of equations and verbal instructions like those shown, and converts it into a series of orders acceptable to the computer, or at least to one of its interpretive routines. This series of orders we call the **hyper-programme**. The second part, the **computing routine**, accepts the hyper-programme formed by the first part, together with the data for the particular case in question, and carries through the calculation to the final result.

If the automatic programming routine is to fulfil its primary purpose of making the computer available to the ordinary user without the mediation of a trained programmer, it is necessary to place as few restrictions as possible on the coded statement of the user's problem. In particular, we should try to relieve him of the trouble of scaling his variables to bring them within the range of the machine. This means that for a fixed point machine such as WREDAC, the computing routine will take the form of a floating-point interpretive routine. We turn this fact to our advantage by building into this interpretive routine operations which will enable us to simplify the whole system as much as possible.

The normal method of evaluating a conventional algebraic expression is to evaluate the innermost bracket first, to substitute this in the next

innermost and so on until the value of the whole expression is found. Such a method is quite suitable for hand computation, as also for hand-tailored programmes, but if we adopt it here we will find the re-arrangements involved in translation very troublesome. It would be much more convenient if the individual symbols or groups of symbols in the coded statement could be translated directly into suitable hyper-orders without re-arrangement. In the next section we will see how this possibility may be realised.

2. THE INTERPRETIVE OPERATIONS

To simplify explanation we will take as examples, a series of algebraic expressions of increasing complexity, and use them to demonstrate the utility of the operations provided in the computing routine. Although the natural order of computation is to start with brackets, these presuppose the machinery for handling sums and products, so that we must deal with these first.

The Variables.

Starting then with a simple product a.b.c. we postulate a register in which such products are formed. We will call it simply the "product register", P. The variable x appearing in an expression is to be translated into the hyper-order, "multiply (P) by x", where "x" we mean the contents of the twenty-fourth location in a series of twenty-six which are reserved for the variables a ... z. Assuming that (P) = 1 initially the result of the three operations representing the symbols a, b and c will be to form a.b.c. in P.

Quotients and Indices.

This may be extended immediately to cover quotient and also terms containing exponents. In a quotient such as a.b/c.d the orders representing the variables change their effect after the stroke to: "divide (P) by c (or d)." The first two hyper-orders representing ab/cd will produce the product a.b in P, while the last two will divide a.b successively by c and d, giving the desired result.

An exponent, for example the 3 in a^3b, is translated by the order "repeat the preceding operation (namely, to multiply (P) by a) three times". The final effect is then the same as if a^3b were written aaab.

Plus and Minus Signs.

We now consider how the sum of several terms is to be computed - for example ab + cd. The first two symbols a and b are translated into orders causing a.b to be formed in P. Thus the order translating + must cause this product to be added into another register S which was clear initially, and must cause P to be reset with 1 so that the next product may be there. This process may be repeated any number of times, the individual products being formed in P and the terms so computed being summed in S. The order translating the minus sign will reset P with -1 instead of +1, so that the sign of the next term will be changed.

This technique of handling signs fails in the particular case of an expression beginning with a sign, since here the effect is to add the contents of P, unity, into S. This difficulty may be overcome by translating the initial sign by a different order.

Brackets.

Turning now to the method of handling brackets, we may start by remarking that a bracketed expression is treated as a single variable in the larger expression containing it. In a + b(c+d), we must evaluate the expression by re-writing it a + bf where f = c + d. In terms of the system we have built up so far, this means that the value of the bracketed expression must be multiplied into P after it has been evaluated. Hence we cannot use P, nor S either of course, in evaluating the bracket. But on the other hand we wish to retain the same kinds of orders as translations of a ... z and $\pm$, so that we are led to postulate a whole sequence of arithmetic registers S and P. The interpretive routine will respond to brackets by passing from one pair to another. We may describe this system by saying that the various sum and product registers are located at different levels, one pair on each level, and that the interpretive routine transfers its operation up or down one level every time a left or right hand bracket is read respectively.

At this stage it would be well to set down the results gained so far. P_m and S_m are the product and sum registers at level m. c_m is a programmed flip-flop which indicates whether P_m is acting as a multiplier or a divider. The dash ' is used throughout to represent the contents of a register after the operation is completed.

The first column of Table I shows the symbol as it occurs in the coded statement of the problem. The second column gives the serial number of the hyper-order which translates it.

TABLE I

Symbol	Order	Effect
A	0	$P'_m = P_m.A$ if $c_m = 0$
		$P'_m = P_m/A$ if $c_m = 1$
+	1	$S'_m = S_m + P_m$, $P'_m = 1$, $c'_m = 0$
-	2	$S'_m = S_m + P_m$, $P'_m = -1$, $c'_m = 0$
/	3	$c'_m = 1$
(	4	$m' = m + 1$
)	5	$P'_{m-1} = P_{m-1}(S_m + P_m)$, $P'_m = 1$, $S'_m = 0$
		$c'_m = 0$, $m' = m-1$
=A	6	Clear $(S_0 + P_0)$ to the location holding variable A, and reset all $S_i = 0$ and all $P_i = 1$

In explanation of operation 6, which is the translation of =A, it should be pointed out that the value of the expression is $S_0 + P_0$ and not simply S_0 since the last term computed remains in P_0. The same applies to $S_m + P_m$ in operation 5. Notice also that only orders 0 and 6 involve an address to date.

Having seen how expressions built up of the elementary symbols A ...Z + - / ( ) = and the integral exponents can be translated directly without re-arrangement into hyper-programme, we pass on to the extensions which are necessary to make the system reasonably versatile.

Subscripts.

We have already mentioned the subscripts, and the convention that the symbol * will indicate that the letter following has a subscript attached Since transfers to and from the magnetic disc backing store in WREDAC take place in 64 word blocks, the most natural way to handle variables with subscripts is to allocate one disc track to each of the twenty-six letters and allow the subscripts to run from 1 to 64. Thus the symbol *A_k implies two operations in the computing programme: make the necessary transfers to bring the track containing A_1 ... A_{64} into the high-speed store, and then multiply the P-register at the appropriate level by A_k. It is assumed here that k is a number. If it is a letter, the contents of register K have to be used in deciding which value of A to take. This latter process is analogous to the B-line facility.

Numerical Coefficients.

The letter O cannot be used as a variable either with or without subscripts owing to the danger of confusion with the zero. When a numerical coefficient is read in, say the n-th occurring in the coded statement of the problem, it is stored in the disc track allocated to the variables O_1 ... O_{64}, and the hyper-order which would be used to translate *O_n is placed in the hyper-programme by the translation routine. This allows for sixty-four numerical co-efficients. With a little finesse it can be made sixty-four distinct coefficients.

A word should be said here about the way in which the different numbers occurring in the coded statement are distinguished. Numbers marking the groups of equations and verbal instructions are preceded by the sign @. Indices are distinguished from coefficients by the fact that they follow a variable. Subscripts are distinguished by the asterisk preceding the letter to which they apply.

Transcendental functions.

Besides the operations so far postulated, we will require the elementary transcendental functions, preferably specified in their usual form SIN, COS, EXP etc. Also, in the absence of the usual symbol, we will write ROOT for the square root, and MOD for modulus. (The product E.X.P. is distinguished from the operator EXP by separating the letters by full-stops or spaces.)

Taking a particular example such as EXP (A + B) we see that the bracket is evaluated first, and then the operator applied to the result. In terms of the technique described above this means that after the sum $S_m + P_m$ representing the bracket has been obtained we must enter a sub-routine to compute the exponential before multiplying the result into P_{m-1}. This may be achieved in practice by inserting an address in the hyper-order corresponding to the right hand bracket to indicate the entry to the sub-routine in questions, with the stipulation that the address O implies that there is no sub-routine involved, to cover the case where there is no operator.

Verbal instructions.

Lastly, before we pass on to the next section of this paper, we will consider briefly, input, output and control operations. These may very well be specified verbally, since we are already in possession of a sub-routine to translate the verbal expressions of the elementary functions.

To begin with, we require the operations:

READ (A, B, ... K)	Read numbers from paper tape into the locations holding the variables shown.
PRINT (A, B, ... K)	Punch out the contents of the locations shown on paper tape.
INPUT (A, n)	Read numbers from tape into locations holding A_1, A_2, ... until n have been read in, or a special terminating character appears on the tape.
OUTPUT (A,n)	Punch out A_1 to A_n.

Operations involving magnetic tape units may be added if required.

Basic control operations are:

GO (n)	Carry out the operations of the mathematical specification, starting with the group of operations numbered n. (See the example given at the beginning of Part II of the paper).
SIGN (A, l, m, n)	If A is positive go to group l; if zero go to group m; if negative go to group n.
REPEAT (m, n)	Return to the group of equations numbered m, n times, and then pass to the instruction following the "repeat" order.

Here again, further operations may be added if necessary.

In addition certain standard computing operations such as integration and quadrature will be necessary, and will also be specified verbally.

3. THE STRUCTURE OF THE HYPER-ORDERS

A conspicuous feature of the system developed in the preceding pages is the great variation in the amount of information borne by the different hyper-orders. Thus the order for "+" involves function digits only, whereas, the order for SIGN involves, in addition to the function digits, four addresses. On the principle that a Procrustean treatment can never be really efficient, we are led to adopt an order code in which the length of the hyper-orders is allowed to vary. They are in fact, still made up of fixed length units, 17 bit half-words which I will call "syllables", but the number of syllables in an order may vary from one totwelve.

There are four types of syllable, but they all have the same structure. Numbering the digits 0 to 16, starting with the most significant, we group them as follows:

$d_0 = 1$	for the last syllable of an order: otherwise 0.
$d_1 \ldots d_5 = T$	the relative track number.
$d_6 \ldots d_{11} = A$	the relative high-speed store address.
$d_{12}, d_{13} = C$	the code number, specifying the type of syllable.
$d_{14} \ldots d_{16}$	are not used.

Corresponding to the four values of C there are four types of syllable. Those having C = 1, 2 and 3 represent variables without subscript, variables with numerical subscripts and variables with literal subscripts respectively. A hyper-order containing one of these as its only syllable represents a variable as it occurs in an algebraic expression. When they represent variables in operations like SIGN or PRINT these syllables always occur in conjunction with a syllable having C = 0, which serves to define the operation.

The hyper-orders translating + - / and (are all composed of a single syllable having C = 0. The relative address, A, of this syllable gives the entry to the sub-routine effecting the operation in question. The remaining hyper-orders (excluding the variables, which we have already discussed) are all made up of two or more syllables. In each case the first syllable gives the address of the sub-routine to be used, together with the track containing it, if it is not stored permanently in the high-speed store during the computing phase of the calculation.

4. THE TRANSLATION ROUTINE

The translation routine is of necessity rather complex, and it will serve no useful purpose to give a detailed description of it here. There are, however, two points which are sufficiently general in their application to justify mention.

Firstly, in the coded statement of the problem there are certain groups of teletape characters which only have significance when taken together - decimal numbers or words for example. It is worth while constructing a closed sub-routine which will read in one such group and form one or more numbers giving an adequate description of it for the purposes of translation. For example, if the next group were a word, say PRINT, it would produce the track and address numbers of the syllable translating PRINT, together with a third number distinguishing it from the other groups, such as signs or variables, which might have occurred.

Secondly, we must take account of the fact that the meaning of a group of teletape characters depends on the context in which it occurs. This is most marked in the case of numbers which, we saw, have four possible interpretations.

After reading in a group, control is thrown to a sub-routine which will make the appropriate addition to the hyper-programme that is being formed. To this end we provide a directory giving the entries to the various sub-routines corresponding to the different kinds of groups. If every group had a unique interpretation one such directory would suffice. But since the interpretation depends on the context we must in fact have several.

Before programming the translation routine, we enumerate all the contexts which affect the interpretation of groups, and reserve a certain location in the store in which to keep track of the current context while the coded statement is being read in. For example, in reading PRINT (A, B, C), the context register would be set with a number representing "verbal instruction" as soon as "PRINT" had been read, and the corresponding directory would come into use. This would cause "(" to be ignored, since it has significance only in algebraic expressions. The syllables for A, B and C would be placed after the original syllable for print, instead of forming three distinct hyper-orders, while ")" would cause the end of order marker to be placed in the syllable for C.

5. CONCLUSION

Since the description of the WREDAC automatic programming routine given in this paper has, if anything, erred on the side of too much detail, it would be well to conclude with a summary of the system as a whole, in order to place it in its proper perspective.

Fundamentally, it is a floating point interpretive routine, provided with input, output and control facilities, and also with the more important functions of one variable. However, it does differ from the ordinary run of interpretive routines, since it has an auxiliary translation programme which enables the user to present his problem in a symbolic form which resembles ordinary mathematical notation.

The form taken by the automatic programming system is determined by two factors; firstly, we consider it essential that it should work in floating mode, and since WREDAC is a fixed point machine this necessitates the use of an interpretive system. And secondly, since we have an adequate backing store with relatively fast access, it is quite practicable to store the entire programme - the translation routine, interpretive routine, and all auxiliary sub-routines - on the disc throughout the calculation. Thus there is no point in using compiler techniques, since there is no question of economising store space.

The completed programme will contain about 1500 orders. If it is used often enough it will be stored permanently on the disc, but even if this is not done, it will take only three-quarters of a minute to read from paper tape, and will then be available for performing as many calculations as desired without reading in again. Once it is in the machine a single manual operation causes the translation routine to read the coded statement of the problem, convert it to hyper-code, and store it on the disc. After placing the data tape in the reader, one more manual operation throws control to the computing routine, and from there on operation is automatic.

We will conclude by remarking that the programme is still experimental in nature. We anticipate many changes before it settles down to a steady state.

DISCUSSION OF PART I

Mr. R.G. Smart, University of Technology, N.S.W.

Mr. Hill mentioned that we may open up loops and perform them as a straight calculation. What do we save by not going round the loop? The only saving I can see is the counting procedure.

Mr. G.W. Hill (In Reply)

This is right. In a polynomial routine the number of orders used in controlling is equal to the number doing the calculation.

Mr. B. Swire, University of Sydney.

In what part of the store is the library? If you alter it, what happens when you want to use it again?

Mr. G.W. Hill (In Reply)

We have the high speed store and four drum stores. On drum four we have the initial orders, compiling routine and library, all of which can be brought down into the high speed store by console switching. We alter them in the high speed store, leave them unchanged on the drum.

DISCUSSION OF PART II

Mr. B.A. Chartres, University of Sydney.

Two questions. How does Mr. Sanderson's programme distinguish between the word READ and the product R.E.A.D? Secondly, since he uses a relatively small number of words would it not be almost as easy for the user and much easier for the programme to use symbols - ϕ for READ for example?

Mr. J.G. Sanderson (In Reply)

We distinguish between words and products by putting some symbol - a full stop say - between the letters of a product. For the second question, we could very well do what you are suggesting, but it is quite simple to translate words, and very much easier for the user.

Mr. R.H. Merson, Royal Aircraft Establishment.

On Whirlwind they have an interpretive system which is working now and on which they are solving differential equations. They do not have to distinguish between products and instructions, since instructions do not occur in equations.

Mr. G.W. Hill, C.S.I.R.O.

Do you know how they distinguish between b = a + exp y and b = a + e.x.p.y ?

Mr. R.H. Merson, Royal Aircraft Establishment.

They use F_1 F_2 F_3 etc., for their functions. They do not use F for anything else.

Mr. J.G. Sanderson (In Reply)

I prefer functions written in their normal fashion for the convenience of the user.

Mr. R.H. Merson, Royal Aircraft Establishment.

In more complicated problems they use about thirty of these operations. It would be difficult to write them all in full.

Mr. G.W. Hill, C.S.I.R.O.

On C.S.I.R.A.C. we had envisaged the possibility that if the machine failed to find EXPY in the directory, it might look for EXP and take Y as the variable. Another point. We considered packing the variables without any gaps by allocating them successive store locations as they occurred in the programme.

Mr. B. Swire, University of Sydney.

I am interested to note that Mr. Sanderson's translation routine translates into an interpretive rather than a machine code. One commonly pampers programmers at the cost of speed by writing interpretive programme. Why should you also pamper the translation routine?

Mr. J.G. Sanderson (In Reply)

We intend to use a completely floating system. Having gone to all the trouble to present the user with this system we cannot then restrict him to numbers less than two. Since W.R.E.D.A.C. is at present a fixed point machine the programme has to be interpretive.

Dr. S. Gill, Ferranti Ltd.

I am disappointed that no one has mentioned symbolic addresses, such as we are using on the Mercury. This allows us to make alterations within a subroutine without re-numbering the whole subroutine. It involves attaching symbolic labels to the places where we re-enter the subroutine or modify it in the machine, and allowing the machine to substitute the addresses for these labels. In the system used on Mercury a label can only be referred to in its own subroutine, so that the same labels can be used in different subroutines without clashing.

Dr. A.S. Douglas, University of Leeds.

I feel these things have got a little out of proportion. I am sceptical as to how far they will increase this power of the machine. In organising a calculation you have several stages -

Mathematical analysis
Preparation of Data
(Input
(Code checking
Computation
Output

These routines only touch the surface of a few of these problems. Mathematical analysis is not helped. Preparation of data is not facilitated as far as I can see. Input and code checking are made more difficult, and computation slowed up. I find it difficult to see a problem where it would make a major contribution, apart perhaps from training. You still have to do the mathematics and break the problem down into its elementary loops.

Dr. S.H. Hollingdale, Royal Aircraft Establishment.

I want to make the point that Prof. Douglas established his case by leaving out one of the items, writing out the programme.

Dr. A.S. Douglas, University of Leeds.

This is included in input and code checking. In practice writing the programme is absorbed in mathematical analysis. Programming in machine code or auto-code take equally long, and the basic trouble is in the analysis.

Dr. S.H. Hollingdale, Royal Aircraft Establishment.

The real advantage of these routines is that they "can" an awful lot of work, so that it is done once and is ready to use.