The main features of Atlas Autocode

By R. A. Brooker, J. S. Rohl, and S. R. Clark*

This paper describes the main features of an ALGOL-type compiler that has been used very successfully with the Manchester Atlas for over two years. Since we are dealing with a working system special attention is given to the input/output and fault monitoring facilities, and also to the efficiency of the system. Not being bound by any particular dogma, the authors have felt free to play with the system, and some indication of recent developments is given.

"Why not ALGOL?" is a question we get asked from time to time. The reason is simply that when we started on AA (Atlas Autocode) it was far from clear, to us, whether ALGOL 60 (see Naur *et al.*, 1963) could be implemented sufficiently well (with the effort at our disposal) to serve as a main programming language, and we had the responsibility of providing such for the Manchester University installation. We hoped we could retain some of the more important ideas of ALGOL, e.g., dynamic storage allocation, in a somewhat simpler structure, and because we had in mind the provision of numerous special facilities (e.g., list processing, special forms of arithmetic) we would not have been content with a subset. Such facilities would be intolerably inefficient (and inconvenient to use) if implemented as formal procedures—the only existing means of extending ALGOL 60. We have of course paid a price for going it alone, namely access to the ACM library of Algorithms, but fortunately most of them can be converted to AA very easily, as we have found with those of interest. We have also found that they could benefit by conversion to machine code, at least in the inner loops.

It is appropriate to describe the main features of the language in ALGOL terms.

Alphabet

The basic characters on the Atlas Flexowriters are:

A . . . Z a . . . z 0 . . . 9
+ − * / | ( ) = < > _ , . : ′ ½ ² [ ] α β π ?

From these it is possible with the backspace facility to form compound characters, e.g., ≠ ⩾ ⩽ ; ⊆ ↑
[α and ↑ do duty for the ALGOL symbols $_{10}$ and ↑, but to avoid confusion we shall use the latter notation in this paper.]

Names

These consist of a letter optionally followed by further letters, digits, primes in that order, e.g.,

A temp a1′ A2 x″.

The length is virtually unrestricted.

Delimiter words

These are underlined as in ALGOL, e.g., **real, cycle**

Types

These are limited to **real** and **integer** (there is no **Boolean**) but otherwise declarations are as in ALGOL. Further types of variables have recently been introduced to facilitate complex and multiprecision arithmetic. These are briefly described in a later section.

Blocks

An ALGOL block structure is used. Blocks may be nested to any depth. They may be entered and left only via **begin** and **end**. The scope of names (except for labels) is the same as in ALGOL. A program is a block with **end** followed by **of program**.

Labels and jumps

Labels are of two kinds N: and $A(N)$:

The former are simple numerical labels; the latter are switch labels. Jump instructions take the form $\rightarrow N$ and $\rightarrow A(I)$ where I denotes an "integer" expression (see later). A label in a block may only be referred to by an instruction in the block and not in a sub-block. Checks are made for labels not set, and labels not referred to are commented on. In the case of $\rightarrow A(I)$, the value of I is checked at run time to ensure that it corresponds to an actual label $A(N)$. [This requires 4 instructions over and above the evaluation of I and the jump itself.]

Expressions

Expressions employ the operators + − * / ↑ ( ) in the usual way. The * sign may be omitted if no ambiguity results (i.e., implicit multiplication). A superscript ² can be used for 2. Brackets ( ) are also used to embrace the arguments of functions and arrays. For example

given **real** y, y' ; **array** a (1 : 10) ; **integer** i

$$2y * y' + log(y^2) + i(i-1) + a(i-1)/3 \cdot 14_{10} - 2$$

Assignments

Assignments take the form

$$v = E$$

where v denotes any variable (location) and E an expression. If v is an **integer** variable, then E is rounded off to

* *Department of Computer Science, The University, Manchester*, 13.

the nearest integer before the assignment takes place. An expression treated in this way is denoted by *I*. In denoting substitution of an expression in a statement we use *E* or *I* to indicate the implied assignment.

Standard functions

Standard functions are similar to those of ALGOL except that *ln*, *abs* and *entier* in ALGOL are written *log*, *mod* and *intpt* respectively in AA. In addition there are the following:

parity (I)	$(-1)^I$
fracpt (E)	fractional part of E
int (I)	*intpt* $(I + \frac{1}{2})$, i.e., nearest integer
radius (E_1, E_2)	*sqrt* $(E_1^2 + E_2^2)$
tan (E_1)	*tangent*

Finally *arctan* in ALGOL becomes *arctan* (E_1, E_2)

$$arctan(E_2/E_1)\begin{cases}\text{result in } \left(\frac{-\pi}{2}, \frac{\pi}{2}\right) \text{ if } E_1 > 0\\ \text{result in } \left(\frac{\pi}{2}, \frac{-\pi}{2}\right) \text{ if } E_1 < 0\end{cases}$$

Conditional instructions

Conditional instructions take the form

$$\left\{\begin{matrix}\textbf{if}\\ \textbf{unless}\end{matrix}\right\} \{\text{CONDITION}\}\ \textbf{then}\ \{\text{UNCONDITIONAL INSTR}\}$$

The CONDITION can be a compound condition, e.g.,

$$(x > 1 \textbf{ and } y = 1) \textbf{ or } n > 20$$

an elementary condition being of the form

$$E_1 \left\{\begin{matrix}=\\ \neq\\ >\\ \geqslant\\ <\\ \leqslant\end{matrix}\right\} E_2$$

The UNCONDITIONAL INSTRS include:

Assignment
Routine call & exit
Jump (including **stop**)
caption

Cycles

Only one form of cycle is permitted, namely

cycle $i = I_1, I_2, I_3$
[LIST OF STATEMENTS]
repeat

Here i denotes any integer variable and I_1, I_2, I_3 are all expressions of the form I, such that $(I_3 - I_1)/I_2$ is an integer $\geqslant 0$. The Is are evaluated and this condition checked as a prelude to the cycle. The Is remain unaltered throughout the cycle. Cycles may be nested to any depth.

Routines and functions

As in ALGOL a routine is essentially a named block with parameters. In AA, however, a routine heading of the form

$$\left\{\begin{matrix}\text{ROUTINE}\\ \text{TYPE}\end{matrix}\right\} \{\text{NAME}\}\ (\{\text{FORMAL PARAMETER LIST}\})$$

replaces the first **begin**

There are three types of routines (RT):

routine **real fn** **integer fn**

the corresponding exit instructions are

return **result** $= E$ **result** $= I$

As with data a routine name should be declared before it is referred to. This means either placing the routine near the head of the block in question, or alternatively giving an advance "specification" in the form

{RT} **spec** {NAME} ({FORMAL PARAMETER LIST})

and putting the routine itself near the end of the block. It is useful when laying out a large program to give, near the beginning of it, a summary of the relevant sub-routines in this way.

The call statement for a routine is

{NAME} ({ACTUAL PARAMETER LIST})

The relations between the FORMAL and ACTUAL PARAMETERS are as follows

FORMAL PARAMETER (FP)	ACTUAL PARAMETER (AP)
integer name i **real name** x **array name** a **integer array name** k	name of a variable of given type
integer i	I
real x	E
routine R **real fn** R **integer fn** R	name of a routine of given type

The names associated with the FPs (which can of course be arbitrary) have the force of declarations inside the routine, but when the FP is a routine (or function) a **spec** must be inserted (from which the RT can be omitted). An **integer** or **real** FP will be assigned the value of the AP at the time of the call ("call by value"). A . . . **name** FP will be assigned the actual store location of the AP; if the latter is an array element its subscripts are evaluated at the time of call ("call by simple name"). In a list of FPs of the same type, the type delimiters after the first may be omitted, e.g.,

routine spec *print* (**real** x, **integer** m, **integer** n)

may be written

routine spec *print* (**real** x, **integer** m, n)

In what follows we shall abbreviate such a specification in the form:

print (E, I_1, I_2)

PERM

A small library of routines is preloaded at the head of the block which surrounds the user's block, which is therefore entirely within their scope. These routines are used for fault monitoring, input/output (see next section), matrix algebra, integrating differential equations, simple list processing, and other operations. For obvious reasons the bulk of these routines are written in machine code. Although these PERManent routines are not, strictly speaking, part of the language, to the user they appear so.

Input and output

Input and output functions are mostly performed by PERM routines, special statements being avoided as far as possible: so far there are only two, namely "read" and **caption**. The permanent routines which we have found most useful are:

select input (*I*) *select output* (*I*)	*I* refers to an input or output "tape" in the Job Description
read symbol (*i*) *print symbol* (*I*)	reads and prints symbols converting to and from numerical equivalents (see later)
print (E, I_1, I_2)	prints the value of E with I_1, I_2 digits before and after the decimal point
print fl (*E*, *I*)	prints the value of E in floating decimal style to I significant figures
read (*v*—list)	(a special instruction) reads nos. from data tape, assigns them to listed locations
read binary (*i*) *punch binary* (*I*)	reads/punches 5, 7, or 12 bits from/on 5, 7 hole tape, or cards. The bits are handled as an integer.

Other routines serve to control the layout of the page:

tab	advances the printing position according to a preassigned table.
newline, newlines (*I*) *space, spaces* (*I*)	
runout (*I*)	punches *I* blanks on a Teletype punch (has no effect on the line printer)
newpage	gives "top of form" on a line printer, 30 newlines on a Teletype punch
next symbol	a parameterless **integer fn** which gives the numerical equivalent of the next symbol on the data tape, but does not advance it
caption STRING	a special instruction which outputs the characters in the string, terminated by ;

A symbol in quotes e.g., '=' is a permissible form for a constant, the value being the numerical equivalent of the symbol, which is always an integer. Precise knowledge of the constant is seldom needed, but can, if necessary, be calculated from a table giving the numerical equivalents of the basic characters. These are integers in the range 0–127. A symbol, or compound character, can consist of up to three superimposed basic characters. The numerical equivalent is $128y + z$ or $128^2x + 128y + z$, where x, y, z are the equivalents of its constituents and $x > y > z$.

In captions and between quotes we use the special symbols

ß or $ to denote space	these symbols being ignored on reading in the program text
ß or $ to denote underlined space	
; or ¦ to denote semicolon	because a ; or newline is used to terminate a **caption** STRING
¶ or ¶ to denote newline	

We must of course know the actual numerical equivalents of these symbols if we wish to use them (a rather unlikely event) and for this reason they are listed in the manual. For example, to print a $ symbol we write: *print symbol* (14807).

Punching conventions

To facilitate punching we have arranged that statements can be terminated by a newline or a semicolon. As a consequence if a statement occupies more than one line, all except the last one are terminated by ***c***. All spaces, underlined spaces, and superfluous terminal symbols are ignored. The special symbols ½ and ² are converted into ·5 and ↑2 on input. (Regardless of how it is punched a special case is made of ↑2 in order to compile efficient code.)

Comments may be inserted by means of

comment STRING or | STRING

The second alternative was introduced to economize on punching. In particular, underlining on Atlas Flexowriters entails backspacing, so that a delimiter word of n letters involves $3n$ tape characters. For this reason we have introduced a compiling mode which allows the user to write his delimiter words in upper case letters, provided all other names use only lower case letters, e.g.,

REAL *a*, *b*, *c* ; INTEGER ARRAY *u*, *v* (1 : 10)

Both forms of delimiters are in fact permitted in this mode which is operative between the statements **upper case delimiters** and **normal delimiters.**

Fault diagnosis

There are several facilities for helping to locate faults in the object program. The first to be implemented were label and routine tracing, and query printing. The former is a facility for recording the main *breaks* in the passage of control, and indicates either the sequence of routines entered, and/or the jump instructions

executed. Query printing allows the user to terminate an assignment statement with a ?, thus

$$x = 1 + 2y \ ?$$

The program (or selected sections of it) can then be compiled in one of two modes

(i) **ignore queries** in which the ?'s are ignored,
or (ii) **compile queries** in which extra instructions are compiled for printing out the assigned value.

These two phrases are the statements actually used to delimit the area of the program in which the facility is operative. A similar device is used in connection with label and routine tracing. These aids to fault diagnosis require some action on the part of the user and of necessity involve a separate run of the program. Perhaps the most useful debugging aid is the stack post mortem which is involved whenever an unforeseen fault arises.

Faults arising in the object program are classed as trappable or untrappable, and they are detected in the first instance either by the supervisor or the object program. Untrappable faults are those regarded as being necessarily catastrophic, e.g., an input stream, or "tape", not defined in the Job Description (see Howarth *et al.*, 1961), while trappable faults are those not so regarded, there being some possibility of retrieving matters by jumping to a preassigned part of the program. An example of the latter situation is an arithmetical fault (e.g., division overflow) occurring in a series of independent calculations. In this type of program if a particular "case" breaks down one can simply pass on to the next.

Faults detected by the supervisor are as follows:

trappable	*untrappable*
div overflow	computing time exceeded
exp overflow	output allowance exceeded
sqrt argument < 0	input/output streams not defined (these refer to declarations in the Job Description)
log argument $\leqslant 0$	illegal instruction (obeying a word which is not an instruction)
inverse trig fn out of range (the above refer to arithmetic faults arising in basic or extracode instrs.)	illegal operand (reference to a private part of the store)
no more data in input stream	
more store required	
local time exceeded	

The following faults are detected by tests compiled in the object program.

trappable	*untrappable*
spurious character in data	switch label not set (e.g., $\rightarrow A(i)$ where $i = 3$ and $A(3)$ is not set)
non-integral data assigned to integer location	array dimensions < 0 (e.g., **array** $A(m: n)$, where $n - m < 0$)
other data faults	non-integral cycle count (e.g., **cycle** $i = 1, 2, p$, where p is even)
	array subscript out of range
	incompatible matrix dimensions (refers to some matrix arithmetic routines in PERM which check that the operands are compatible.)
	call for non-existent routine (could in theory be detected at compile time, but is easier to handle at run-time)

Note: "array subscript out of range" is only detected as a fault on request. This is done by delimiting the areas of interest in the program by the statements **compile array bound check** and **stop array bound check**. Outside these areas faults of this kind will not be detected and may in fact cause extravagant effects. The reason for making this facility optional is that the extra instructions required in the program may substantially increase both the size and running time of the program, the latter by up to 100%. It would be essential that checks of this kind should be built into the hardware of any future machines.

When a trappable fault occurs the supervisor, or the object program, refers to a "trapping vector" to find the address to which to transfer control. This trapping vector contains one entry for each type of fault (the faults being numbered 1, 2, 3, . . .) and is set up by the user, or rather by the compiler from information supplied by the user. This takes the form of instructions, for example

fault $1, 5 \rightarrow 3, 2, 4 \rightarrow 1$

This would cause a jump to label 3 should a (trappable) fault of the type 1 or 5 subsequently turn up, or a jump to 1 in the case of a fault of type 2 or 4. The labels must be local to the block in which the above **fault** statement occurs, usually the outer block. In the event of a fault the stack is cut back to its extent at the time the fault statement was obeyed, but some variables may have been altered in the meantime and allowance must be made for this in planning the rescue operation.

If a trappable fault is not trapped the program terminates with the standard monitoring, as in the case of an untrappable fault. This monitoring consists, firstly, of printing out the line number of the faulty statement and the name of the routine (or serial no. of the block: see program "map") in which it occurred. Following this it prints out a summary of the working space (scalars and cycle counts) in the last block entered at each level as far as the current (faulty) block. To identify the monitored variables the table of names is

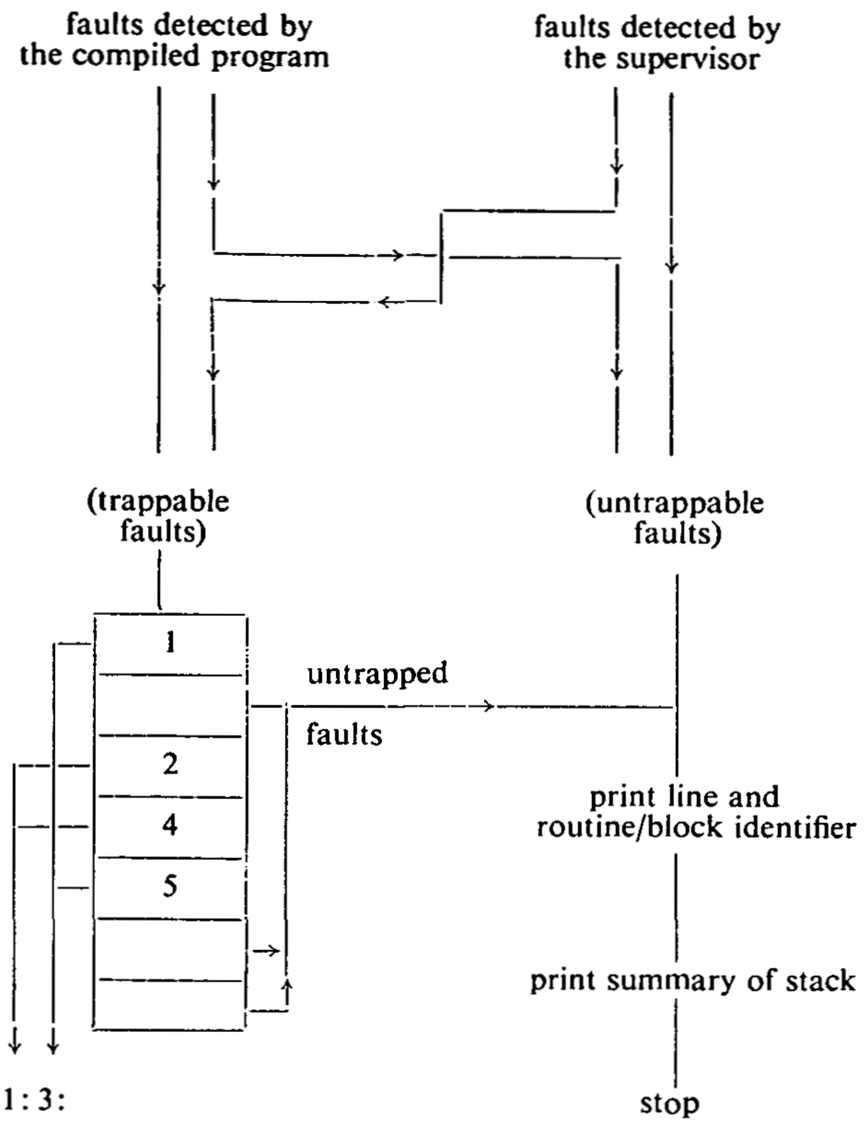

Fig. 1.

retained at run-time. An example of the monitoring for a specimen program is given in the appendix. **Fig. 1** illustrates the fault handling system using the trapping vector set up for **fault** 1, 5 → 3, 2, 4 → 1.

Efficiency of the object program

As in most ALGOL translators the object program uses a stack to store data and for working space (see e.g., Watts, 1963). An index register is associated with each textual level and points to the section of the stack opened up by the activation of the current block at that level. The previous contents of the index register (and the link if the block is a routine) are preserved in the stack immediately on entry to the block, and are subsequently restored on leaving it. When a routine is handed on as a parameter of another routine it is also necessary to hand on a picture of the relevant index registers as they were at that point. When the parametric routine is eventually called in, it is necessary to reset temporarily the index registers to their original state. This latter operation is performed by three central banks of instructions which preserve, reset and restore up to 9 index registers according to the point of entry, this being chosen so that only the relevant index registers are interchanged.

For each block the stack is divided into a static and dynamic part. Scalars, array boxes, cycle parameters, whose size is known at compile time are allocated to the static part where they can be accessed by a single (modified) instruction. **reals** are stored as floating point numbers, **integers** as destandardized numbers which allows them to be used either as address additives, or as operands in an arithmetical expression. All these quantities are initialized to zero on entry to the block, and the stack pointer advanced to the end of this part of the stack, the extent of which is known at compile time. Dynamic items, i.e., arrays, are then allocated storage from this point onwards. An array box consists of two address words, one of which points to the dope vector (which may be shared by several arrays) and the other points either to the array itself (if it is a vector) or to a hierarchy of address words which point to the rows, planes, etc., of the array (a scheme suggested by Iliffe, 1961). In this way any particular array element can be reached by a simple succession of store references (one for each dimension), without any multiplications. This, together with the fact that **integers** are stored in address units, means that array elements such as $a(i + 1, j + 2)$ can be "fetched" in 5 instructions. On the other hand, it is difficult to improve the system without resort to optimization procedures which would equally well benefit a simple scheme not involving the "Iliffe" vectors. To give the reader some idea of the efficiency (or lack of it) of our object program, we give below the translation of the following cycle.

sum = 0
cycle *i* = *p*, *q*, *r*
sum = *sum* + *a*(*i*) *b*(*i*)
repeat

prelude to evaluate $p, r + q, r$ and check that $(r - p)/q$ is an integer $\geqslant 0$

```
334    0    (p)        enter cycle with
121  127    —          initial value of i
330    0    (r + q)    form i + q = (i − r)
                           + (r + q)
356    0    (i)        restore i
101   97    (i)        form address of a(i)
104   97    (a)        in index register 97
324    0    97   0     fetch a(i)
101   97    (i)        form address of b(i)
104   97    (b)        in index register 97
362    0    97   0     multiply a(i) b(i)
320    0    (sum)      add sum
356    0    (sum)      restore sum
334    0    (i)        form difference
331    0    (r)        i − r in acc
235  127    —          jump if acc ≠ 0
```

The use of machine code

Facilities exist for using machine instructions in AA programs, although needless to say, this is not encouraged. In some cases, however, it is useful

(*a*) to speed up a critical inner loop
or (*b*) to perform some function not available through formal Autocode statements.

Examples of the latter are the use of local timers, and the use of peripheral equipment in certain special modes. To illustrate machine code we give in **Fig. 2** the optimum form of the scalar product loop used in the previous section.

It will be seen that this loop takes 6 instructions compared with the 13 of the formal translation. [Note: the above loop can be improved still further by moving the "124" instruction to follow the "362" so that it will be completely overlapped. This will necessitate other minor changes, however, and we shall not give details.]

```
prelude to compute addresses of a(r) and b(r) and place them in the index registers 97 and 98, and evaluate p and q

        121 ,  99 , — , p      ; | form p — r
        122 ,  99 , — , r      ; | in IR 99
        121 , 127 ,  0 , 1:
3:      124 ,  99 , — , q      ; | increment by q
1:      324 ,  97 , 99 , 0     ; | fetch a(i)
        362 ,  98 , 99 , 0     ; | form a(i) b(i)
        320 ,  0 ,  — , sum    ; | and add
        356 ,  0 ,  — , sum    ; | to sum
        215 , 127 , 99 , 3:    ; | test for last cycle

Set  i = r
```

Fig. 2.

Speed of translation

Having revealed something of the quality of the object code it is appropriate to say something about the speed of compiling it. Several improvements have been made to the original AA compiler. The figures in **Table 1** refer to one of the latest versions.

The following table gives a breakdown of the compiling time for 4 programs. The figures are based on instruction counts provided by the Atlas Supervisor. As may be expected there is a good deal of variation from one source program to another.

Notes on Table 1

1. This is simply the time spent in reading in the program. More precisely it is the time charged to the compiler by the Supervisor which supplies the characters in 6-bit internal code, with its own "shift" system. It is approximately proportional to the length of the program tape.
2. This is the time spent in converting to 7-bit code, and then to reconstructing the image of the symbols on the line. Each symbol may be formed by superimposing up to 3 distinct basic characters (a compound character). It also includes the removal of spaces, underlined spaces, and erases.
3. In this stage all the names, constants, and delimiter words are converted into 48-bit words, and categorized to simplify the decision processes of the next stage. Stages 2 and 3 account for well over half the total compiling time.

Table 1

input of program (see Note 1)	22·3	17·4	12·4	19·8	% of total compiling time
line image construction (see Note 2)	45·0	55·7	42·0	44·8	
elementary syntactical analysis (see Note 3)	19·9	14·4	29·8	17·2	
further analysis and compilation (see Note 4)	12·8	12·5	15·8	18·2	
instrs. obeyed in compiler / 1024	1106	1934	3452	1628	
size of object program (instrs.)	3528	3704	1673	3980	
total number of statements	406	664	936 (approx.)	513	
length of program tape (including blank tape) in feet	116	214	?	176	

4. This is the translation stage proper. The instruction is analyzed and either converted to object code, or used in some way to control the compiling process.

Stages 1, 2, and 3 are applied a (complete) line at a time; stage 4 a statement at a time.

Program no. 1	was a test program using short identifiers.
Program no. 2	used largely English words as identifiers.
Program no. 3	consisted mainly of machine instructions, each of which occupy 10–20 characters.
Program no. 4	was a typical applied mathematical program.

Except in the case of programs like no. 3, between 300–400 instructions are executed in the compiler for every instruction compiled.

Recent developments

We can deal with these only briefly here, a full account will be published elsewhere.

Further arithmetic types. Work in this area has taken two directions.

1. Introduction of a type **complex** which like **real** or **integer** can be used in expressions of the form already described, e.g., **complex** z, **real** u, **integer** k ; $z = u + 2\pi k\boldsymbol{i}$.
2. Introduction of an entirely new form of expression which is at the same time a statement, e.g.,

   [A α β [A β [A α β A] α] β A]

   Here the A's denote operands, and the α's and β's are (right) unary and binary operators, respectively. All operators have the same precedence and, except for [], the expression is processed from left to right. Included in the β's are assignment $=>$, and jump $\rightarrow$. The operands used in such expressions represent a new family of composite types, these being numeric (single, double, and multi-precision, real and complex) and non-numeric (logical and label). (*See* Brooker, 1964.)

Structures. Some facilities for defining, constructing, and analyzing tree structures (*not* list processing) have been included in order to give AA some capability in the area covered by the compiler-compiler.

Appendix

Example of a faulty program and associated monitoring.

```
begin
real a, b, c,alpha,a1,a''
integer i, j, k,theta
array x(1:10)
cycle k = 1, 1, 10
x(k) = k
repeat
k = 0
routine spec test 1(real d, integer m)
routine spec test 2 (real name e, integer name n,real fn test3)
real fn spec  test 3(array name y)
a = 1 ; alpha = 7.663 ; a1 = 10.4 ; a'' = 21.6
j = 2 ; theta = 123
test 1 (2, 3)
routine test 1 (real d, integer m)
real f
integer o
f = 1.5; o = 3
test 2(f, o,test3)
end
routine test 2 (real name o, integer name n ,realfn test3)
spec test 3(array name y)
real g
integer p
g = 6.7; p = 4
g=test 3(x)
end
real fn test 3 (array namey)
real h
integer g
h = 6.43
cyclo g = 1, 1, 10
h=sqrt(-2)
repeat
cyclo g=1,1,3
h=h+g
repeat
result =h
end
end of program
***Z
```

```
COMPILER AB/3
        1  BEGIN BLOCK NO = 91  ADDRESS = C0115050
       15    BEGIN ROUTINE<TEST1> NO = 92  ADDRESS = 00116272
       20    END ROUTINE<TEST1>  OCCUPIES 37 LOCATIONS
       21    BEGIN ROUTINE<TEST2> NO = 93  ADDRESS = 00116750
       27    END ROUTINE<TEST2> OCCUPIES 42 LOCATIONS
       28    BEGIN REAL FN<TEST3> NO = 94  ADDRESS = 00117500
       39    END REAL FN<TEST3> OCCUPIES 79 LOCATIONS
       40  END BLOCK OCCUPIES 244 LOCATIONS
PROGRAM ENTERED



A RUN TIME FAULT HAS OCCURRED AT


LINE 33 REAL FN <TEST3>
SQRT -VE


THE FOLLOWING BLOCKS AND/OR ROUTINES WERE EXECUTED

BLOCK 91
A= 1.000010  0    ALPHA= 7.663010  0    A1= 1.040010  1
A''= 2.160010  1    J= 2    THETA= 123
CYCLE <K> EXECUTED 10 TIMES
THE PROGRAM NEXT CALLS IN AT LINE 14

ROUTINE <TEST1>
F= 1.500010  0    O= 3
THE PROGRAM NEXT CALLS IN AT LINE 19

ROUTINE <TEST2>
G= 6.700010  0    P= 4
THE PROGRAM NEXT CALLS IN AT LINE 26

REAL FN <TEST3>
H= 6.430010  0    G= 1
CYCLE <G> EXECUTED 0 TIMES
CYCLE <G> NOT ENTERED
THIS IS THE REAL FN IN WHICH THE FAULT OCCURRED
```

Acknowledgements

Some material from the Atlas Autocode "Mini-Manual" (Lunnon and Riding, 1965) has been adapted for the first part of this paper, and we would like to thank the authors of that document for simplifying our task in this way.

References

BROOKER, R. A., and ROHL, J. S. (1965). *The Atlas Autocode Reference Manual*, Manchester University.
LUNNON, W. F., and Riding, G. (1965). *The Atlas Autocode Mini-Manual*, Manchester University.
HOWARTH, D. J., PAYNE, R. B., and SUMNER, F. H. (1961). "The Manchester University Atlas Operating System Part II: User's Description," *The Computer Journal*, Vol. 4, p. 226.
ILIFFE, J. K., and JODEIT, J. G. (1962). "A dynamic Storage Allocation Scheme," *The Computer Journal*, Vol. 5, p. 200.
WATT, J. M. (1963). "The realization of ALGOL procedures and designational expressions," *The Computer Journal*, Vol. 5, p. 332.
NAUR, *et al.* (1963). "Revised report on the algorithmic language ALGOL 60," *The Computer Journal*, Vol. 5, p. 349.
BROOKER, R. A. (1964). "A programming package for generalised arithmetic," *Comm. A.C.M.*, Vol. 7, p. 119.

Correspondence: Commercial English languages

To the Editor,
The Computer Journal.

Sir,

I should like to comment on the very informative article by R. M. Paine in the October 1965 issue of *The Computer Journal*, entitled "The Gradual acceptance of a variety of commercial English languages." Under the heading 'Compact COBOL' the second paragraph describing the method of translation is a little misleading. The reader might well get the impression that COBOL and other languages on the I.C.T. 1900 are translated first into PLAN. This is not true. The form of output from the COBOL compiler is the same as that from the PLAN compiler and from the compilers for the other languages Rapidwrite, FORTRAN, E.M.A. and ALGOL. This is also the form in which many of the library subroutines are stored. This common form is an acceptable input to a Consolidation routine and it is this which lends itself to the automatic inclusion of subroutines and the easy amalgamation of sections of program written in different source languages.

Yours sincerely,
E. HUMBY

International Computers and Tabulators Limited,
Bridge House, Putney Bridge, London, S.W.6.
27 October, 1965.

To the Editor,
The Computer Journal.

Sir,

It is surprising that Mr. Paine's paper on commercial English Languages (1) should have been delivered to an international conference in New York as a balanced statement of affairs in the U.K.

It is typical of this article that it nowhere mentions Language H by name, although this is a British language that has already been implemented on three widely different NCR/Elliott computers (405, 315 & 803) and is being developed on another (4100). The section of the article on ACL is very misleading and I will return to it later in this letter.

Use of the phrase "anti-COBOL, anti-standardization" is a way of "proving" guilty by association. Presented with an adequate COBOL compiler for a specific machine, I will gladly use it as a normal tool of the trade; but if you insist on raising it up as an idol of standardization, then I may point out its very obvious feet of clay. My Biblical metaphor is perhaps provoked by that other phrase "tower of Babel"; the actual plot of this story has no relevance whatever to our problems, and the phrase seems to serve as a shorthand way of sneering at people with the ability and energy to create new computer languages.

Where we need standardization is in our thinking and talking abour computer languages—notation, standard meta-language, definition of concepts (2), analysis of structual features etc. There was an opportunity about five years ago for standardization in the U.K., when a meeting of all manufacturers was held. Unfortunately, one representative at that time prevented the meeting from getting down to hard thought and work, and persisted in wanting merely to follow the latest transatlantic fashion (which happened to be COBOL). Another lost opportunity was the European meeting at Amsterdam a few years ago, to consider adaptation of ALGOL for business data processing.

Mr. Paine's article has the usual references to scores of man-years being spent on compiler writing. One is tempted to generalize, unfairly, that the more the man-years the worse the result. You do not need a large team to write a compiler; you need a small group of intelligent people. There are hundreds of such competent people in this country, but the frustration has been that management of British and ostensibly British computer manufacturers and the large users have seldom had the guts to invest in the ability of their technical staff.

Where the man-years are needed is in the "infra-structure" of compilers, in the operating systems and file conventions etc. Most of all they are needed in the publication of manuals and in continually updating them.

To return to ACL (Atlas Commercial Language), there are a number of inaccuracies in the article:

(1) ACL was not developed by the Institute of Computer Science, but by University of London Atlas Computer Service, who operate the computer.
(2) At the date of the article, May 1965, there were already several working ACL programs.
(3) The remark about "Print $(A + B - C)$" is obscure. This feature is not in fact part of ACL, but in any case it is a routine facility of computer languages (3). The point of ACL is that a particular "shape" of printed line is defined by column positions across the page, using literals and data names (either new ones for printing, or already defined somewhere else), together with the pernickety editing details that are so important in business reporting.