GEORGE

IA and II

A semi-translation programming scheme for DEUCE

PROGRAMMING AND OPERATION MANUAL

GEORGE, or the General Order Generator, is a program for DEUCE permitting mathematical problems to be presented to the machine in a simple "addressless" instruction language, here called "G-code". To use this code the programmer must learn a special method of writing mathematical formulae, known as "reverse Polish" notation. Once this is mastered, however, programming is considerably easier and quicker than by other methods.

Automatic partial translation of G-code into DEUCE instructions secures a higher speed of operation than is usual with interpretive schemes. Calculations are in floating-point arithmetic to an accuracy of about six decimal digits.

Prepared by:

C. L. Hamblin

University of New South Wales

GEORGE IA and II

General Description

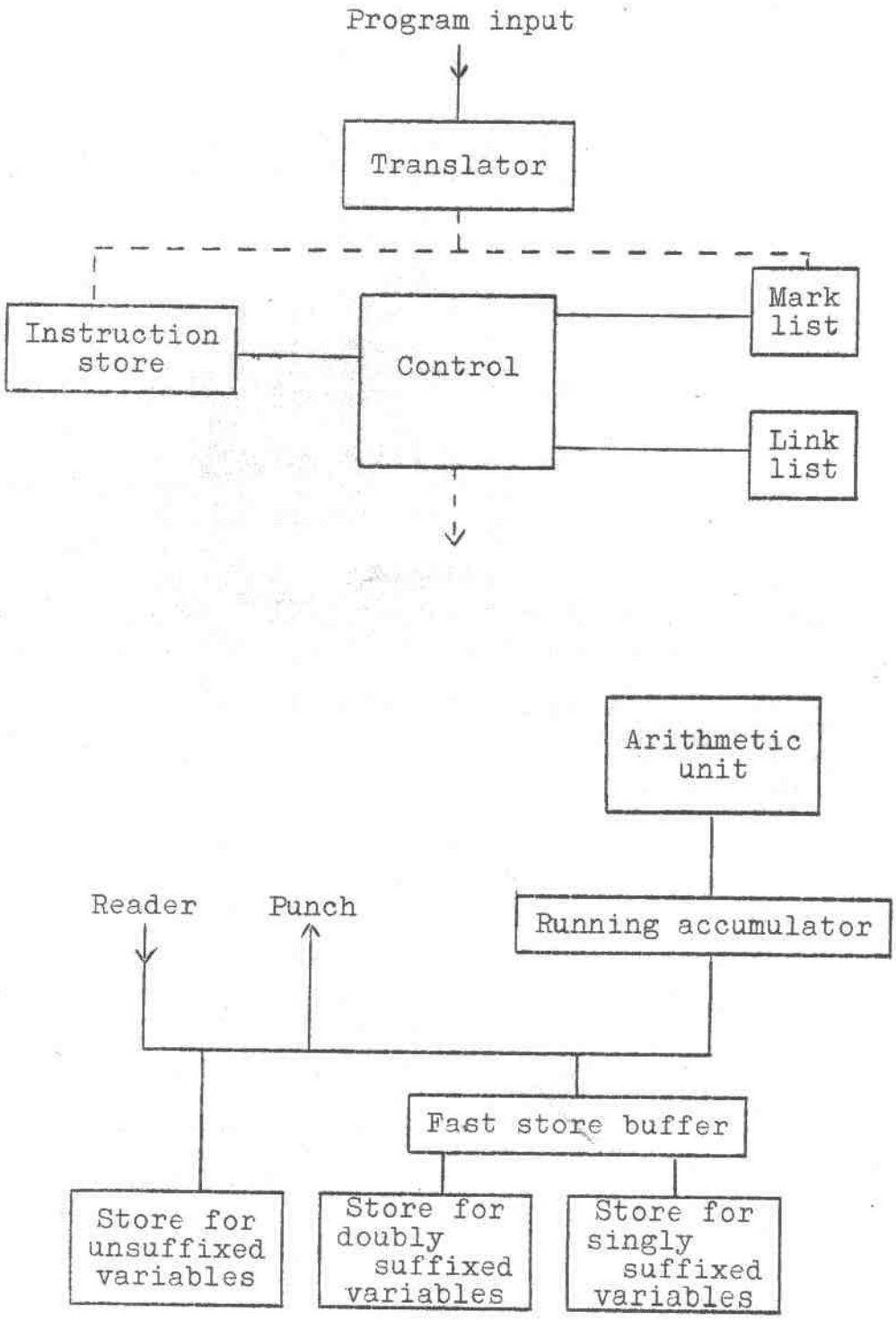
GEORGE IA and GEORGE II are alternative versions of a scheme to simplify the programming of DEUCE for routine mathematical problems. In either case, a basic program is read into the machine and stored in the magnetic drum. So long as one of these programs is in the machine, programs for the solution of particular problems may be read in in "G-code", which is a highly simplified and condensed instruction language. A program in G-code in fact resembles a mathematical formula for the result required more than it does an orthodox machine program. In particular, the programmer never has to specify any "addresses" for numbers or instructions inside the machine.

As written, a program in G-code is a sequence of mathematical symbols such as numerals, variables and arithmetical and other special signs; and the symbols are transcribed to cards in a numerical code. Broadly, each symbol may be regarded as an individual "instruction" of the program.

It is not in general necessary for the programmer to be familiar with details of the operation procedure, unless he wishes to follow a program through step-by-step for checking purposes. Given reasonable care in program preparation, errors should be rare.

GEORGE IA and GEORGE II accept the same programs and differ only in details of internal operation. GEORGE IA is a "semi-translation" scheme, and the program goes through two phases: (i) translation into "keywords", which are DEUCE instructions leading to the stored routines required by the program, and (ii) calculation, in which the keywords are taken one at a time and the appropriate routines performed. The first phase however takes at most a few seconds, and transition to the second phase is automatic. GEORGE II is a "pure interpretive" scheme and is slower in operation but has considerably more facilities for manual operation: it will be used principally for the testing of programs for subsequent use on GEORGE IA.

The following description is divided into two sections, viz. (1) Programming and (2) Operation. The first section applies indifferently to both versions of the scheme and assumes no detailed knowledge of the operation of DEUCE.



Block diagram of the GEORGE pseudo-machine

1. PROGRAMMING

1.1 "Reverse Polish" notation

The majority of the symbols used in programming are those normally used in mathematics, but the order in which they occur is different from the usual order: formulae are written in "reverse Polish" notation. This notation has a number of advantages which fit it for use with machine computatior

In this preliminary description let us take for granted that we can use the letters "a", "b", "c", ... as ordinary algebraic variables. A mathematical formula is a prescription for operating on the numbers such variables represent. We can classify the operators involved as:-

(i) monadic operators, or operators on a single number, such as the minus-sign in "-a", the modulus sign in "$|a|$", and various functional operators such as "log", "exp", "sin" etc.; and

(ii) diadic operators, or operators on a pair of numbers, such as the operators for addition, subtraction, multiplication, division and various others.

In ordinary mathematical notation, a monadic operator is most frequently written in front of the variable concerned, as in "-a", and a diadic operator between two variables, as in "a + b". There are, however, common exceptions: multiplication is indicated by simple juxtaposition as in "ab"; the sign "-" is used indifferently for subtraction and as a minus-sign (subtraction from zero); and brackets are needed to distinguish, say, "-a + b" and "-(a + b)".

In reverse Polish notation the operator-signs, whether monadic or diadic, are written after the variables concerned, as follows:-

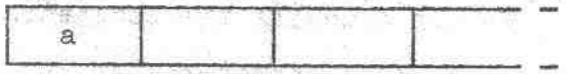
For	$a + b$	write	a b +
"	$a - b$	"	a b -
"	ab	"	a b ×
"	$\frac{a}{b}$	"	a b ÷
"	$-a$	"	a neg
"	$\|a\|$	"	a mod

--- and so on. One result of this is that brackets are never needed: any of the above expressions may be used directly and without ambiguity as an element in a longer expression. For example:

For	$-(a + b)$	write	a b + neg
"	$\|a + b\|$	"	a b + mod
"	$\|a\| + \|b\|$	"	a mod b mod +
"	$a + b + c$	"	a b + c + (or a b c + +, etc.)
"	$(a + b) \times c$	"	a b + c ×
"	$a + bc$	"	a b c × +
"	$ab + \frac{a}{b}$	"	a b × a b ÷ +

In understanding the use of this notation in GEORGE it will be of assistance to have in mind a picture of the internal logic of operation. The pseudo-machine into which DEUCE is converted by the basic GEORGE program can be envisaged as equipped with a "running accumulator": this is a set of storage locations (known as "cells") arranged in a linear order and operated on the "last-in-first-out" principle. When a variable occurs in a program its "value" (from an appropriate position in the main number-store) is placed in the first vacant cell of the accumulator, i.e. in cell 1 if this does not already contain a number, otherwise in cell 2 or etc.: and when an operator-symbol occurs, the specified operation is carried out on the contents of the last occupied cell or last two occupied cells, depending as the operator is monadic or diadic. The operation of the formula "a b +" may for example be pictured as follows:

(i) "a": number a transferred to cell 1.

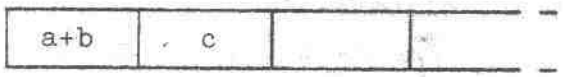


(ii) "b": number b transferred to cell 2.

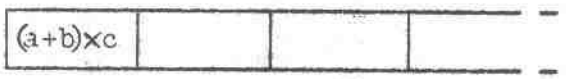
(iii) "+": contents of last two occupied cells added together; cells cleared and result replaced.

The result is exactly as if a single number, the sum of a and b, had been specified. In consequence, this number is available for further operations if required. For example, to calculate "(a + b) × c" we can write "a b + c ×". The first three steps are as above, after which:

(iv) "c": number c transferred to cell 2.



(v) "×": contents of last two cells multiplied; cells cleared and result replaced.



Alternatively this calculation could have been carried out in the form "c a b + ×". In this case three cells would first have been filled with numbers before any calculations were carried out: the result of the addition would go into cell 2, and the final result as before into cell 1.

Quite generally, the overall effect of any calculation of a number is to place the number concerned in the first vacant cell of the accumulator. (It may then be punched out or transferred back to the main number-store).

There are two special operators which, although not strictly necessary, are frequently useful. These are:

(i) "dup": this "duplicates" the contents of the last occupied cell in the next cell. For example:

For $(a + b)^2$ write a b + dup ×

(ii) "rev": this "reverses" (i.e. interchanges) the

contents of the last two occupied cells. For example:

For $a + b + \frac{c}{a + b}$ write a b + dup c rev ÷ +

Certain special functions such as "log", "exp", "sin" etc. are available and are treated as monadic operations as indicated in the table below. In addition it is frequently possible to write subroutines in G-code in such a way that the symbol which calls in the subroutine may be treated as an operator-symbol.

TABLE OF OPERATIONS PROVIDED IN BASIC PROGRAM

Symbol	Monadic or diadic	Operation	Example For:-	Example -write:-	
+	diadic	addition	a + b	a b +	
-	diadic	subtraction	a - b	a b -	
×	diadic	multiplication	ab	a b ×	
÷	diadic	division	$\frac{a}{b}$	a b ÷	
neg	monadic	minus sign	-a	a neg	
mod	monadic	modulus	|a|	a mod	
max	diadic	maximum	max(a, b)	a b max	
log	monadic	natural logarithm	$\log_e a$	a log	
exp	monadic	exponential	e^a	a exp	
pow	diadic	power	a^b	a b pow	
rem	diadic	remainder	rem. of a on div. by b	a b rem	(see note)
√	monadic	square root	$\sqrt{a}$	a √	
sin	monadic	sine } angle in radians	sin a	a sin	
cos	monadic	cosine } angle in radians	cos a	a cos	
dup	duplicates last occupied cell in next cell				
rev	interchanges last two occupied cells				

Note: the "remainder of a on division by b" is defined as a number x such that $nb + x = a$, where n and b are both integers and $0 \leq x < b$. Note that "a 1 rem" gives the positive fractional part of a.

1.2 Numerals

Data required in a calculation will normally be read in separately from the program by means of special instructions. It is frequently convenient, however, to include numbers as part of the program itself. For this purpose the symbols 0--9 are provided, together with a symbol for decimal point. Sequences of these symbols (of not more than nine digits in length) are treated as numbers in the usual way. They are converted by the translator to floating-point binary form (22 binary digit signed mantissa, 10 binary digit signed

exponent; hence about six decimal digit accuracy).

In operation the number concerned is placed in the first vacant cell of the accumulator. Hence constant numbers may be included in a formula in the same way as variables.

For	$3a + 4b$	write	a 3 × b 4 × +
"	$.14(a + 33.8)$	"	a 33.8 + .14 ×

When two numbers must be written consecutively they should be separated by a comma (special symbol, omitted in translation):

For	$\frac{4.3}{16 - x}$	write	4.3, 16 x - ÷

1.3 Variables

Variables represent storage locations. An "alphabet" of 32 letters is provided, comprising all letters of the Roman alphabet except "o", and certain Greek letters (see coding table in Appendix I). There are three entirely separate ranges of storage locations as follows:-

(i) The "unsuffixed variable" store: this is a range of 32 locations, one for each letter of the alphabet. It is this store which is referred to when letters are used alone in the usual way.

(ii) The "single suffix" or "vector" store: this has 32 × 32 = 1024 locations, and is referred to by using variables with single suffixes. Any of the 32 letters may be used with a suffix, and suffixes may range in value from 0 to 31. (See section 1.9 below on the use of larger values).

(iii) The "double suffix" or "matrix" store: this has 32 × 32 × 4 = 4096 locations and is referred to by using the letters a, b, c and d with double suffixes. Either suffix may range in value from 0 to 31. (See section 1.9 below on the use of larger values).

The entire number-store is "random access". Hence the programming of a mathematical problem usually requires little or no change in nomenclature.

A special notation is however used for suffixes: these are written first, and are separated from the variable-symbol by a bar sign "|" (single suffixes) or double-bar sign "‖" (double suffixes).

For	a_1	write	1 | a
"	x_{14}	"	14 | x
"	$a_{1,1}$	"	1, 1 ‖ a
"	$c_{6,31}$	"	6, 31 ‖ c

The advantage of this notation is that suffix-values are first treated in the ordinary way as numbers, and are placed in the accumulator. This permits the use of algebraic suffixes and of arithmetic operations in suffixes; the bar sign simply takes the number in the last occupied cell and treats it as a suffix of the letter which follows. (Similarly the double-bar sign takes the numbers in the last two occupied cells).

For	a_n	write	n | a
"	a_{j+n}	"	j n + | a

For	a_{ij}	write	i j ‖ a
"	$a_{4,x+1}$	"	4 x 1 + ‖ a
"	a_{j_k}	"	k | j | a

Suffixed variables may be used in formulae in exactly the same way as simple variables:

For	$a_i + b_i$	write	i | a i | b +
"	$a_{ij}x_j$	"	i j ‖ a j | x ×

1.4 Names

When a number has been calculated in the accumulator it may be put away in the number store by using a "name" symbol. This is equivalent to "naming" (or renaming) the number with a certain letter or suffixed letter. A name-symbol is written as the corresponding variable in brackets, e.g. "(a)" (treated as a single symbol). Names may be suffixed in the same way as variables, e.g. "1 \| (a)", "i j ‖ (a)" etc. In all cases such a combination of symbols operates to transfer the number in the last occupied cell to the appropriate location in the number store.

A name does not, however, "cancel" the last occupied cell of the accumulator. For this purpose a special symbol ";" is used if required.

For	"Put c = a + b"	write	a b + (c) ;
"	"Put n = 6"	"	6 (n) ;
"	"Add 1 to x"	"	x 1 + (x) ;
"	"Put $b_{ij} = a_{ij}^2$"	"	i j ‖ a dup × i j ‖ (b) ;

1.5 Input and output

The symbol "R" represents an instruction to read one number from a card and place it in the first vacant cell of the accumulator.

The symbol "(P)" represents an instruction to punch out the number in the last occupied cell of the accumulator. It is written in brackets to indicate that it operates like a name-symbol; i.e. that it does not "cancel" the cell.

For	"Read in the number x"	write	R (x) ;
"	"Punch out the number x"	"	x (P) ;
"	"Punch out the sum of a and b"	write	a b + (P) ;
"	"Read two numbers and punch out their sum"	write	R R + (P) ;

Provision is also made for input or output of blocks of numbers directly to or from the vector and matrix stores, as in the following examples:-

To read in a sequence of numbers $x_a, x_{a+1}, \ldots x_b$

write a b R_1 (x)

To punch out the same sequence of numbers write

a b P_1 (x)

To read in a matrix of numbers a_{ij} -- a_{mn} write

i m j n $R_{\|}$ (a)

To punch out this matrix write

i m j n $P_{\|}$ (a)

A matrix is read or punched row by row, e.g. in the order

a_{ij}, $a_{i,j+1}$, ... a_{in}, $a_{i+1,j}$, $a_{i+1,j+1}$, ... $a_{i+1,n}$, ... etc.

Numbers are punched one per card, in floating decimal. For details see section 2.2 below.

1.6 Repetitive operations

It is very frequently required that a calculation be performed repetitively on each of a sequence of numbers: e.g. on each element of a vector. For this purpose a special facility is provided in G-code; thus:

To perform an operation for each value of x, in unit steps, from a to b, write

a b rep (x) .....]

The program for the operation to be performed is inserted as shown by the row of dots, "x" being used in it as a variable. For example:

To calculate the squares of the natural numbers from 1 to 20, punching them out as calculated, write

1, 20 rep (j) j j × (P) ;]

To put $a_j = b_j^2$ for each value of j from 1 to n, write

1 n rep (j) j | b dup × j | (a) ;]

A frequent use of this facility will be for repeated summation. For this purpose an accumulator cell can first be marked out with zero, and then terms repeatedly added to it. For example:

To calculate the sum of the squares of the natural numbers from 1 to 100 $\left(\text{i.e. } \sum_{j=1}^{100} j^2\right)$ write

0, 1, 100 rep (j) j j × +]

Repeated products can be handled similarly, first marking out a cell with the number 1:

To put $a_n = n!$ for all n from 1 to 15, write

1, 1, 15 rep (x) x × x | (a)] ;

Simple repetition a fixed number of times can be effected using a "dummy" variable, i.e. one not required in the calculation. For example:

To iterate five times from initial value unity using the formula $x_{(n+1)} = \frac{1}{2}(x_{(n)} + \frac{y}{x_{(n)}})$ write

1, 1, 5 rep (a) dup y rev ÷ + 2 ÷]

(This is Newton's method for the square root of y).

Repetitive operations may be "nested" one within another: i.e. the operation to be repeated may itself be repetitive. For example:

To calculate the sum of squares of the natural numbers from 1 to n, for each value of n from 1 to 20, storing the results as a_1 -- a_{20}, write

1, 20 rep (n) 0, 1 n rep (x) x x × +] n | (a) ;]

(Here the "inner" repeat performs the operation "x x × +" and the "outer" the operation "0, 1 n rep (x) x x × +] n | (a)

To put $b_i = \sum_{j=1}^{n} a_{ij}x_j$, for all i from 1 to n, write

1 n rep (i) 0, 1 n rep (j) i j ‖ a j | x × +] i | (b) ;]

To calculate $\sum_{i,j=1}^{n} a_{ij}x_ix_j$ write

0, 1 n rep (i) 1 n rep (j) i j ‖ a i | x j | x × × +]]

(or for somewhat higher speed)

0, 1 n rep (i) 0, 1 n rep (j) i j ‖ a j | x × +] i | x × +]

PROGRAM EXAMPLES

Many complete programs will need no other facilities than those already introduced. It will be noticed that these include the majority of problems handled by "tabular" or "matrix" interpretive schemes. Although GEORGE is not intended as a competitor of these, it may in the case of problems which are not too large compare in speed with them; since a sequence of elementary operations may be performed in a single "repetition", without intermediate access to the number-store.

Example 1. A set of 20 numbers is to be read in, and their mean and root mean square computed and punched out.

A complete program is as follows:-

```
1, 20 R1 (a)                                   (read in data)
0, 1, 20 rep (j) j | a + ] 20 ÷ (P) ;
                                   (calculate and punch mean)
0, 1, 20 rep (j) j | a dup × + ] 20 ÷ .5 pow (P) ;
                          (calculate and punch root mean sq.)
```

Alternatively the mean and root mean square could be calculated in a single repetition as follows:

```
1, 20 R1 (a)
0, 0, 1, 20 rep (j) j | a + rev j | a dup × + rev ]
20 ÷ (P) ; 20 ÷ .5 pow (P) ;
```

Example 2. To repeat the above program n times for different sets of data, where n is given on a first card, prefix with "R (n) ; 1 n rep (x)" or "1 R rep (x)" and add "]" at end.

Example 3. To calculate the definite integral

$$\int_1^5 \frac{e^{-\frac{1}{2}x^2}}{1 + e^{-ax}}\,dx$$

by Simpson's rule with 16 intervals of argument, for each of five values of a, as given on cards.

Let $s_0, s_1, \cdots s_{16}$ represent a "Simpson vector",

i.e. with the values 1/48, 4/43, 2/48, 4/43, ... 1/48 : this can be read in from cards with "0, 16 R_1(s)" or it can be prepared by a preliminary section of program, e.g.:

```
[ 1, 48 ÷ 0 | (s) 16 | (s) 2 × 1, 7 rep (j) j 2 × | (s) ]
  2 × 0, 7 rep (j) j 2 × 1 + | (s) ] ;
```

Now the main part of the program can be set out as follows:

```
1, 5 R1 (a)                 (read in values a1, ... a5)
1, 5 rep (i) i | a (a) ;
                            (repeat as follows with a = a1, ... a5 in turn)
0, 0, 16 rep (j)
                            (mark acc. space with zero and repeat as follows
                                for j from 0 to 16)
j 4 ÷ 1 + (x)               (calculate argument)
dup × 2 ÷ neg exp x a × neg exp 1 + ÷
                            (calculate integrand)
j | s × + ]
                            (multiply by Simpson coefficient and add;
                                end inner repeat)
i | (r) ; ]                 (store result; end outer repeat)
1, 5 P1 (r)                 (punch results)
```

1.7 Subroutines

Subroutines written in G-code may be simply attached to the end of a program. Each subroutine is introduced by the symbol "*" followed by a number, and terminated by the bracket-symbol "]". A subroutine may be called in in the main program by giving its number, followed by the symbol "↓". When the subroutine is complete, the main program will automatically be resumed from the point of interruption.

For example, a subroutine (number 6) to calculate the square root of the sum of squares of a pair of numbers given in the last two occupied cells of the accumulator can be written as follows:

```
* 6 dup × rev dup × + .5 pow ]
```

If this subroutine is attached to a program, it may be called in by means of the symbols "6 ↓". Thus for the square root of the sum of squares of a and b write

```
a b 6 ↓
```

(Note that "6 ↓" can be considered here simply as a diadic operator).

The number of a subroutine, as written at the beginning of the subroutine itself, must be represented by a single numeral symbol. (Besides the symbols 0 -- 9 there are however also symbols $\overline{10}$, $\overline{11}$, ... $\overline{31}$: see coding table in Appendix I). The main program may specify the subroutine number in any manner.

Sometimes a subroutine will use locations in the number store. In this case some care must be taken that the locations concerned are not also in use in the main program when the subroutine is called. It is generally convenient to reserve Greek letters for use in subroutines.

Subroutines may themselves involve reference

[1] And Addendum.

to other subroutines. For example, a subroutine (say number $\overline{14}$) which when given a pair of numbers in the accumulator will divide them respectively by the square root of the sum of their squares can be written using subroutine no. 6 in the example above:

```
* 14 (β) rev (α) 6 ↓ (γ) α rev ÷ β γ ÷ ]
```

When subroutines are attached to a program, the program itself must be terminated by a bracket-symbol "]" so that it will not carry on into the subroutines. The bracket-symbol in fact has three uses: (i) to return a subroutine to the main program; (ii) to restart or terminate a repetitive operation; and (iii) to terminate a program. It fulfils the third function provided no subroutine or repetitive operation is in progress:

To read in two numbers and punch the square root of the sum of their squares, using subroutine no. 6 as above: the complete program is:

```
R R 6 ↓ (P) ;                    (main program)
]                    (bracket to indicate end of main
                                                program)
* 6 dup x rev dup x + .5 pow ]        (subroutine)
```

1.8 Skips and discrimination

A place marker, consisting of the symbol "*" followed by a single numeral symbol as described in the previous section, may be inserted anywhere in a program: it will be omitted by the translator, which however notes in a special list called the "mark list" the address of the next instruction following. A "skip" instruction, or instruction to resume the program from a point so marked, may be given by writing the number of the mark followed by the symbol "↑". This symbol operates in the same way as the "skip to subroutine" symbol "↓", except that it makes no provision for later returning to the point of interruption.

For example, to continue indefinitely reading in pairs of numbers and punching out their sum, write

```
* 0 R R + (P) ; 0 ↑
```

Skips may however be <u>conditional</u> on the results of a calculation. For this purpose two "relational" operators "=" and ">", and three "logical" operators "~", "&" and "v" are provided. A skip ("↑" or "↓") is treated as conditional whenever one of the two relational operators has preceded it (since the previous skip).

The relational operators are diadic operators which give as result a digit (the sign digit) to indicate whether the relation is "true" or "false". For example:

"a b =" gives "truth" if a = b, "falsehood" otherwise.

"a b >" " " a > b, " " .

When a skip is "conditional", the last occupied cell of the accumulator is examined to see if it indicates "truth"; and the skip takes place only if this is so. Thus:

For "If x = 0 skip to mark no. 8 (otherwise continue)" write

```
x 0 = 8 ↑
```

Both the "truth" indication and the mark number are cancelled from the accumulator, whether the skip takes place or not.

The "logical" operators are operators on truth-values, and extend the range of possible relations.

"~" (or "not") is monadic and gives logical negation: that is, it changes "truth" to "falsehood" or vice versa. Thus:

For	"a ≠ b"	write	a b = ~
"	"a ≤ b"	"	a b > ~

"&" (or "and") is diadic and gives logical conjunction; i.e. "truth" if both operands have "truth", "falsehood" otherwise. Thus:

For	"a = b = 0"	write	a 0 = b 0 = &
"	"a < x < b"	"	x a > b x > &

"∨" (or "or") is diadic and gives logical disjunction; i.e. "truth" if either or both operands have "truth", "falsehood" only if both have "falsehood". Thus:

For	"x = 0 or 1"	write	x 0 = x 1 = ∨
"	"a or b or both exceed n"	write	a n > b n > ∨

Note that as constants "0" may be written to represent "falsehood" and "-1" (i.e. "1 neg") for "truth".

For "If any a_j is zero (j = 1, ... n) obey subroutine no. 3" write

0, 1 n rep (j) j | a 0 = ∨] 3 ↓

To iterate from initial value unity using the formula

$$x_{(n+1)} = \tfrac{1}{2}(x_{(n)} + \frac{y}{x_{(n)}})$$ until successive values differ by not more than e, write

1 * 0 (x) y x ÷ + 2 ÷ dup x - mod e > 0 ↑

(The result is left in the accumulator).

1.9 Miscellaneous programming points

The following notes deal with a number of special points not so far mentioned.

(i) The accumulator is limited to 12 cells. This is more than enough for all normal calculations; but a program which demands more cells may be written inadvertently, e.g. by omitting a "cancel" sign (";") in a repetitive operation. Such a program will give failure; as also will a program which instructs taking a number from an empty accumulator.

(ii) Subroutines and repetitive operations involve the use of a special set of locations called the "link list". An entry is inserted in this list whenever a subroutine or repetitive operation is entered, and cancelled (by the bracket-symbol "]") when it is completed. The link list is limited to six entries, and a program must not at any one time have more than this number of subroutines and/or repetitive operations nested one within another.

(iii) There is room in the instruction store for 512 symbols before translation, and for a similar number of keywords after translation. Translation of symbols into keywords is not exactly one-for-one: certain symbols such as

comma and marks are omitted, but a few symbols give rise to two keywords each (see Appendix II). On the average there will generally be about 10% more keywords than corresponding symbols. To avoid detailed counting of symbols and keywords programs can in cases of doubt be sectionalised (see operating instructions).

(iv) In certain cases symbols are translated in pairs or groups, and the members of a pair or group must not be separated by other symbols such as comma, mark, skip etc. These are: numeral sequences; bar-signs "|" and "‖" with their following variables or names; "rep" with its following name; "$R_|$", "$P_|$", "$R_‖$", "$P_‖$" and their following names; the asterisk "*" and its following numeral.

(v) Numbers used in a program for "indexing" purposes (i.e. as suffixes, skip numbers, ranges of repetition) are assumed to be integers of modulus less than 2^{15}, and use of numbers with fractional parts or outside this range may give incorrect results. Skip numbers are interpreted modulo 32.

(vi) Although suffixes are normally limited to the range 0 to 31, use of values outside these limits is possible under certain circumstances. When a <u>single</u> suffix exceeds 31, the storage location represented is one corresponding with a letter alphabetically following: e.g. "a_{32}" represents the same number as "b_0", "a_{33}" the same as "b_1", "a_{64}" the same as "c_0" and so on. Hence if the larger suffix-values are required the appropriate storage-locations must be reserved. (Consult the alphabetical order of the letters in the coding table in Appendix I if necessary). In the extreme case the entire vector store may be reserved for (say) $a_0, \cdots a_{1023}$, no other singly-suffixed variables being used. Larger values again will merely cause reduplication. Negative suffixes are similarly interpreted modulo 1024, causing spill to locations of alphabetically preceding letters: e.g. "a_{-1}" represents the same number as "v_{31}".

In the case of doubly-suffixed variables and names the <u>second</u> suffix is interpreted modulo 32 and does not cause spill; but the <u>first</u> suffix is interpreted modulo 128 and causes spill through a, b, c and d in the same way as described.

(vii) The symbol "wait" performs no operation, but holds up the calculation until a manual signal (single shot) is given, incidentally displaying on the output lights the number in the last occupied cell of the accumulator (or the special indication P_{1-10} if the accumulator is empty). Numbers are in standard "semi-floating" binary, with signed exponent in positions 1-10 and signed mantissa in positions 11-32 (binary point between positions 31 and 32).

(viii) Symbol "I" operates to transfer a number set in on the input keys to the first vacant cell of the accumulator. This permits manual control of a program by means of discrimination: e.g. for "If the input lights are clear, skip to mark no. 4" write "I 0 = 4 ↑". It is generally advisable that the program should have "wait" at some preceding point to permit the input lights to be set as required. (See however the operating instructions for GEORGE II).

2. OPERATION

2.1 General procedure

The basic GEORGE IA program or GEORGE II program is read into the machine and is stored on the magnetic drum. Provided this remains intact, a program in G-code (or "G-program") can be inserted at any time using the "initial input" key, and will be translated and performed without interruption. A program in G-code consists of:-

(a) A special "G-program initial card".

(b) A card or cards containing in order the symbols of the program in numerical code, with a special punching (two adjacent digits in Y-row) to indicate the last card.

(c) Cards containing data, if any, in the order in which it is required by the program.

If a second G-program (with its own initial card) is stacked behind the first it will be read in without interruption as soon as the first is finished. The accumulator is cleared between programs, but the number store is not: hence a program too large for the instruction store may be sectionalised, later sections using the results of earlier.

The store for unsuffixed variables is D.L.12, and this is cleared when the "initial input" key is used. If it is desired to avoid this, a G-program may be inserted using the "call read" and "run in" keys, provided control has 0-0x, W=T, any m.c. (e.g. clear and enter D.L.8 with "external tree").

The stores for suffixed variables are in the magnetic drum. A special card is provided to clear them between programs if required: this must be read in, preceded by a blank card, with the "initial input" key. It is in any case normally incorporated in the basic pack (inserted between cards 0 and 1).

The I.D. should normally be clear when a G-program is read in. Certain settings, however, give special test facilities (see below).

2.2 Punching of program

Each program symbol is represented, in the internal operation of the machine, by a combination of 8 binary digits. For convenience, however, symbol-codes are punched in decimal as pairs of numbers in the range 0--15; e.g. "+" is "5/0", "n" is "13/4", etc. (See table in Appendix I). Symbol-codes of a program are punched consecutively up to sixteen per card, the first number of each code in an odd column and the second in the following even column. Code numbers 10, 11, ... 15 are punched as 0--5 with an over-punching in X-row (i.e. X counts as 10).

Code 0/0 is ignored by read; and blank columns are read as zeros, whence unused portions of the Deuce field of a card may be simply left blank.

The last card of a G-program should be punched in Y-row in two successive columns: this stops program read-in when all the symbols on the card have been read. Y-row is otherwise ignored.

2.3 Punching of data

Numbers as data are punched one per card in floating decimal, with sign of mantissa in col. 1, mantissa (≥ 1) in cols. 2-10 with decimal point between cols. 2 and 3; sign of exponent in col. 11, exponent (integer) in cols. 12-20.

Output cards are always in this standard form. The mantissa is however rounded to six significant figures, and the exponent cannot exceed three digits: hence cols. 8-10 and 12-17 will in any case have zeros.

Input cards are essentially in this form, but some flexibility is allowed for ease of manual punching, as follows:-

(i) Plus signs and zeros need not be punched.

(ii) Minus sign of the mantissa may be punched in any of cols. 1-10, and minus sign of the exponent in any of cols. 11-20. (The latter is often conveniently over-punched in col. 20).

Note that figures in cols. 8-10 and 12-17 are in any case treated as zero by the reader.

2.4 "Stops"

The following is a virtually complete list of the "stops" that can occur in the course of a program. Note that in the case of purely arithmetical failures the program can usually be resumed by giving a single-shot, though of course incorrect answers will be obtained.

The annotations "IA" and "II" refer to the two versions of GEORGE.

Indication	Cause	Action
Loop with buzz (on read in or after diag. or EPS card)	Minor cycle synchronisation slip (machine fault)	Non-resumable. Basic program must be read in again
1 12-24x (read in or EPS card) 1 11-1x (diag. card)	Reader operated with machine at "stop"	Switch to "normal"
5 9-24← (data read) 6 9-24← (program read)	S.S. missed (reader fault)	Non-resumable
5 9-24← (program read)	More than 512 symbols in program	Non-resumable
7 0-14x (read called)	Program card required	Supply card or supply "end program read" indication
3 2-2x (IA) 0 2-2x (II)	Mark number used twice	Resumable with S.S. (second entry written over first in mark list)
3 31-29x (IA only)	More than 512 key-words in translated program	Non-resumable
6 0-14, 0 24-28← (IA only)	I.D. has entry in P_{1-31}	Clear and continue

Indication	Cause	Action
7 30-28x 7 15-31x	"Initial stop"	(See below under test facilities)
1 0-0x (II only)	"Test stop"	(See below under test facilities)
1 30-21x (IA) 1 1-1x (II)	Program uses too many accumulator cells or has instructed taking number from empty accumulator	(IA) Non-resumable (II) S.S. calls in next symbol
1 8-21x (IA) 1 3-3x (II)	Too many entries in link list (I.e. nested subroutines or repeats to higher than sixth order)	(IA) Non-resumable (II) S.S. calls in next symbol
7 25-29↰	Program calls for skip to non-existent mark	Non-resumable. Mark number x P_{17} appears on O.S.
3 14-21x	Result of arithmetic operation exceeds permissible magnitude	S.S. resumes with exponent halved
1 4-24x	Divisor zero (+) or argument negative (√)	S.S. resumes giving result zero
1 30-16x	Divisor negative or zero (rem)	"
1 30-21x	Argument zero or negative (log)	"
7 30-21x	Result exceeds permissible magnitude (exp or pow)	"
5 30-21x	Fractional power of negative number (pow)	"
1 16-29x	Programmed result of symbol "wait"	Number in last occupied cell is given on O.S. Program resumed by S.S. Or call and cancel punch (machine at "stop") to obtain a copy on a card
2 0-16x 1 4-15x (II only)	Programmed stops for symbol "I"	Set I.D. and give S.S. Reset I.D. and give S.S.
4 9-24↰	Data card with exponent too large	Non-resumable
1 11-6x 1 7-16x (read called)	Data card wanted	Supply card or emend program
0 0-0x (read called)	End of program	

2.5 Diagnostic routine

A "diagnostic card" is provided which, when run in with the "call read" and "run in" keys, leads to punching of details relevant to the location of a program failure. (Control must have 0-0x, W=T, any m.c. -- e.g. first clear and enter D.L.8). Five cards are punched, as follows:-

Card 1
row Y — keyword track quasi (IA) or symbol-code track quasi (II); number of current track in P_{5-8} in either case.

row X keyword m.c. quasi (IA) with m.c. number of current keyword in P_{26-30}; or symbol-code m.c. quasi (II) with m.c. number of current symbol in P_{17-21}.

0 accumulator quasi, number of occupied cells minus one in P_{17-21}.

1 link list quasi, number of occupied link list cells plus 17 in P_{17-21}.

2 and 3 blank

4-9 contents of link list. Subroutines have simply a keyword or symbol-code address in P_{17-25}; repetitive operations are distinguished by P_{32} and besides a keyword or symbol-code address have the number of the variable of repetition in P_{27-31}, and a count-down number (integer representing the number of complete repetitions still to go) in P_{1-16}.

Card 2 Contents of accumulator.

Cards 3-5 Current track of keywords (IA) or symbol-codes (II); first four rows blank.

Note that the accumulator and link list may have entries beyond the cell shown as the "last occupied", since old entries are not in general erased.

The link list gives keyword addresses in the case of GEORGE IA and symbol-code addresses in the case of GEORGE II. In either case P_{22-25} is track number. In GEORGE II, P_{17-21} is m.c. number. In GEORGE IA to determine m.c. number take P_{17-21}, multiply by 2, and if the result exceeds 31, subtract 31.

At the conclusion of punching, the diagnostic routine leads to 30-28x at the start of the calculation as described under testing facilities below.

Used with the "initial input" key, the diagnostic card leads directly to the beginning of the stored program.

2.6 Program checking facilities in GEORGE IA

When P_{32} is present on the I.D. during translation all first keywords of the translated program will be "stopped", permitting the calculation to be taken step-by-step with the single-shot key. Appendix II gives a list of keywords for identification on the instruction staticiser.

At the end of the translation the presence of P_{32} causes a stop on 7 30-28x. A single-shot leads to the start of the calculation as usual. Alternatively if a single-shot is given with "discrim" key down (giving 7 15-31x), followed by another with "discrim" key normal, the tracks containing the keywords and the mark list are punched out: control is returned to 7 30-28x. Details of the arrangement of keywords and mark list are also given in Appendix II.

2.7 Facilities for manual control in GEORGE II

In GEORGE II, the symbol-codes are not translated into keywords, but instead interpreted one by one as the calculation proceeds, and re-interpreted every time they are used. (Marks are however pre-listed: the mark-symbols

remain in the symbol-sequence but are ignored when encountered). The binary code of each symbol is displayed on the O.S. as it is obeyed, the first number of the code in P_{17-20} and the second in P_{21-24}: this permits identification of any symbol leading to failure.

"Initial stop". If there is anything at all on the I.D. there will be an "initial stop" on 7 30-28x as for GEORGE IA. A single-shot leads to start of program. Alternatively if a single-shot is given with "discrim" key down (giving 7 15-31x) followed by another with "discrim" key normal, the tracks containing the codes and mark list are punched out.

"Stop" and "request stop". So long as P_{32} is present on the I.D. the machine will stop at each symbol. Alternatively if P_{31} is present on the I.D. the machine will stop whenever the code of the symbol about to be obeyed matches the contents of P_{17-24} of the I.D. In either of these cases the machine stops on 1 0-0x, and this is referred to as the "test stop" position. The code of the symbol about to be obeyed is displayed on the O.S., and the symbol will be obeyed normally when a single-shot is given if the I.D. is otherwise clear. (If P_{31} is not present, P_{17-24} must be clear also).

"Slow motion". If P_{30} is present on the I.D. each symbol is "held" for about ¾ sec. before being obeyed. The code of the symbol is displayed on the O.S. during this period and the machine will go to "test stop" 1 0-0x if P_{32} is given.

Manual insertion. When the machine is at "test stop" 1 0-0x program symbols may be inserted one by one manually on P_{17-24} (with S.S. each time: P_{31} must not be present). P_{32} should be present each time and will lead again to "test stop" when the manually inserted symbol has been obeyed. The program will be resumed normally when S.S. is given with P_{17-24} clear

Symbols of the stored program may be "skipped" by giving S.S. with P_1 present. P_{32} must again be present to secure a further stop, i.e. on the next symbol following; or alternatively a "request stop" setting, leading to skipping of all symbols down to the one indicated.

Manual insertion of symbols permits punching out of numbers from store or the accumulator; display of number from the accumulator on the O.S. (using "wait"); skipping to marked positions, etc.; as well as temporary alteration of the course of a faulty program pending re-punching. It must be kept in mind, however, both when inserting and when skipping symbols, that the result depends on the context of the program at the point concerned, and that this context may be altered by the process.

"Final stop". If there is anything at all on the I.D. there will be a "test stop" 1 0-0x at the end of the program. (The O.S. is clear). Symbols may be inserted manually at this point, e.g. for additional punching out of results etc.: otherwise S.S. will lead out in the usual way to "call read" for a further program.

"Restart". Symbol-code 15/15, given manually, restarts the stored program from the beginning, i.e. from initial stop 7 30-28x. The number store is not cleared.

"Looking-in". The accumulator is in D.L.10, m.c. 1-12: m.c.0 has P_{1-10}, which may be used to "set scope" and marks the position of the exponents of stored numbers. Each time the machine goes to "test stop" 1 0-0x it first clears old entries (if any) from unused cells of the accumulator to facilitate inspection.

It should be noted that numeral symbols do not

lead to the insertion of a number in the accumulator until after the first following non-numeral symbol has been called in. (A "comma" may however be inserted manually after the last symbol of the numeral).

The store for unsuffixed variables is D.L.12. While the machine is at "test stop" 1 0-0x it is permissible to use "external tree to call any track of data from the suffixed-variable stores to D.L.11. (Use "cont. TT" key with C=0). For store locations see next section.

The link list is in D.L.10, m.c.18-23. Details of its arrangement have been given above under diagnostic facility.

Note on symbol "I". Since in GEORGE II the I.D. is used for manual control, symbol "I" has been programmed to lead to two stops, the first 2 0-16x to permit the desired input number to be set on the I.D. and the second 1 4-15x to permit the I.D. to be cleared or reset before the program is continued.

2.8 External program synchronisation

In using other programming methods in conjunction with GEORGE it may be necessary to synchronise minor-cycles in order to permit access to the GEORGE data store. An "external program synchronisation" card (or EPS card) is provided for this purpose. When a GEORGE basic program is in the machine, this card may be used with "initial input" key (or "call read" and "run in" keys provided control has 0-0x, W=T, any m.c.) and will perform the usual functions of an initial card, clear D.L.8, and lead out to 0-0x in m.c.0 with the reader running. Provided the basic program (in drum positions 12-15) is not interfered with, an external program may be followed by a further G-program if it finishes by calling read and giving 0-0x, W=T, any m.c.; or a further G-program may be inserted using the "initial input" key as usual if the unsuffixed-variable store need not be preserved.

Store locations are as follows:

Unsuffixed variables: D.L.12, with m.c. in order corresponding with variable numbers (i.e. five least significant digits of binary symbol-code).

Singly-suffixed variables: positions 8 and 9 of the drum. Tracks in order correspond with variable numbers and m.c. correspond with suffix-values.

Doubly-suffixed variables: positions 0-7 of the drum, variable "a" in positions 0 and 1, "b" in 2 and 3 etc. For each variable, tracks in order correspond with values of first suffix and m.c. correspond with values of second suffix.

Appendix I

GEORGE

CODING TABLE

	0	1	2	3	4	5	6	7	8	15
0	(not used)	|	0	$\overline{16}$	a	q	(a)	(q)	log	R
1	,	||	1	$\overline{17}$	b	r	(b)	(r)	exp	(P)
2	;	~	2	$\overline{18}$	c	s	(c)	(s)	pow	
3	*	&	3	$\overline{19}$	d	t	(d)	(t)	rem	
4	wait	∨	4	$\overline{20}$	e	u	(e)	(u)	√	
5	+	!	5	$\overline{21}$	f	v	(f)	(v)	sin	
6	-	↓	6	$\overline{22}$	g	w	(g)	(w)	cos	
7	×	↑	7	$\overline{23}$	h	x	(h)	(x)		
8	÷	rep	8	$\overline{24}$	i	y	(i)	(y)		R_I
9	neg	I	9	$\overline{25}$	j	z	(j)	(z)		P_I
10	mod		$\overline{10}$	$\overline{26}$	k	α	(k)	(α)		R_{II}
11	max		$\overline{11}$	$\overline{27}$	l	β	(l)	(β)		P_{II}
12	dup		$\overline{12}$	$\overline{28}$	m	γ	(m)	(γ)		
13	rev		$\overline{13}$	$\overline{29}$	n	λ	(n)	(λ)		
14	=		$\overline{14}$	$\overline{30}$	θ	μ	(θ)	(μ)		
15	▶		$\overline{15}$	$\overline{31}$	p	ν	(p)	(ν)		

The code of each symbol is its row number followed by its column number. For example "+" is 5/0, and "n" is 13/4. Columns 9-14 are not used. Following an asterisk, symbols of column 3 represent numbers $\overline{16}$-$\overline{31}$ as shown: otherwise any symbol of column 3 represents decimal point.

Appendix II

Keywords corresponding with the various symbols

In a few cases keywords go in pairs: in these cases the first always has source 2 and represents an instruction to place the second in a specified location.

G-code symbol	First keyword	Second keyword (if any)
,	(not translated)	
;	1 4-15	
*	(not translated; following numeral also not translated)	
wait	1 14-30	
+	4 10-1	
-	5 10-1	
×	4 4-15	
÷	4 3-15	
neg	3 6-13	
mod	3 5-13	
max	1 7-30	
dup	1 3-24	
rev	6 3-24	
=	4 7-30	
>	3 7-30	
| (with following symbol)	1 2-14	variable number x P_{17} + P_{32} if name.
|| (with following symbol)	3 2-14	"
~	1 10-30	
&	4 10-30	
v	3 10-30	
]	3 12-30	
↓	1 15-30	
↑	4 15-30	
rep (with following name)	4 2-14	variable number x P_{17}
Numeral sequence	1 2-16	number
Variable (unsuffixed)	1 12-16	
Name (unsuffixed)	1 2-15	1 16-12
Col. 8 operations	5 2-14	G-code x P_{26}
Read and punch (single)	5 2-14	G-code x P_{26}
Read and punch (vector and matrix)	5 2-14	G-code x P_{26} plus name-symbol x P_{17}
End of track	1 17-21	
End of program	1 13-31	
I	1 0-16	

Appendix II (contd.)

The "number" of a variable (or name) is given by the five least significant digits of its binary symbol-code. Suffixed variables and names are translated together with their bar-signs and the variable number is given by the second keyword corresponding with a bar-symbol, as shown. In the case of an unsuffixed variable the variable number may be inferred from the wait number of the keyword by adding two plus the number of the minor cycle in which it is stored (modulo 32); and in the case of an unsuffixed name the variable number may be inferred from the second keyword by subtracting 9 (modulo 32).

When keywords are punched out using GEORGE IA on diagnostic punch or as described in the section on program checking they are in triads and in the latter case are headed by their track number in P_{1-8} of Y row (tracks are 0-15 of position 10). In each track the keywords are interlaced, i.e. the first sixteen in even minor cycles 0, 2, ... 30 and the second sixteen in odd minor cycles 1, 3, ... 31. Minor cycle 29 or 31 always has 1 17-21, leading to a routine to call the next track of keywords: if this instruction is in m.c.29, m.c.31 is blank. The mark list is a single track (12/0) with entries in minor cycles corresponding with mark numbers used in the program, other minor cycles having P_{32}. Each entry gives the track and m.c. of the "marked" keyword in P_{5-8} and P_{26-30} respectively.

Punching of codes using GEORGE II is from tracks 0-15 of position 11, and the codes are in P_{17-24} of successive minor cycles. The mark list addresses have track number in P_{5-8}, repeated in P_{22-25}, and m.c. number in P_{17-21}.

Acknowledgements

Acknowledgement is due to Mr G. Karoly, and to Messrs R. Brigham and G. Bell, and to Mr R. Smart, for stimulation and suggestions.