K Autocode

By A. Gibbons*

This paper describes a development of Mercury Autocode (Brooker, 1958) known as K Autocode. The first implementation of K Autocode was for the English Electric KDF9 and the second for the IBM System/360. In addition to the usual numerical facilities and procedures, the language provides comprehensive input/output, random and sequential file processing, symbol manipulation and a matrix scheme.

(First received October 1967)

In 1962, ICI ordered a KDF9 to replace its existing Mercury computer. When the KDF9 was delivered in February 1964, the Mercury was being used on a 3-shift basis for five or more days a week and the workload was programmed almost entirely in Mercury Autocode. There was an Autocode library of over 100 programs, several hundred personnel were conversant with the Autocode language and over 2,000 programs were in use. It was the intention at that time to use Autocode on both machines during the transition period and gradually to convert programs and programmers to ALGOL.

A compatible compiler was written and came into use in May 1964. Another compiler was also provided which contained extra facilities. These were welcomed and suggestions were made for further improvements. As a result, K Autocode was defined and a preliminary implementation made available in October 1964. Subsequently the language has been extended and, because of the relative inefficiency of the ALGOL implementations, is now the most widely used technical programming language in ICI.

In October 1965, the first of a series of System/360 computers was installed and a 360 implementation of K Autocode started. At the present time, March 1968, nine 360s have been delivered and K Autocode is available on the larger models.

For the future, PL/I is probably the preferred language but the inefficiency of the initial implementation in relation to K Autocode does not yet make this an attractive proposition.

The description of the language which follows is far from complete and is intended to give a general understanding only. More information can be obtained from:

The Information Officer,
ICI Management Services,
Fulshaw Hall,
Wilmslow,
Cheshire.

Abbreviations

A, *B*	upper or lower case letters
Ap, *Bq*	integer or real array variables
*A*0, *B*0	base elements of arrays
E, *F*	integer or real expressions
I, *J*, *K*, *L*	integer expressions
Ip, *Jq*	integer array variables
IV	integer variable
N	unsigned integer
R	relation
Xp, *Yq*	real array variables
RV	real variable
S	a language symbol
U, *V*	integer or real variables
X, *Y*, *Z*	real expressions

1. The elements of the language

1.1 The alphabet

The alphabet of the language consists of 86 symbols:

$a\ b \ldots z\ A\ B \ldots Z\ 0\ 1 \ldots 9$
$\pi + - / \neq < = > . , ; : ( ) [ ]\ '\ \phi * ?\ _{10}\ \_$

and the non-printing symbols:

'space' (sp) and 'newline' (nl)

The symbols of this alphabet are represented in the IBM 60 character set either directly

(01 . . . 9 + − / < = > . , ; : ( ) ′ * ? _)

or as follows:

ϕ	as	%
[	,,	⏋
]	,,	|
$_{10}$	,,	≠
π	,,	£ (or $)
a–*z*	,,	*A*–*Z*
A–*Z*	,,	≠*A*–≠*Z*
$\neq$	,,	⏋=
sp	,,	blank column
nl	,,	end of card

The space symbol only has significance in messages which may be printed.

Statements in the language are terminated either by a newline or by a semi-colon.

* *ICI Management Services, Fulshaw Hall, Wilmslow, Cheshire.*

1.2 Instructions and directives

There are basically two kinds of statements in a programming language: instructions, and directives. The instructions of a program are held inside the computer and are obeyed when the program is run. Directives, on the other hand, are orders to the compiler and do not exist at run time. The distinction between instructions and directives is sometimes nebulous.

1.3 Numbers

Essentially two types of number exist in Autocode, *integer* and *real.*

An integer number is stored exactly and lies in the range $-2^{\beta-1}$ to $2^{\beta-1} - 1$, where β is the number of bits used for integer arithmetic (on KDF9, $\beta = 48$ and on System/360, $\beta = 32$).

A real number is stored in a floating point representation. Its modulus is either zero or lies in the range $10^{-\alpha}$ to $10^{+\alpha}$ (on KDF9, $\alpha = 38$ and on System/360, $\alpha = 75$). The accuracy to which a real number can be stored is δ significant decimal digits. (On KDF9, $\delta = 11$ and on System/360, $\delta = 16$.) The number of bits used to hold either type of number is γ. (On KDF9, $\gamma = 48$ and on System/360, $\gamma = 64$.)

1.4 The computer store

The storage provided by Autocode falls into two categories, *main* and *auxiliary*. The main store can hold 1,024 words of instructions and at least 6,144 numbers. The block of instructions is known as a *chapter* and all programs are divided into chapters. The auxiliary store can hold 102,400 numbers although part of this store is used to hold the Autocode program.

All variables are referred to by single letters, integer variables by the letters *i–t* and real variables by the letters *a–h* and *u–z*.

There are two kinds of variables, *scalar* or *special* variables and *array* or *working* variables. Space is allocated to an array by the directive

$A{:}N$

which allocates space to the $N + 1$ variables

$A0, A1, \ldots, AN$

Special variables do not need to be declared.

1.5 Comments

Programs may be annotated with *comments*. A comment consists of a string of symbols enclosed in brackets and is ignored by the compiler.

2. Integer and real assignment instructions

An *assignment instruction* is used to assign a value to a variable. An *expression* is a rule for computing a value, and an assignment instruction consists of a variable and an expression.

Expressions are classified by the types of values they define. Thus, in particular, there are integer expressions, denoted by I, J, K and L, and real expressions, denoted by X, Y and Z. Expressions are also used as subscripts and arguments of functions—in fact wherever a value is required.

There are 104 special variables

$a, b, \ldots, z, A, B, \ldots, Z, a', b', \ldots, z', A', B', \ldots, Z'$

and up to 52 arrays with names

$a, b, \ldots, z, A, B, \ldots, Z$

A reference to an array variable consists of the name followed by a subscript. A subscript is either an integer or an integer expression in brackets. In addition, the subscript of a real array variable can be an integer variable. Thus

$i, J', k3, P(Q(R))$ are integer variables and
$a, B', c3, Di, E(j + 3)$ are real variables.

2.1 The constants facility

A list of constants may be stored in a chapter and referred to as an array, by preceding the list with the directive

constants A

where the letter is of the same type as the constants which follow. The constants are referred to as

$A0, A1, \ldots$

Integer constants are preceded by the symbol =, thus:

constants i
=0; = −1; =77

Real constants may be written with a decimal point and a decimal exponent and are preceded by + or −, thus:

constants a
+1
−2·4
−3·5, 6
+1, −7

The constants are only available in the chapter in which they are declared.

2.2 Functions

A *function* in K Autocode is written as a ϕ followed by the name of the function and possibly an argument list in brackets.

A function, like an expression, defines a value and is classified by the type of value it produces. The definition of an integer function (a function which defines an integer value) is preceded by

$\phi i{:}$

and the definition of a real function by

$\phi x{:}$

2.2.1 Elementary functions

In addition to the elementary mathematical functions, the following functions are available:

ϕx: $\phi radius\,(X, Y)$	Result is $\sqrt{(X^2 + Y^2)}$
ϕi: $\phi sign\,(X)$	Result is $\begin{cases} -1, & X < 0 \\ 0, & X = 0 \\ +1, & X > 0 \end{cases}$
ϕi: $\phi minusone\,(I)$	Result is $(-1)^I$
ϕi: $\phi divide\,(I, J, IV)$	Result is the quotient of I/J and remainder is assigned to the integer variable IV

If the value of an argument is found to be outside the permitted range, control is transferred to the instruction labelled 100, if this exists. (See § 3.1 for description of labels.) The programmer can then use the function $\phi error$ (which has an integer result in the range 0–21) to determine the cause of the error. If there is no label 100, the program is terminated and a postmortem given (see § 9.4).

2.2.2 Array functions

The functions described below have a one-dimensional array as one argument.

ϕi: $\phi max\,(Ap, I, J)$

Here Ap is the first element in a set of either real or integer array variables. The function scans the set of variables

$A(p + I)$ to $A(p + J)$

and gives as its result the position of the maximum element of the set relative to Ap, i.e. the maximum element itself is

$A(p + \phi max\,(Ap, I, J))$

If two elements of equal value qualify for selection, the one with the smaller subscript will be chosen.

ϕx: $\phi poly\,(Yq, X, I)$

This function evaluates the polynomial obtained by taking $I + 1$ coefficients from the real array commencing with Yq, i.e. the result is

$$\sum_{i=0}^{I} Y(q + i)\, X^i$$

ϕx: $\phi chebyshev\,(Yq, X, I)$

This function gives as its result the value of the expression

$$0{\cdot}5\ Yq + \sum_{i=1}^{I} Y(q + i) Ti(X)$$

where $Tr(X)$ is the Chebyshev polynomial of degree r.

2.2.2.1 Array instructions

In addition to the matrix instructions described in § 8, there are

copy (Ap, Bq, I)

which copies I variables commencing with Bq, one at a time, into Ap, $A(p + 1)$, . . . and

clear (Ap, Aq)

which sets the variables Ap to $Aq (q \geqslant p)$ to zero.

2.2.3 Logical functions

These functions operate on integer values and depend upon the fact that these values are held in binary format. Hence they imply a measure of machine dependence.

The functions

ϕi: $\phi nonequiv\,(I, J, \ldots)$
ϕi: $\phi and\,(I, J, \ldots)$
ϕi: $\phi or\,(I, J, \ldots)$

are unusual in that they may have a variable number (two or more) of arguments.

ϕi: $\phi shl\,(I, J)$
ϕi: $\phi shc\,(I, J)$
ϕi: $\phi sha\,(I, J)$

In these shift functions (logical, cyclic and arithmetic) the value I is shifted, left or right, through J places.

ϕi: $\phi bits\,(I)$

The result of this function is the number of 1 bits in I.

2.2.4 Packed data

Packed data may be extracted from an array of integer variables by the function

ϕi: $\phi string\,(Ip, J, K)$

which selects K bits, $K \leqslant \beta$, from the integer array commencing at Ip, starting with bit J. The reverse operation, that of replacing a string, is performed by the instruction

storestring (Ip, J, K, L)

which selects the least significant K bits of L and stores them in bit positions $J, J + 1, \ldots$ in the array commencing with Ip.

2.2.5 Calculation of pseudo-random numbers

The function

ϕx: $\phi random\,(IV, 1)$

where IV is an integer variable, calculates the next value in a sequence of rectangularly distributed pseudo-random numbers in the range 0 to 1. The particular sequence selected depends upon the initial value assigned to IV.

The function

ϕx: $\phi random\,(IV, 2)$

calculates the next value in a sequence of normally distributed pseudo-random numbers in the range -6 to $+6$. The distribution has nominal zero mean and unit standard deviation.

2.3 Expressions

Expressions are built up from constants, variables, functions of expressions and expressions in square brackets. Multiplication is indicated by the juxtaposition of the two operands and takes precedence over division. Hence a/bc means $a/[b \times c]$. The factors of an expression are always evaluated from left to right. Exponentiation is not permitted.

Integer expressions consist solely of integer factors. They are evaluated with fixed point arithmetic and division is not permitted.

Examples:

$3[i + j]$; $pq + \phi intpt\,(x + y)$

Real expressions may contain both real and integer factors and the constant π. Real expressions are evaluated with floating point arithmetic.

Examples:

$3{\cdot}1\ x + y/4z$; $\phi\exp\,(\pi y\phi\log(x))$

When an expression is enclosed in square brackets, its value may be assigned to a variable of the same type by writing

$[E = > V]$

The assignment is performed immediately the expression has been evaluated. For example

$i([j + 1 = > j]) = p$

is equivalent to

$j = j + 1$
$i(j) = p$

2.4 Assignment instructions

An assignment instruction consists of one or more variables and an expression.

Examples:

$i = 1$; $p(q) = q = r + 3s - t$
$x = y = z - 2\pi\phi\sin\,(a)$

In performing an assignment, the expression on the right is evaluated first. Its value is then assigned to the left parts, one at a time, from right to left. Thus

L1 = L2 = . . . Ln = E

is equivalent to

W = E; Ln = W; . . . L2 = W; L1 = W

where W is a working space location.

A query symbol, ?, may be written at the left of an assignment statement. If the query option is specified in the job heading (see § 9.4) the value of the expression in a queried assignment will be printed each time the instruction is obeyed. If the program is run 'without queries', all query symbols are ignored.

3. Control statements

3.1 Labels

A *label* consists of an integer followed by a right bracket

N)

The directive

labels: *N*

where $1 \leqslant N \leqslant 1023$, may appear before the first instruction of a chapter. It specifies the extent of the labels field to be used within the chapter, and consequently how much space remains for instructions. After the directive, labels 1 to *N* inclusive may be used. If the directive is omitted,

labels: 127

is assumed.

3.2 Unconditional jump instructions

jump (*I*)

where *I* is an integer expression, transfers control to the instruction labelled *I* in the current internal procedure or chapter. Hence it can be used as a multiway switch.

3.3 Subsequences

The instruction

jumpdown (*I*)

in addition to transferring control, places the address of the instruction following the jumpdown in a stack.

return

transfer control to the address at the top of the stack.

3.4 Conditional operations

Several facilities depend upon the truth or falsity of a relation *R*. A relation may be either a simple relation, i.e. a comparison of two expressions, or a logical combination of simple relations.

Examples:

$i \neq j$
$(a > b)$ *and* $(b > c)$ *or* $(x = y)$
not $((i < j)$ *nonequiv* $(m < n))$

In a relation, all the operations have the same precedence, and evaluation proceeds from left to right. The order of evaluation may be modified by using brackets.

Each simple relation gives rise to a numerical value of -1 or zero depending on its truth or falsity. These values are combined by the logical operators and the relation is true if the result is non-zero.

3.4.1 Conditional expressions

Expressions in square brackets may be made to depend upon relations

$[(R1(E1, (R2)E2, \ldots, E_{n+1}$

Example:

$[(a > b)c, (d < e)f, g]$

To find the value of the expression, the relations are tested in order from left to right. The expression following the first true relation is taken as defining the value of the conditional expression. If none of the relations is true, the unqualified expression, E_{n+1} is used. A conditional expression may be used wherever a value is required.

3.4.2 Conditional instructions

The instructions

```
if(R)
orif(R)
else
continue
```

may be used to select alternative courses of action in a similar way to conditional expressions.

Example:

```
if(a = 0); print('a = 0')
orif(a < 0); print('a < 0')
else; print('a > 0')
continue
```

3.4.3 Conditional jump instructions

Both the jump and the jumpdown instruction can be made conditional by appending a relation.

Examples:

```
    jump (1) a > b
jumpdown ([(a > b)i, j]) (c > = d) and (e < =f)
```

3.5 Repeat cycles

```
IV = I, J, K
repeat
```

These instructions are used to form a repeat cycle. Their effect is equivalent to

```
    IV = I
1)  . . .
    . . .
    jump (2) IV = K
    'error if φsign (K − IV) ≠ φsign (J)'
    IV = IV + J
    jump (1)
2)  . . .
```

3.6 Terminating the calculation

end

This instruction returns control to the operating system and signifies that the computation has finished correctly.

4. Input and output

The facilities described in this section are concerned with reading data from the input stream (i.e. data immediately following the program) and printing results. Only a single stream of input and output is considered (but see § 7.1).

Autocode does have facilities for input and output to other peripheral devices and these are described in § 5.

4.1 Reading decimal numbers

When a read instruction, which specifies n variables, is obeyed, the next n data values are assigned to the n variables one at a time from left to right.

Example:

read (a, b, i, wi)

would accept as data

```
  3·1; −4·2, 17
  5; (code)
203; (weight)
```

This would assign $3\cdot1$ to a, $-4\cdot2 \times 10^{17}$ to b, 5 to i and 203 to $w5$. Comments can be placed between data values which are terminated by newline, semi-colon or double space.

4.2 Printing decimal numbers

A print instruction specifies a list of items to be printed. An item may be a string of symbols in quotes, a single symbol to be printed a number of times, or an expression.

Example:

If $x = 321\cdot4$ and $y = -0\cdot75$ the instruction

```
print('temp=' ; x, 3, 2; (.)10; 'press=' ; y, 0, 4)
```

produces

```
temp= 321·40 ..........press=−7·5000, −1
```

The layout of a printed number is determined by the two expressions I and J written after it.

If I and J are non-zero, the layout of the number is ⟨sp or −⟩⟨I integral digits⟩.⟨J fractional digits⟩⟨sp⟩⟨sp⟩ If J is zero, the decimal point and the fractional digits are omitted. Non-significant zeros at the leading end are replaced by spaces.

If I is zero, the number is printed in floating point style ⟨sp or −⟩⟨digit⟩·⟨J fractional digits⟩,⟨decimal exponent⟩ ⟨sp⟩⟨sp⟩. If J is zero, the decimal point and the fractional digits are omitted.

4.3 The readword instruction

readword $(V, IV1, IV2, IV3, I)$

This instruction is used to read in a string of symbols, or word. Any numeric symbols are assembled to form a value which is assigned to the variable V. The sum of the internal codes of any non-numeric symbols occurring in the word is assigned to *IV*1. The number of numeric symbols is assigned to *IV*2 and the number of non-numeric symbols to *IV*3. If the value of the numeric symbols is outside the range permitted for *V*, or if any non-numeric symbols are encountered, control is transferred to the instruction labelled *I*. The integer function

ϕcode('*S*1*S*2*S*3 . . .')

gives the sum of the internal codes of *S*1, *S*2, *S*3, . . .

4.4 Manipulation of texts

A text is a string of symbols enclosed in square brackets. A text may be read from the input stream and packed into an array of integer variables, printed out, simply copied, attached to another text or moved from the chapter space to an array. In addition, the individual symbols of a text can be inspected and replaced.

printtext (*Ip*)

prints the text (but not the enclosing brackets) which is stored in the integer array starting at *Ip*. The output is terminated by the closing square bracket.

concatenate (*Ip*, *Jq*, *IV*)

attaches the text stored in *Jq* to the text stored in *Ip*, eliding the internal pair of brackets. The final length is returned in *IV*.

storetext (*Ip*, *IV*)
[*symbols*]

stores the *symbols* in the array starting at *Ip* and assigns the number of variables used to *IV*.

ϕi: *ϕsymbol* (*Ip*, *I*)

gives as its result the *I*th symbol packed into the array starting at *Ip*.

storesymbol (*Ip*, *I*, *J*)

is used to replace the *I*th symbol packed into the array starting at *Ip* with the value of *J*.

4.5 Input and output of symbols

A number of instructions and functions are available for reading, printing and testing symbols.

The function

ϕi: *ϕreadsymbol*

gives as its result the next symbol in the data stream.

The function

ϕi: *ϕlastsymbol*

gives the result of the most recently obeyed *ϕreadsymbol*. This function can be used after any read instruction, e.g. *read*(*x*), to determine the terminator.

The function

ϕi: *ϕnextsymbol*

gives the symbol that will be read by the next *ϕreadsymbol*. This function can be used to advantage before a read instruction.

The function

ϕi: *ϕcode* (*S*)

gives the internal code of the symbol *S*.

The instruction

printsymbol (*I*)

prints the symbol whose internal code is given by *I*.

4.6 Miscellaneous output facilities

caption
symbols

causes the *symbols* to be printed (excluding the newline at each end)

nlcaption
symbols

causes a newline and the *symbols* to be printed. In addition there are

space
spaces (*I*)
newline
newlines (*I*)
newpage

5. File processing

Autocode allows files on peripheral devices to be accessed either randomly or sequentially. On KDF9, only magnetic tapes may be used, but in either mode. On System/360, any peripheral may be used but tapes, printers and card devices can only be accessed sequentially. File labelling methods also differ. A file is connected to the Autocode system via a fictitious channel when the file is claimed or opened.

5.1 Random access

A random access file consists of a continuous set of locations numbered 0, 1, ... Transfers of information take place between these locations and array variables.

readfile (*I*, *J*, *Ap*, *K*)

transfers *K* numbers from locations $J, J + 1, \ldots$ on the file on channel *I* to the array variables $A_p, A_{p+1}, \ldots$.

writefile (*I*, *J*, *Ap*, *K*)

transfers *K* numbers from an array to the file on channel *I*.

5.2 Sequential access

In this mode of operation the locations of a file are grouped into blocks called *records*. Records need not all be the same size and are not numbered.

readrecord (*I*, *Ap*, *J*)

transfers the contents of the next record from the file on channel I to the array variables A_p, A_{p+1}, ... If n is the number of numbers in the record, the actual number transferred is *min* (n, J).

writerecord (I, Ap, J)

transfers a block of J numbers from an array to the file on channel I. Records can only be written to the end of a sequence of records, i.e. it is not possible to read a record that may remain on the file beyond the record just written.

rewind (I)

causes the file on channel I to be positioned with the recording head at the load point.

skiprecord (I, J)

causes the file on channel I to move J records past the recording head. The skip will be forwards if J is positive and backwards if J is negative.

There are also facilities for testing the state of 'parity' and whether either end of a file has been reached.

6. Facilities for complex arithmetic

A complex variable is denoted in Autocode by a pair of real variables, the real and imaginary parts, enclosed in single brackets. The functions *sin*, *cos*, *exp*, *log*, *sqrt* and *conjugate* may be used in expressions.

Examples:

$(a, b) = (c, d)\,(u1 + u2, v1 + v2)$
$(x, y) = \phi sqrt\,((x, y))$
$(a, b) = (b, a)$

Apart from query printing, there are no special instructions for reading and printing complex numbers. These operations can easily be done with the real instructions, e.g.

read (x, y)
print $(X, 1, 6;\ Y, 1, 6)$

7. The auxiliary store for numbers

The auxiliary store consists of 102,400 locations numbered 0, 1, . . ., 102399. Some of these locations are used to hold the chapters of the program. The instruction

readdown (I, Ap, J)

transfers J numbers from auxiliary locations $I, I + 1, \ldots$ to the array variables A_p, A_{p+1}, ... A single number may be transferred to a special variable. The instruction

writeup (I, Ap, J)

causes a transfer to take place in the opposite direction.

7.1 Auxiliary store input output

The auxiliary store may also be thought of as a large array of symbols numbered 0, 1, ... The instruction

open output (I)

routes all subsequent output (apart from failure messages) to auxiliary store symbol positions $I, I + 1, \ldots$. The instruction

close output (IV)

terminates the current stream and returns the address of the last symbol in IV, so that the stream can be re-opened. The instructions

open input (I)
and *close input* (IV)

are used to control the source of input in a similar way.

In Autocode, it is possible to return to the compiling mode and compile additional items of program (*rmp* instruction, meaning read more program, see § 9.5). If auxiliary store input is used, the program to be compiled can be held in symbolic form in the auxiliary store. Consequently a translation program, e.g. Mercury Autocode to K Autocode, could place its output in the auxiliary store and then obey the sequence

open input (. . .)
rmp

to compile it. The net effect is to present the user with something very like a Mercury Autocode compiler.

8. The Autocode matrix scheme

The matrix scheme permits operations to be performed on arrays of numbers held in the auxiliary store. The auxiliary address, M, of the first element is used to refer to the array. Two dimensional arrays are stored by rows. The dimensions of the operands are stated explicitly in each operation.

Each matrix operation is essentially an assignment instruction, although it is written in the same way as a function:

ϕoperation number (parameter list)

The matrix operations provide facilities for input, output, manipulation and arithmetic. Three typical operations are given.

$\phi 5\ (M, I, J, K)$

prints the $I \times J$ matrix in M in tabular form. The elements of the matrix must be real numbers and they are printed in floating point style with K significant figures. As many columns as possible are printed across a page.

$\phi 11\ (M1, M2, I)$

extracts the diagonal of the square matrix of order I stored in $M2$ and stores it as a vector in $M1$.

$\phi 30\ (M1, M2, M3, X, Y, I, J)$

forms the linear combination of two $I \times J$ matrices stored in $M2$ and $M3$, i.e.

$$(M1) = X\,(M2) + Y\,(M3)$$

The special cases which arise when X and Y are 0 or 1 are recognised.

9. Program assembly

9.1 Chapters

A progam is divided into one or more chapters each of which may contain about 200 Autocode instructions. A chapter is preceded by the directive

chapter N

where $0 \leqslant N \leqslant 100$ and terminated by the directive

close

The chapters which comprise a program must be compiled in monotonically increasing order with the exception of chapter 0 which must appear at the end. A program is always entered at the first instruction of chapter 0. The instruction

across (I, J)

transfers control to the instruction labelled I in chapter J. The instruction

down (I, J)

is similar but, in addition, stores the current chapter number and the address of the instruction following the *down* in a stack. The instruction

up

takes the most recently entered chapter number and address from the stack and transfers control to this address and chapter.

9.2 Internal procedures

Procedures in Autocode are identified by numbers and enclosed by the directives

procedure N
and *finish*

An internal procedure is contained within a chapter. It may only be called from within that chapter. Full recursion is allowed.

Three instructions are used to establish communication between a calling program and a procedure. They are

```
call (I, J; E1, E2, . . .; Ap, Bq, . . .; V1, V2, . . .)
data (U1, U2, . . .; A, B, . . .)
results (F1, F2, . . .)
```

The *data* and *results* instructions serve as the entry and exit points in a procedure, and the *call* instruction is used to transfer control to label I in procedure J.

When the *call* instruction is obeyed, a private set of special variables is provided for the procedure and its own list of array bases is set identical to the current set of array bases, providing access to global arrays. Next, the values of $E1, E2, \ldots$ and the addresses of $Ap, Bq, \ldots$ are passed to the procedure. When the *data* instruction is obeyed, these values are assigned to $U1, U2, \ldots$ (any type conversion being done automatically) and the addresses are used as the bases for the formal arrays $A, B, \ldots$. When the procedure has completed its task, the *results* instruction is obeyed, the values of $F1, F2, \ldots$ are passed back to the *call* instruction and assigned to the variables $V1, V2, \ldots$

In addition to global and formal arrays, a procedure may declare local arrays with dynamic bounds by the instruction

local $A: I$

Inside a procedure, there is a dynamic equivalence facility. The instruction

equivalence (A, Bq)

assigns the address of Bq to the base for the array A. References to $A0, A1, \ldots$ are then references to $Bq, Bq + 1, \ldots$

A simple procedure to calculate the average of a set of real numbers could be

```
   procedure1
1) data (n; x)
   a = 0
    i = 0, 1, n − 1
   a = a + xi
   repeat
   results (a/n)
   finish
```

which could be called with the instruction

call $(1, 1; p + 1; ym; b)$

to calculate the average of $y(m)$ to $y(m + p)$ and assign the result to b.

9.3 External procedures

An external procedure consists of one or more chapters enclosed in the directives

procedure N
and *finish*

Several external procedures may be included between the main program and chapter 0.

An external procedure may be called from anywhere in the program, by the same calling sequence as for an internal procedure, and it is always entered at its first chapter. External procedures may contain internal procedures.

9.4 Job heading

When an Autocode job is submitted for processing, it is preceded by an Autojob heading. This consists of the word 'autojob', optionally some message for the operator, and a job statement which specifies the estimated and maximum running times and any of five options. This statement is followed by the title which identifies the job.

Example:

```
autojob
this job requires three magnetic tapes.
job (5/10, p, q, r, s, t)
title
demonstration . . .
```

In this example, the job statement specifies an estimated running time of 5 min, a maximum time of 10 min and all the options. The various options are:

p to produce a postmortem in the event of a failure, i.e. the names and values of all variables currently defined.
q to run the job in the query printing mode.
r to restart the job at the first instruction in chapter 0 after a failure. This allows several sets of data to be processed even if one set unexpectedly fails.
s to produce a listing of the source program.
t to produce a listing of the translated program.

If an option is not required, it is omitted, e.g. *job* (5/10, *s*).

9.5 Reading more program

The instruction

rmp

meaning read more program, causes the system to return to the compiling mode and read any monotonically increasing sequence of chapters terminated by a chapter 0. The contents of the number store are preserved during the operation. Control is then passed to the first instruction of the new chapter 0.

The instruction is useful since it permits a linked sequence of programs to be run, e.g. an amender and the amended program, and also since it allows one or more chapters to be specified as data, e.g. transformations to be applied to numerical data.

Acknowledgments

The author is indebted to Professor R. A. Brooker for introducing him to syntax-directed compilation and to his colleagues, both past and present, in I.C.I. Ltd. for the success of the implementations on KDF9 and System/360.

Reference

BROOKER, R. A. (1958). The Autocode Programs developed for the Manchester University Computers, *Computer Journal*, Vol. 1, p. 15.

Book Review

Automaton Theory and Learning Systems, Ed. D. J. STEWART, 1966, 215 pp. (London: *Academic Press*, 63s.)

This book is a collection of papers with a strong 'cybernetics' slant. It opens with a brief and very readable paper by J. C. Sheperdson introducing the mathematical notion of algorithm, and the need for a formal description of algorithms such as those of Turing and of Markov to show that some problems have no algorithm for deciding them. The other papers are unconnected with mathematical automata theory. Two papers (one by F. H. George and D. J. Stewart and one by George) describe the elements of computer programming in machine code with a few pages on logic and on phrase structure grammars. One by W. Ross Ashby lists a number of definitions and theorems of the algebra of sets and indicates how they might be used to formulate a theory of mechanisms and homeostasis more clearly than by continuous variable theory. Another paper by Stewart explains the use of logical nets for classification and learning, and a rather lucid contribution by A. M. Andrew surveys various approaches to learning, discussing topics such as parameter adjustment versus the use of a selection of key points in the input space, the need for modelling the environment and hierarchical organisation. A long and technical paper by G. Pask describes a theoretical model for the acquisition of skills and some supporting experimental work using computer simulation and various adaptive mechanisms for training subjects. A training machine controls learning by adjusting the difficulty of the problems presented. This is the only contribution which sets out to present any appreciable amount of research by the author.

In general it would have been encouraging to be told more of mechanisms or computer programs which have actually been constructed and which exhibit interesting learning or problem solving behaviour. The point has long since been made that such performance is possible in principle, the question is how to achieve in practice. Most of the book is written at the level of rather elementary exposition and readers seeking an introduction to the field would probably have preferred a more unified treatment instead of a collection of papers by different authors.

R. M. BURSTALL (Edinburgh)