An Atlas Autocode to ALGOL 60 translator

By A. T. McEwan*

An Atlas Autocode to ALGOL 60 program translator has been used to translate a large number of programs developed by the users of the Cripps Computing Centre. The translator was written in Atlas Autocode in a machine independent subset, and successfully translated itself. It developed through four main phases in a modular form using earlier working versions designed to achieve partial translation.

1. Introduction

The decision to write the translator was taken in March 1965 when Nottingham University ordered a KDF9 computer. Most user programs were written in Atlas Autocode, and at that time no Atlas Autocode compiler was available for KDF9. In fact, an excellent KDF9 compiler for the language has now been provided by the Edinburgh University Computer Unit. Nevertheless, the translator has still proved a sound investment, as users may translate their Atlas Autocode programs or subroutines for subsequent use as, or in, ALGOL programs, and take advantage of the KDF9 Post facilities offered for program text editing. It was necessary that some form of working translator should be made available by February 1966, the delivery date of the KDF9.

The Flowers Report (Council of Scientific Policy (1966), p. 43, para. 159) alludes to the need for such a translator to be written:

"There is one special difficulty that arises in Universities which have made substantial use of Atlas Autocode on the Manchester, London and Chilton machines and which may therefore have large numbers of programs already written in this language. It is evident that none of the new series are to be provided with compilers for this language and it will be necessary for those Universities wishing to continue its use to provide a compiler on their own or in collaboration with other similarly situated institutions. The Working Group sought out the views of the computer manufacturers on the provision of compilers for institutions whose programs are in Atlas Autocode. It appeared that if Universities wished to continue to use this language they should consider producing their own compiler."

At first glance, the programming languages Atlas Autocode and ALGOL appear very similar, but a closer inspection reveals some fundamental differences. The most important is that the Atlas Autocode language has dummy subroutine headings or declarations, in addition to the actual subroutine headings, which enable Atlas Autocode to be compiled statement by statement since all names are declared in advance. Other differences will be indicated below.

When the definite need for such a translator had been established, the problem of implementation arose. Apart from the Director, the Computing Centre had at that time only one member of academic staff—the author. He was the sole programmer available for such a project, but he also had many other commitments concerning the Data Link service to Manchester, and it could be seen that these duties would increase rather than decrease as the installation date for the KDF9 approached.

Macro-generation techniques offered a practical approach for part of the problem. For example, the Compiler Compiler (Brooker, MacCallum, Morris and Rohl, 1962) seems the best solution, especially as "phrase" and "format" definitions were also being implemented (and are now available) in the Manchester Atlas Autocode Compiler. Unfortunately, Compiler Compiler programs do not run on KDF9, and at that time it was desirable, if not essential, that a working version of the translator be implemented for KDF9. There was in fact no macro-generator common to both Atlas and KDF9, and the writing and debugging of such a program was considered too risky since its abandonment would have left nothing to help in translations. For similar reasons the idea of designing an intermediate machine-independent language, into which Atlas Autocode would be translated before translation into ALGOL, was not followed up, although it was borne in mind when writing the translator. Since it was unlikely that a series of translators for different programming languages would be written, this idea was also not very practicable.

A crude approach had to be adopted, and so the translator program was bootstrapped in a series of complete sub-sets of the final translator specification, with each new program making use of the previous one. Had the project been suffocated, there would at least have been a program available to do some of the dreary task of translation.

2. Objective

The obvious objective is the translation into the ALGOL 60 language of any program written in Atlas Autocode in such a way that the ALGOL program can be compiled without any further alteration. This could not be achieved because of several fundamental differences:

* *Cripps Computing Centre, University of Nottingham.*

One facility not available in ALGOL is a counterpart to the store mapping routines available in Atlas Autocode. These routines essentially compute the address of a store location. For example:

real map x (**integer** i, j)
result $= addr(a(i)) + \frac{1}{2}i*(i-1)+j-1$
end

calculates the address of the (i, j)th element of a real lower triangular matrix stored by rows starting with $x(1, 1)$ at location where the vector element $a(1)$ is stored. The crucial point is in the use of such functions, as they may appear on the left- or right-hand side of an assignment statement. For example:

$$x(i, j) = x(i, j)^2$$

It would be easy enough to implement store mapping functions in ALGOL if they were not used as assignment variables, but a procedure call in the ALGOL 60 language is not permissible as an assignment variable.

Another difference is the use by Atlas Autocode compilers of "dope vectors" in conjunction with each array declared. These vectors contain information about all the bounds and the dimension of each array. The two standard Atlas Autocode subroutines "*dim*" and "*bound*" give the dimension and bounds of any specified array, and consequently matrix subroutines need only pass down array names. The standard matrix operations routines available in Atlas Autocode include, for example:

matrix add (**arrayname** A, B, C)
matrix sub (**arrayname** A, B, C)
eqn solve (**arrayname** A, b, **realname** *det*)

The first two perform the matrix operations

$$A = B + C$$
$$A = B - C$$

The "*eqn solve*" routine solves the set of linear first order equations $Ax = b$ and leaves the value of the determinant of the matrix A in the variable specified in the routine call; the solution vector over-writes the vector b.

3. Specification

Apart from the differences mentioned above, the translator converts programs written in the Atlas Autocode language as specified in the Atlas Autocode Mini-Manual (see Lunnon and Riding, 1965) into the official ALGOL 60 language; i.e. the facilities currently available in the AB compiler. Atlas ALGOL type input/output procedure headings, which are conveniently similar to those of Atlas Autocode, are implemented. As far as possible the translator is machine-independent.

Since symbolic character representation is available in the Atlas Autocode language, it was felt that programs containing the machine dependent numerical representation of characters need not be guaranteed to work correctly after translation.

4. Implementation

Having determined the general strategy of implementation, a basic decision had to be made concerning the use of a programming language for the translator itself. ALGOL, being available on most computers seems the obvious choice (thereby making the translator available to any computer with an ALGOL compiler, and consequently the KDF9), but Atlas Autocode is also a candidate since the translator itself could be translated into ALGOL, providing working versions in both languages. Atlas Autocode was chosen. As mentioned above, the Atlas Autocode language contains symbolic character representation which enables the translator to be independent of the machine's internal character code. Another useful facility known as "define special compiler" enables part of a program to be compiled and written onto magnetic tape, so that it can be read down later and completed. Another by no means trivial consideration was that the author had to punch his own tapes, and time could be saved by using upper case delimiters (e.g. **begin** can be typed BEGIN), a facility then available in Atlas Autocode but not in Atlas ALGOL. It is now implemented for ALGOL (Atlas ALGOL Paper No. 10, 1966).

The translator developed through four phases. Phase 1 provides text editing only, and takes no account of the context in which symbols occur. Phase 2 contains the backbone of the translator. It contains a large number of routines which were extended as more facilities were translated, and takes account of the context in which symbols appear. Phase 3 makes a distinction between the types of variables (**integer, real, array, switch, routine,** etc.) and their properties. Phase 4 rearranges the translated routines (now procedures) into their correct position, with other declarations at the head of the blocks in which they appear. During this phase the translated program is reorganized, and a line-reconstruction routine is added to read Atlas coded paper tape.

Originally it was intended to have another phase to provide further syntax checking. This proved to be unnecessary, since the program is a translator and not a compiler. It seems reasonable to assume that only working programs or subroutines will be submitted, but a syntax checker to remove un-translatable facilities from the incoming text might be useful, so the translator was written to allow for this possible addition in ALGOL or Atlas Autocode.

Phase 1

During this phase the input of program text is covered by three subroutines. The first

read ch (**integer name** i)

reads the next symbol from the reconstructed line of data (program text). Spaces and underlined spaces are ignored. Upper case letters are converted into underlined lower case letters if the appropriate flag is set. It also ignores **c** when followed by a newline.

The second subroutine

read character (**integer name** *t*)

ignores or assigns to delimiter words a negative numerical character value. Assuming that only valid delimiter words are being tested, an initial switch is made on the first letter of the delimiter word, followed by a tree sort until the delimiter word is uniquely determined. This usually involves looking at the first two or three letters only. A final check is made by testing that the last letter of the delimiter word is correct; for example, only the first two letters and the **y** of the delimiter word **array** are examined. The following delimiter words are ignored:

compile array bound check;
production run;
upper case delimiters;
and **normal delimiters;**

except that the last two set or unset a flag denoting whether upper case delimiters or normal delimiters are currently being used.

The third subroutine

read instruction 1

reads into a buffer a pseudo-statement. A pseudo-statement consists of either a single delimiter word or an expression (assignment or conditional) terminated by a newline, a semi-colon, a colon, or a delimiter word. This subroutine also converts the symbols π, ^{2} and $\frac{1}{2}$ into 3·14159265359, ↑2, and ·5 respectively. If the first character of the pseudo-statement is a | then it is replaced by the character value −14 which corresponds to the delimiter word **comment.** This subroutine also inserts extra multiplication signs where they are not explicitly given, and replaces symbolic character representations by their internal numeric values.

For example, these three routines from the input text:

upper case delimiters
delta = $2a*b^2(sin(\frac{1}{2}a\pi)+b^2)$ *IF z* = '*a*' *OR* c
  (*x* = '*b*' *AND y* = 3)
normal delimiters; begin
C = *D*+*a*; | *example statements*
stop if *C* = 0

produce the pseudo-statements

(i) *delta* = 2**a***b* ↑ 2*(*sin*(·5**a**3·14159265359)
  +*b* ↑ 2) **if**
(ii) *z* = 96 **or**
(iii) (*x* = 97 **and**
(iv) *y* = 3) [*newline*]
(v) **begin**
(vi) *C* = *D*+*a*
(vii) **comment**
(viii) *example statements* [*newline*]
(ix) **stop**
(x) **if**
and (xi) *C* = 0 [*newline*]

where the delimiter words **if, or, and, begin, comment** and **stop** have character values −18, −21, −10, −12, −14, and −29 respectively. The [*newline*] shows that the character value for newline is also present in the buffer. Empty lines are ignored.

The development to this stage started with a simple read character subroutine "*read ch*". A "*read instruction*" subroutine and a new "*read character*" subroutine were subsequent additions.

The output subroutines had a similar history, starting as a simple output loop and a pseudo-print symbol routine called "*pprint symbol*".

The latter was extended to take account of symbols with negative values, which cause an entry to a "*pcaption*" subroutine, (pseudo-caption), which outputs the corresponding **caption.** For example the instruction "*pprint symbol* (−14)" gives rise to an entry to the "*pcaption*" subroutine; there a switch leads to the following line of program:

c(14): **caption comment**; **return**

All such delimiters are listed in this routine, so it is a simple matter to change the captions to accommodate the quirks of ALGOL compilers for other machines.

The phase 1 main program is a loop containing calls for the two routines "*read instruction*" and "*scan and output*". The latter was developed from a simple "pprint symbol" loop to replace the Atlas Autocode symbols → and ↑ by the delimiter words **goto** and **exp** respectively.

Phase 2

Atlas Autocode instructions either begin with a delimiter word, or contain no delimiter words, or are conditional instructions. This phase is based on a routine which recognizes the three types of statement. For those statements which commence with delimiters a switch is determined by the delimiter word's assigned value. At most of the switch labels there is an instruction which sets a flag to denote the type of statement (e.g. a declaration), followed by an entry to a routine. In phase 2 no account is taken of the variable types, and some of the routines are virtually copy text.

The setting of flags for statement types enables extra blocks in the object program to be opened when a declaration does not immediately follow another declaration or block heading. (Routine headings are classed as declarations, but in this case new blocks are not necessary.) For example:

begin
integer *n*, *i*, *j*
read (*n*)
array *a*(1: *n*, 1:10)
cycle *i* = 1, 1, *n*
cycle *j* = 1, 1, 10
$a(i,j) = \frac{1}{2}|x+y+|u+v|^2|$
repeat
repeat
end

becomes

> **begin**
> **integer** n, i, j;
> *read* (n);
> **begin**
> **array** $a(1: n, 1{:}10)$;
> **for** $i := 1$ **step** 1 **until** n **do begin**
> **for** $j := 1$ **step** 1 **until** 10 **do begin**
> $a(i, j) := \cdot 5 * abs(x+y+abs(u+v)$**exp**$2)$;
> **end**;
> **end**;
> **end**;
> *end*: **end**;

This example shows the expansion of modulus and **cycle** instructions. The translation of modulus signs is achieved by counting them and replacing each by the appropriate ALGOL equivalent. For example:

$$|x+y+2(3+||u+v|+6|)|$$

would undergo three stages in translation. The first has the commencing modulus sign changed to "*abs*(" (or rather the appropriate pcaption number), then the signs are counted to find the corresponding closing modulus, ignoring those following an operation sign (e.g. +, −); a count is also kept of any opening and closing round brackets, adding or subtracting 1 for each bracket respectively. This count, originally set to zero, has to be zero before a modulus sign can be changed. After the first scan through, the statement becomes:

$$abs(x+y+2*(3+||u+v|+6|))$$

it is then processed and the next modulus sign translated similarly to yield

$$abs(x+y+2*(3+abs(|u+v|+6)))$$

and again for the last pair to give

$$abs(x+y+2*(3+abs(abs(u+v)+6))) \quad \text{as required.}$$

The translation of the delimiter words **return** and **result** requires an unconditional jump to the end of the block. With this in mind the **end** of the block is marked with the label "*end*". For example:

> **return if** $a > b$;
> . . . .
> **end**

becomes

> **if** $a > b$ **then goto** *end*;
> . . . .
> *end*: **end**

Similarly for **stop**. The last **end** of the program (which corresponds to the Atlas Autocode **end of program**) is labelled "*end of program*: **end**". **result** cannot be translated during phase 2, as the name of the procedure in which it appears is not available.

Counts are initiated on entry to the routines dealing with the delimiter words **begin**, **routine** and **fn**. These counts keep track of all blocks opened and closed. When the **end** matching the original delimiter is recognized the extra **ends** to match additional inserted **begins** are output. During phase 2 the conditional statements are the most complicated to decipher. The reversed conditional statement surprisingly proved the easiest to translate. An example of such a statement is:

> $a = b$ **if** $(x > 0$ **and** $y < 1)$ **or** $z = 2$

where the condition or Boolean expression follows the delimiter word **if**. The pseudo-statement "$a = b$ **if**" is read into a buffer, and is examined to determine whether it is one of the following Atlas Autocode types of statement which are to be ignored:

> **routine trace on if** $a > b$
> **jump trace off if** $c < d$
> **queries on unless** $x \leqslant y$
> **array bound check off unless** $y \neq z$

If the conditional statement is not ignorable, then the condition is read into the buffer and analyzed, pseudo-statement by pseudo-statement. The alternative ALGOL-like form

> **if** $(x > 0$ **and** $y < 1)$ **or** $z = 2$ **then** $a = 6$

requires all the pseudo-statements in the conditional expression to be stored before the statement following the delimiter word **then** can be examined to determine whether it is one of the test facilities—**routine trace**, etc.—that are to be ignored.

Two flags are set during the evaluation of the conditional part to control the use of the "*scan and output*" routine; the first denotes a Boolean expression, and the second whether it is an **if** or **unless** type condition. If the pseudo-statement being analyzed is not part of a Boolean expression, then a ":" is automatically inserted before any "=" signs. If the Boolean expression is part of an **unless** statement, then it is inverted, for example:

> $a = b$ **if** $(x > 0$ **and** $y < 1)$ **or** $z = 2$

is translated into

> **if** $(x > 0$ **and** $y < 1)$ **or** $z = 2$ **then** $a := b$

but the statement

> $a = b$ **unless** $(x > 0$ **and** $y < 1)$ **or** $z = 2$

has to be inverted

> **if** $(x \leqslant 0$ **or** $y \geqslant 1)$ **and** $z \neq 2$ **then** $a := b$

As already mentioned, a pseudo-statement can end with a colon. If this happens for the first part of a statement, the statement must be complete and can be identified as an ordinary label or switch label. Atlas Autocode has integer labels; although these are permissible in the ALGOL 60 specification, they are not usually implemented, so they are converted into names by routine "scan and output", which prefixes and postfixes the letter "*l*". This avoids confusion with any names the Atlas Autocode programmer may use, as letters after numbers are not permitted in Atlas Autocode names. Primes are not allowed in ALGOL, so names with primes are prefixed with the letter *p* and the integer of the number of primes. For example:

integer x, x', x'', x''';

becomes

integer *x*, *p*1*x*, *p*2*x*, *p*3*x*;

In the phase 2 translator the output routines have become complicated.

Phase 3

Phase 3 was developed to overcome the only major problem remaining. This is, the conversion of variable declarations and routine headings. From this a distinction can be made between array and routine names, e.g. to determine whether round or square brackets are required in the ALGOL object program.

To make the conversion, a "*property list*" must be kept of all names used in the Atlas Autocode program. This is organized by the routines

set up pl
add to pl
and *look up pl.*

"*set up pl*" sets up a property list of all the standard subroutine names. These are divided into three groups; those which are too machine-independent to translate, those which the translator can modify or provide, and those which the user has to add, or change the call for, e.g. the matrix routines.

"*add to pl*" adds names to the property list, together with an indication of the type of name, e.g. **real** or **integer, array, switch, routines**, etc. The **switch** indication also shows—indirectly—the lower bound of the switch.

"*look up pl*" looks up the name in the property list to determine its type.

With the aid of these routines the example text

```
integer i, j, n
array x, y(1:100), z(1:2, 1:100)
routine spec triangle mult (array name A, B, c
  integer n integer name m)
read (n)
read array (x); read array (y)
triangle mult (x, y, n, j)
       cycle i = 1, 1, 100
       z(1, i) = y(i)²
       repeat
```

is translated into

```
integer i, j, n;
array x, y [1:100], z[1:2, 1:100];
n:= read;
read array (x); read array (y);
triangle mult (x, y, n, j);
for i := 1 step 1 until 100 do
begin
        z[1, i] := y[i] exp 2;
end;
```

The switch list declaration

switch *a*, *b* (2:6), *c* (−4:−3)

is translated into

switch *a*:= *a*1*l*, *a*2*l*, *a*3*l*, *a*4*l*, *a*5*l*;
switch *b*:= *b*1*l*, *b*2*l*, *b*3*l*, *b*4*l*, *b*5*l*;
switch *c*:= *c*1*l*, *c*2*l*;

while the switch jumps

$\rightarrow a(2)$, $\rightarrow b(i)$, $\rightarrow c((a^2+3)^2+i)$

are translated to

goto *a*[2−1], **goto** *b*[*i*−1],
goto *c*[(*a* **exp** 2+3) **exp** 2 + *i* + 5],

and the switch labels

a(2):, *b*(3): and *c*(−3):

thus

*a*1*l*:, *b*2*l*: and *c*2*l*:.

For routine headings the types of parameters and their names are added to the property list with an indication whether they are **addr** parameters or not.

The following routine headings

(*a*) **routine** *mat mult* (**addr** *s*1, *s*2, *s*3, **integer** *n*, *m*)
(*b*) **routine** *matrix mult* (**array name** *a*, *b*, *c*)
and (*c*) **real fn** *F* (**real fn,** *f*, **array name** *a*, **real** *x*, *y* **c** **integer name** *i*)

are translated

```
(a) procedure mat mult (s1, s2, s3, n, m);
    value s1, s2, s3, n, m;
    integer s1, s2, s3;
    integer n, m;
    begin
(b) procedure matrix mult (a, b, c);
    array a, b, c;
    begin
and (c) real procedure F (f, a, x, y, i);
    value x, y;
    real procedure f;
    array a;
    real x, y;
    integer i;
    begin
```

The procedure call

mat mult (*a*(1, 1), *b*(1, 1), *c*(1, 1), *n*, *m*)

becomes

mat mult (*addr*(*a*(1, 1)), *addr*(*b*(1, 1)), *addr* (*c*(1, 1)), *n*, *m*) where "*addr*" is an integer procedure which calculates the address of the variable given as its entry.

This translation of routine headings is done by the four routines

procedure
scan 1
scan 2
and *scan* 3

The initial entry is to routine "*procedure*", which distinguishes between actual routine headings and their specifications; if it is an actual routine heading with parameters, these are read into a buffer by "*scan* 1", with symbol editing to ensure a uniform entry. For example:

real *x*, *y* **integer** *i*, **integer name** *j*)

is stored in the form

−3 *x* , *y* , −2 *i* , −2 −10 *j*)

remembering that the negative integers are the delimiter word values. Note that an extra comma has been inserted.

The buffer is then scanned by "*scan* 2" and a **value** list drawn up, thus for the above example:

value *x*, *y*, *i*;

On the third scan by "*scan* 3" the parameter type list is produced; for the above example:

real *x*, *y*; **integer** *i*; **integer** *j*;

and finally an entry is made to the pcaption routines for the delimiter word **begin**.

routine and **fn**, **specs** are treated in the same way as actual routine headings, except that they produce no output and so appear to be ignored.

The delimiter word **result** can be translated in phase 3 by using the property list to find the last **integer fn**, **real fn**, or **fn** name. For example, the block of program

fn *fourier* (**integer** *a*, *b*, *n*)
result = $2\pi(sin(a)+sin(b)) \uparrow \frac{1}{2}/n$;
end

becomes

real procedure *fourier* (*a*, *b*, *n*);
value *a*, *b*, *n*; **integer** *a*, *b*, *n*;
fourier := 2*3·14159265359*(*sin*(*a*)+*sin*(*b*))
    exp·5/*n*;
goto *end*;
end: **end**;

The **result** and "*read* (*variable list*)" statements require more than one corresponding statement in ALGOL. Therefore, when they are used inside conditional statements, a compound statement must be introduced, adding a further complexity to the routines dealing with conditions. For example:

read (*a*, *b*, *c*) **if** $n < 0$

becomes

if $n < 0$ **then begin** *a*:= *read*; *b*:= *read*; *c*:= *read*;
end;

When, during the "*scan and output*" routine, a name occurs, an entry is made to routine "*name*"; this determines the name's type, e.g. **integer** or **real**, **array**, **routine**, or **switch**. If it is an array name, the round brackets are converted into square brackets by a bracket count similar to that described for modulus signs. If the name is a function, then a check is made whether it has any parameters. If so, the termination of each parameter expression is noted, a flag set and each parameter expression treated as a new instruction to be scanned and output. Any parameter indicated by the property list to be an **addr** type is transformed into an addr function entry. For example, from the routine definition

routine *mat add* (**addr** *s*1, *s*2, *s*3 **integer** *m*, *n*)

the following call

mat add (*a*(1, 1), *b*(1, 1), *c*(1, 1), *m*, *n*)

becomes

mat add (*addr*(*a*[1, 1]), *addr*(*b*[1, 1]), *addr*(*c*[1, 21]), *m*, *n*)

Phase 4

By the end of phase 3 the translator could do the work for which it was intended. The only facilities needed to run the translated program were: a rearrangement of the block structure to ensure that all **procedure** declarations appeared at the head of the block in which they occurred, and the provision of input, output and other procedures corresponding to the standard routines available in Atlas Autocode. This reshuffle also enables the translated program to be improved by removing unnecessary **begins** and their corresponding **ends**, and jumps to the next instruction—these may be introduced by the translation of the delimiter words **result** and **return**. As a further sophistication, the program text could be edited at this stage to give a more pleasing lay-out.

The translator's original target language was Atlas ALGOL. Recently, however, a new line-reconstruction input package for the KDF9 Post system has been written in the Cripps Computing Centre to avoid the expense of recoding thirteen seven-hole Atlas-coded Flexowriters. An outcome of this was the acceptance of Atlas ALGOL program texts for input to the KDF9 ALGOL compilers. The main difference between the Atlas and KDF9 ALGOL implementations is the input and output procedures; a standard set of Atlas type input and output procedures was prepared so that Atlas ALGOL programs could be run without any change. It follows that the majority of the input/output procedures needed by the translator are available, since the input and output subroutine for Atlas Autocode and Atlas ALGOL are similar. Some alterations are necessary; for instance "*read* (*variable list*)" and "*print fl* (*r*,6)" and "*newlines* (10)" which become on translation "*print* (*r*, 0, 6)" and "*newline* (10)" in Atlas ALGOL.

Atlas Autocode has further standard facilities for handling symbols, but these were rewritten in terms of the two basic routines "*read binary*" and "*punch binary*". These two routines read in or punch single characters from or onto paper tape. These, together with "*write text*", are the only routines which are machine-dependent in the translator.

Run-time differences

There are three fundamental differences, two involving the **cycle** and the corresponding **for** statement. The first arises from the interpretation of the **cycle** and **for** statement parameters. The Atlas Autocode increment and final value parameters in the **cycle** instruction are evaluated on entry to the loop and remain fixed throughout the evaluation of the loop. But the **for** statement's corresponding parameters are re-evaluated each time around the loop. Therefore Atlas Autocode programs which use the permanency property of the increment and final value become logically incorrect ALGOL programs on translation. Fortunately Nottingham programmers seldom achieve such sophistication. The second arises in that the ALGOL 60 report specifies no particular value for the loop variable at the natural conclusion of the loop. The third (pointed out by Mr. R. A. Brooker, of Manchester University) is conceptually similar to the evaluation of the **cycle**—**for** loop parameters. Routine parameters which are passed down by name in Atlas Autocode have any arithmetic expressions in the name of a variable (e.g. indices in array elements names—$a(i+2)$) evaluated on entry and the variable name passed down remains the same throughout that entry. ALGOL instead evaluates the arithmetic expression in the name every time the name is used in the procedure. Consequently it is possible to change the variable being passed down during the life of a procedure (Jenzen's Device).

The example illustrates this difference

```
begin
integer i; array A(1:10)
(real name x routine Y)
i=1; x=1
end
i=2; Y(A(i))
```

gives $A(2)=1$ whereas the ALGOL equivalent gives $A[1]:=1$; as the value of the index "i" will have changed before the assignment of the value 1.

Conclusion

The exercise has proved successful. At one time, it appeared that the Atlas Autocode–KDF9 Compiler and operating system provided by the Edinburgh University Computer Unit might make the translator redundant. In fact, it has had the opposite effect. In the past, Nottingham users have been discouraged from taking advantage of the Atlas Autocode programming language, fearing that their programs would be useless elsewhere. Now they have confidence that at any time their programs can be translated into the internationally accepted ALGOL standard.

Many user programs have been translated. A few required minor modifications because of their use of numerical character values, before they successfully translated. Only one program proved difficult to translate, because it made extensive use of **array fns**. All of the programs ran correctly and appeared to have none of the run time differences discussed in the previous section.

Versions of the translator exist in Atlas Autocode and in ALGOL. The translator was originally written to provide Atlas ALGOL object programs, but it is easily modified to match the ALGOL implementations on other machines.

Acknowledgements

The author would like to extend his thanks to Dr. M. L. V. Pitteway and Dr. C. A. G. Webster for discussions which greatly benefited the writing of the translator. He is also indebted to Mr. A. Chandler and other staff in the Cripps Computing Centre for help in the programming preparation and editing, and to the Director and staff of the Manchester Atlas Computer Laboratory for assistance in the running of the programs.

References

ALGOL Paper No. 10 (1966). Atlas Computer Laboratory, Chilton.

Brooker, R. A., MacCallum, I. R., Morris, D., and Rohl, J. S. (1963). "The Compiler Compiler", *Annual Review in Automatic Programming*, Vol. 3.

Bratley, P., Rees, D., Scholfield, P., and Whitfield, H. (1965). *Atlas Autocode Compiler for KDF9*, Edinburgh University.

Brooker, R. A., and Rohl, J. S. (1965). *The Atlas Autocode Reference Manual*, Manchester University.

Brooker, R. A., Rohl, J. S., and Clark, S. R. (1966). "The main features of Atlas Autocode", *The Computer Journal*, Vol. 8, p. 803.

Council of Scientific Policy, University Grants Committee (1966). *A Report of a Joint Working Group on Computers for Research*, H.M.S.O.

Foxley, E., and Neave, H. (1965). *Introduction to Programming in ALGOL*, Nottingham University.

Lunnon, W. F., and Riding, G. (1965). *The Atlas Autocode Mini-Manual*, Manchester University.

Naur, *et al.* (1963). "Revised report on the algorithmic language ALGOL 60", *The Computer Journal*, Vol. 5, p. 349.