Algol 60 Compilation and Assessment

B. A. Wichmann

August 1973

Preface

The ease and effectiveness of computer utilisation depend very heavily on the quality of the implementation of the programming language used. This book is dedicated to raising the quality of ALGOL 60 implementations. It gives an objective comparative assessment of over twenty existing compilers in accordance with numerical and non-numerical criteria, and describes in greater detail the techniques which contribute to high quality.

The book is required reading for any programmer embarking on an implementation of ALGOL 60; and is equally valuable for implementors of many subsequently designed languages. For users of ALGOL 60, it will not only assist in the selection of a high quality compiler, but it also shows how to use an existing compiler more effectively. Finally, for all designers and users of programming languages and other software, it gives an example of how the quality of software may be objectively defined and assessed.

The first chapter considers the compilation of the more straightforward aspects of ALGOL, by using an illustrative one-address computer. It is clear that most intercomparisons must depend upon statistical information of the use of ALGOL 60 and this is presented and analysed in chapter 2 to make it useful for performance assessment. These statistical data also give guide lines on the construction of efficient algorithms, which are detailed in chapter 3.

The problem of syntax analysis of ALGOL 60 source text is not considered in detail, as this is handled very ably elsewhere. However, some difficulties are enumerated in chapter 4, since a few existing compilers are known to handle these matters incorrectly.

As far as the user is concerned, the diagnostic features are all-important. If he cannot understand these diagnostics, or inexplicably gets the wrong answers, it is of little consequence that the compiler is “efficient” in terms of machine resources. These aspects are discussed in chapter 5. A related topic is the documentation of computer software, or in this case, ALGOL 60 source text. Several automatic aids and compiling options are considered.

In chapter 7, the compilation of the obscure points of ALGOL 60 is discussed. In many cases, these features are omitted from an implementation, not necessarily because of their difficulty, but because they are rarely used. A related problem is that of demonstrating the apparent correctness of a compiler. Certainly the known trouble-spots can be the subject of explicit tests, which, to be reasonably effective, must amount to about 100 programs. The design of such tests is described in chapter 8.

A detailed comparison of six existing compilers is made in chapter 9: for Atlas, KDF9, 1900, CDC 6600, B5500 and the 1108. The range of architecture enables a judgement to be made of the desirable features of a machine for good ALGOL object code. Numerous examples are given from test programs to illustrate the points made. The improvement of the object code in a compiler by various optimisation strategies is discussed in chapter 10. Simple local optimisation is compared with more ambitious schemes. This analysis leads naturally to what must be regarded as minimal machine features to support ALGOL 60 and the reasoning behind this is presented in chapter 11.

The implementation of ALGOL 60 implies various mechanisms within both the compiler and object-code. These mechanisms are capable of supporting features which are not present in ALGOL 60, and are hence candidates for extensions to the language. This leads to a discussion on language design in the final chapter which is arranged in order of increasing complexity from minor extensions to a brief analysis of ALGOL W and ALGOL 68.

I should like to thank all the "ALGOL men" with whom I have had much useful discussion, enabling this book to be written. Professor C. A. R. Hoare, Mr R. Scowen and Mr M. Woodger have contributed substantially to my understanding of ALGOL 60. Dr P. T. Cameron and Mr J. Wielgosz have contributed to some of the work reported in chapter 9, and Dr C. Phelps to that in chapter 2; this collaboration has proved very useful and I am glad to acknowledge their support. I should also like to thank Mr D. S. Akka for permission to quote from his M.Sc. Thesis.

This work would not have been possible without the permission of the Director of the National Physical Laboratory to use the published reports and papers upon which this book is largely based. Any opinions expressed however, are mine, and not necessarily those of the Laboratory. Finally, I should like to thank my wife for her support in the preparation of the book.

August 1973 B.A.W.

Logical order of the chapters

$$\begin{tikzcd}
1 \arrow[r] & 5 \arrow[dr] & & & & & \\
& 6 \arrow[r] & 7 \arrow[dr] & & & & \\
4 \arrow[ur] \arrow[rrr] & & & 8 \arrow[dr] & & & \\
2 \arrow[rrrr] \arrow[dr] & & & & 9 \arrow[r] \arrow[dr] & 11 \arrow[r] & 12 \\
& 3 & & & & 10 &
\end{tikzcd}$$

Contents

Chapter 1

The Generation of Machine-Code

1.1 Arithmetic Expressions

The generation of machine code is usually performed from a reverse Polish string or from a tree structure rather than from the ALGOL source text itself. A good discussion of such forms is to be found in Gries (1971). This is ordinarily output from a syntax parse, although some additional processing may be required, especially if optimisation is performed. The exact form of the input to the code generation part of the compiler is not relevant to this chapter except in that it may impose limitations upon the machine-code produced. Arithmetic expressions represent a very important part of the compiler since the inner loop of most programs will be spent evaluating expressions. The semantics of such expressions are well understood and a considerable amount of literature is available giving details of sophisticated techniques of code generation. Nevertheless many such techniques are irrelevant simply because the average length of expressions is very small (see 2.3.1.11).

1.1.1 SIMPLE ARITHMETIC EXPRESSIONS

Access to a simple variable can almost always be accomplished in one instruction — the details of this appears in 1.8. So in the following examples it is assumed that variables can be accessed as in assembly language programming. The computer used to illustrate the production of machine code has one floating point accumulator (it is described more precisely in Appendix 1). Hence real expressions must be evaluated in this accumulator. So the expression

$$y \times z + u$$

would result in the code

```
LDA    Y
MLA    Z
ADA    U
```

The convention being that the result of an expression is to leave the value in the accumulator. Even with multiple accumulator machines, a rigid convention is sometimes applied simply because a more general scheme would have to take into account the many different contexts in which an expression can occur.

Care must be taken to ensure that the order of evaluation of an expression is as stated in the Report (R3.3.5). Firstly floating point operations are not associative nor do they follow any simple rules (Knuth, 1969), so that any apparently reasonable reordering may actually give different answers. Of course, the compiler-writer may be more knowledgeable on these matters than the programmer, nevertheless the program and the Report must be the final arbiter. Secondly there is the vexed problem of side-effects on both function calls and call-by-name. Although it may be possible by global analysis of the source text to discover whether such side-effects are being used, the potential gain from this cannot justify the effort required. Producing code which allows for side-effects is likely to involve an extra store and load for some function calls. Yet the time involved in the function call (or the evaluation of a call-by-name parameter) is at least ten times greater than this. Even with the evaluation of integer expressions, the order of evaluation can be significant because of overflow conditions.

From the compiler-writers point of view, it is preferable to have overflow as an interrupt condition (possibly masked). Provided the position of the interrupt, if it occurs, is known, the run-time system can generate suitable diagnostic information. On some machines, explicit test instructions must be generated. This is so on KDF9, where the jump on overflow instruction does not give the position of the test. Hence on overflow it is impossible to give accurate indication of the point of error. Some users may wish to test for overflow explicitly in order to alter a scaling factor so that a calculation can be redone. This is a very reasonable requirement which the compiler-writer should allow for. A masked interrupt condition seems the obvious solution, but care must be taken to ensure that if overflow does occur, the user is aware of the fact. With the generation of explicit tests, assignment would appear to be the obvious place. This is so on KDF9, and has the disadvantage of generating rather a large number of instructions. A test on procedure exit is more reasonable from this point of view but has the disadvantage of allowing indeterminate values to be assigned before the error is detected. Similar problems arise with underflow. On some machines this is regarded as an error (B5500) but on others it is an interrupt condition which can be masked (360), while on others zero is produced with no indication at all (KDF9). If no detection of underflow is possible, it can be very awkward if the standardisation convention means that accuracy is lost completely when the number just underflows. It is important for the users that the conventions adopted concerning overflow and underflow should be consistent, and include such things as function evaluation, the subroutines for exponentiate (see 1.1.4.2) and other system functions. Overflow on integer operations is much less common, but nevertheless can be important. The run-time control routines do many fixed point calculations, so there is always a risk that overflow may be set as a result, rather than by the user program.

Integer expressions tend to be extremely short, a very frequent example being the subscript expression which is dealt with more fully in 1.2. For this reason, handling the full complexity of long expressions in an optimal manner is not as important as with real expressions in most scientific programs. So the convention is adopted that integer expressions will be calculated only in register 1, even though the machine has several integer registers. Hence the expression

$$i \times j + k$$

would generate

```
LDI,1    I
MLI,1    J
ADI,1    K
```

Unfortunately many machines insist on giving a double length result in two registers for a fixed point multiply. This means that the double length answer must be correctly shortened, taking into account the possibility of overflow. To produce a double length result when it is so rarely required seems unnecessary. Unary plus need never generate machine-code, although it can effect the semantics (see Randell, 1964, p 220). On the other hand a unary minus often requires extra instructions. If a LOAD NEGATIVE order exists on the machine then the unary minus can often be incorporated in the load operation. In any case, an additional instruction to change the sign of a register is likely to be fast as no operand is required.

1.1.2 ANONYMOUS WORKING STORE

In the evaluation of expressions, and in some other contexts, the compiler uses extra core-store locations which do not directly arise from the user-declared variables. When necessary, the compiler must allocate additional storage, ensuring that it remains available for the duration of the expression calculation. Because of recursive function calls, fixed storage locations cannot be used, but nevertheless it is usually possible to access the necessary variables in one instruction as with user variables. This problem is covered more fully in 1.8.

In the following examples, the working variables are given the names TEMP1, TEMP2, etc. So one would expect

$$x + (y \times z - u) \times w - a$$

to generate

```
LDA    X
STA    TEMP1
LDA    Y
MLA    Z
SBA    U
MLA    W
ADA    TEMP1
SBA    A
```

The use of temporary storage in this case, is, of course, unnecessary but it is typical of many current ALGOL 60 compilers. Even without elementary reordering of the instructions, some saving can often be made. For instance, the reverse divide and subtract instructions or an instruction to interchange the accumulator and the operand can sometimes be used. In any case, the savings tend to be slight compared with the volume of code which is clearly demanded by the user program.

In the machine-code given in the last example, strict left to right evaluation of the expression was maintained. However, some compilers in analysing <simple variable> <operand> <complex expression>, generate the code for the complex expression first in the hope that the use of temporary storage will be avoided. This is particularly simple when the expression is stored as a list structure rather than a reverse Polish string.

Few ALGOL compilers attempt detailed analysis of expressions to reduce the amount of anonymous working store to an absolute minimum. With multiple register machines, evaluation can be faster by exploiting the registers properly, although the number of instructions executed is often not significantly different. Removing instructions to access working storage shortens the code generated, and this can be significant in machines with a small address limit on instructions or with a small core-store. Examination of post mortem dumps from the KDF9 compiler shows that working store variables are much less numerous than user-declared variables which in turn are less than array storage.

1.1.3 TYPE CONVERSION

Integer expressions can appear as sub-expressions in a real expression, in which case the calculated value must be converted to real. Subscript expressions (or bound-pair expressions) which have a real value must be converted to integer, in the same way as the assignment of real expressions to an integer. Conversion of integers to reals is very common, but the converse is comparatively rare (see 2.3.4.2).

The number of instructions involved in the two conversions varies very substantially from machine to machine. Even if the basic instructions exist for separating the exponent and mantissa, it may require several more instructions to fix a floating point variable in the way demanded by the Report. Error conditions arise due to the different ranges of values, and care must be taken to ensure correct rounding in marginal cases.

An examination of the Algorithms published in the Communications of the ACM, or indeed most user programs, shows that integer constants are very common in real expressions. On the other hand integer variables do not occur so frequently in real expressions. Hence a lengthy type conversion of an integer constant to a real should be avoided by incorporating this in the compiler. Very few existing compilers do this, although it presents little difficulty (see 10.3). On the KDF9 computer, it has been found that a large proportion of the real to integer type conversions are caused by the standard input procedure. This is a real function, which can be used for reading integer values. Apart from the double conversion overhead, it also has the disadvantage that if the input media contained 3.4 (say) when reading an integer, no error indication would be given. The loss of information on this type conversion is a potential source of error, which is perhaps worthy of a warning by the compiler.

With the computer used for illustrations of generated code, the specification of the type conversion instructions is obvious. The FLOAT macro takes the value of an integer expression in register 1 and produces a standardised floating point number in the floating point accumulator. The converse takes place with the FIX macro. The exact instructions used are not given, since it is too machine-dependent to be of general interest. Use of subroutines may not be very effective on a multiple register machine, since one may need to vary the input and result registers.

Consider the following example, where the FORTRAN convention is used for variable types,

$$i + x + y$$

would clearly give

```
LDI,l     I
FLOAT
ADA       X
ADA       Y
```

But now consider

$$i + x \times y$$

where strict left to right evaluation is used (a more complex example would be necessary otherwise). This would give

```
LDI,1     I
STI,1     TEMP1
LDA       X
MLA       Y
STA       TEMP2
LDI,1     TEMP1
FLOAT
ADA       TEMP2
```

This is unfortunately typical of the rather poor code produced by most ALGOL compilers. Register 1 is stored because the remaining expression might require it (for instance, for a subscript expression). This illustrates that temporary variables should be stored in the type required later, i.e. type conversion is done as early as possible. This would give

```
LDI,1     I
FLOAT
STA       TEMP1
LDA       X
MLA       Y
ADA       TEMP1
```

It is not now necessary to detect whether the remaining expression requires register 1.

1.1.4 OPERATORS NOT AVAILABLE ON THE HARDWARE OF THE MACHINE

Two of the arithmetic operators are commonly not available as one or two instructions on a computer. These are integer divide ($\div$) and exponentiate ($\uparrow$).

1.1.4.1 Integer divide

This is for dividing two integer values, and is defined to be
$a \div b = sign(a/b) \times entier(abs(a/b))$ (Report 3.3.4.2).

This apparently does not conform to the usual machine conventions. Most computers have a single integer divide instruction, but it does not give the same answer as the above when one or both the arguments are negative. This forces the compiler writer to use a subroutine for the operation, even though it is probably never executed with negative arguments. The only difficulty from the point of view of code generation is that the arguments must be placed in fixed locations for the subroutine. Using a second register, say number 2, the obvious convention would be to put b in register 1, and a in 2. Hence the machine code would be

```
CALCULATE    ‘a’
STI,1        TEMP1
CALCULATE    ‘b’
LDI,2        TEMP1
DIVIDEI
```

This means that a general integer expression could use the second register although this would be rare. Clearly this register may be more effectively used elsewhere, but its use both in integer divide and (say) to reduce the use of temporary storage can clearly cause difficulty in the compiler. In fact, a second register is required for array subscript working, which can also be used for integer divide.

A frequent use of integer divide is the following

$$i \div 64 \times 64 - i$$

which gives

```
LDI,1    I
STI,1    TEMPI
LDI,1    64
LDI,2    TEMP1
DIVIDEI
MLI,1    64
SBI,1    I
```

Removal of the use of temporary storage is clearly possible, indeed the division could be done on most machines by an arithmetic shift but such optimisation is not likely to be worthwhile. Optimisations of this type are done extensively by the PASCAL compiler (Wirth, 1972), but the language is designed for systems programming where such operations are more common.

All the above assumes that the types of arithmetic expressions are deduced at compile time. This means that an attempt at integer divide with real operands will be an error detected by the compiler.

1.1.4.2 Exponentiation

This is by far the most complex operator, and can rarely be implemented strictly according to the Report (3.3.4.3). The major difficulty is that the type of $i \uparrow j$ depends on the sign of j. If j is less than zero, then the result is real, otherwise it is integral. Since type handling at run-time is far too time-consuming to be considered for a practical compiler, some method must be used to overcome this.

The most straightforward solution is to redefine the operation so that $i \uparrow j$ is always of real type. The actual values produced by the operator can be the same as the Report, so for most purposes the solution is perfectly adequate. The only disadvantage is that real operations may be involved where none was intended, and that numerous type conversions would arise. To produce a real for $i \uparrow 2$ seems rather unnecessary. In an ALCOR compiler, Grau (1967), defined i ↑<unsigned integer> to be of type integer, which avoids this particular difficulty. However, replacing an integer constant by a variable could then have effects on the program which is not possible in other ALGOL constructs.

Alternatively, $i \uparrow 2$ with $j < 0$ could be regarded as an error so that $i \uparrow j$ could always produce an integer result. In the case where j is negative, it is reasonable to assume that the programmer is aware of this, so that he can write $(1/i) \uparrow (-j)$ to give an obviously real result.

Yet another solution is to give a floating point answer for $i \uparrow j$ but to regard it as a way of representing an integer if the need arises. Consider $i \uparrow j \div k$; when $j > 0$ the floating point value for $i \uparrow j$ is converted to integer for the integer divide operation. If $j < 0$, then $i \uparrow j$ will produce a non-integral valued floating point number (except when i = 1), and so can be trapped by the conversion subroutine. This subroutine, therefore, is not the macro FIX of 1.1.3 but one which fails unless the floating point number is an exact representation of an integer. Floating point operations can only be allowed with this "integer or real" type if the floating point hardware produces exact results with integers. Even then, mantissa overflow conditions could arise, producing different results from integer calculations due to rounding. This solution seems the least elegant of the three, but is described in detail because this is the method adopted by the KDF9 (Kidsgrove) compiler (see 2.3.4.2 and 9.4).

Even apart from type handling difficulties, the exponentiation subroutine is not as easy as one might expect. In extensive tests on the two KDF9 compilers used at the National Physical Laboratory, both were found to be in error, the reason being the many exceptional cases listed in the Report. For instance $x \uparrow 0$ (x real) produces 1.0 except when x = 0.0. Since equality of floating point numbers implied in this test is somewhat dubious, the discontinuity at zero is very surprising. Knuth has insisted upon continuity in ALGOL W. As far as the implementers are concerned, the writers of the Report could have saved a lot of work by defining exponentiation by an algorithm using the other operations (as was done with integer divide).

To illustrate the ease of defining the operator in terms of ALGOL 60, the first method is given where $i \uparrow j$ is always real. The four cases according to the type of the two operands can be reduced to two by always converting the base to type real. The two remaining cases are, of course, logically quite different, and can be given by two procedures.

```
real procedure $expr(x, r)$;
    value $x, r$; real $x, r$;
    if $x > 0.0$ then
        $expr := exp(r \times ln(x))$
    else if $x = 0.0 \wedge r > 0.0$ then
        $expr := 0.0$
    else
        error

real procedure $expn(x, n)$;
    value $x, n$; real $x$; integer $n$;
    begin real $result$; integer $i$;
```

```
if $n = 0$ then
        begin
        if $x \neq 0.0$ then $result := 1.0$ else error
        end
else
        begin
        if $n < 0$ then
                $result := x$
        else
                begin $x := result := 1/x$;
                $n := -n$
                end;
        for $i := 2$ step $1$ until $n$ do
                $result := result \times x$
        end;
$expn := result$
end
```

The procedure *expn* is perhaps not quite as intended by the Report. On evaluating $10.0 \uparrow (-1000)$ the result will underflow rather than overflow. Also $0.0 \uparrow 0$ will fail because of division by zero rather than by an explicit test. The routines are quite short except for the call of *exp* and *ln* in the real case. If a computer has an independent compilation system, then calls of *expr* should be arranged so that *exp* and *ln* needed not be in the user program unless they are needed.

The procedure *expn* evaluates powers by repeated multiplication. However, Knuth (1969) discusses at great length faster methods of calculating powers. These methods do, however, produce different values with floating point variables. Experiments on KDF9 reveal a difference of about half a bit between $(x \times x) \times (x \times x)$ and $x \times x \times x \times x$. So in practice, the differences are very small except for large exponents. In fact, exponentiation is very largely used for squaring and cubing, so the simplest method is fastest because of the smaller overheads.

1.1.5 NESTED EXPRESSIONS

As far as the generation of machine code is concerned, nesting of expressions can occur with or without explicit bracketing. Operator priority is equivalent to a particular bracketing, and so gives the same problems in code generation. If the code generation is from a reverse Polish string, then only the necessary information is preserved. A list structure is often more convenient if some optimisation is done, and this may preserve the explicit bracket structure. For instance, with $x + (1.0) + y$ the Atlas ALGOL compiler generates the equivalent of

```
LDA          X
STA          TEMP1
LDA          1.0
ADA          TEMP1
ADA          Y
```

In other words, the redundant brackets have produced unnecessary use of temporary storage. This does not appear to be a practical deficiency of the compiler, since it very rarely occurs in ordinary programs. Compilers, can therefore, deal with arbitrarily complex nesting of expressions by use of temporary storage.

1.2 Array Variables

The handling of array variables in ALGOL 60 is very different from that of simple variables or from the conventional assembly code equivalent. There are two basic reasons for this. Firstly, the size of arrays is not, in general, known until execution time, and secondly the array may be a formal parameter to a procedure. These difficulties are usually overcome by having an "array word".

1.2.1 THE "ARRAY WORD"

This word is accessed in the same manner as a simple variable, and gives details of the array addressing. The exact form that this word takes (indeed it could be more than one machine word) depends critically on the instruction code facilities. Usually it consists of at least two parts. Firstly the address of the word $a[0, \ldots, 0]$. Access to the variable $a[i]$ can then be found by simply adding i to this. Secondly the array word often contains the address of "further details" of the array. Again, the form that this will take will depend upon the accessing technique and the machine instructions. Two different accessing techniques are in common use in ALGOL compilers and are illustrated below.

1.2.2 THE DOPE VECTOR TECHNIQUE

This is the most commonly used method, whereby the address of an n-dimensional array variable is calculated by $(n-1)$ multiplications. For instance, if one has

array $e2[1:6, 3:8]$

and the array elements are stored in the order $e2[1,3], e2[2,3]$, etc. then one has

$$add\{e2[i,j]\} = i + 6 \times j + add\{e2[0,0]\}$$

The constant 6 in this formula is not, in general, known until execution time, and so must be stored as data in the program. As there are $(n-1)$ such constants, it is usual to store these separately from the array and the array word. The constants, of course, represent the number of words apart of the array elements which have all subscripts equal except one, which is different by one. The amount of space required for these constants can be calculated at compile time, and their value set on the array declaration, as they can be calculated from the subscript bounds.

To illustrate this with a more complex example, say one has
array $e3[1:3, -6:6, 1:8]$
Then

$$add\{e3[i,j,k]\} = i + 3 \times j + 39 \times k + add\{e3[0,0,0]\}$$

The array word would then consist of $add\{e3[0,0,0]\}$ and a pointer to the constants 3 and 39. An alternate representation of the address calculation is

$$add\{e3[i,j,k]\} = i + 3 \times (j + 13 \times k) + add\{e3[0,0,0]\}$$

Of course, the subscript order may be inverted, since in the above form it is likely to be more convenient to evaluate $(i \times 13 + j) \times 3 + k$ as the subscripts are themselves evaluated in a left to right order. Hence it is more convenient if arrays are stored by columns on most one-address computers, as the address calculation can proceed with the subscript evaluation, using only one extra register.

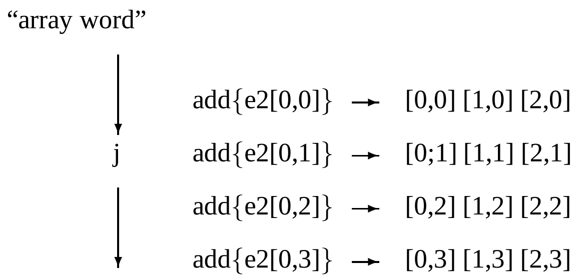
The majority of ALGOL compilers do the address calculation for an array variable for every occurrence of the subscript list in the source text. This calculation is often performed by in-line code for one or two dimensional arrays, but by subroutine for higher dimensional arrays. Illustrations of the dope vector technique are given in 9.6 and 9.8.

More information must usually be stored about an array apart from the dope vector constants, for instance, for subscript bound checking. Further requirements are discussed in 5.3.1 and 7.3.

1.2.3 THE CODE WORD METHOD

It is possible to access array variables without fixed point multiplications at the cost of additional data storage. With one dimensional arrays, a simple addition gives the required address. For two dimensional arrays, a vector of addresses can be used, so that one level of indirection gives the required address. Say one has

array $e2[0:2, 0:3]$



"array word"
j
add{e2[0,0]} → [0,0] [1,0] [2,0]
add{e2[0,1]} → [0;1] [1,1] [2,1]
add{e2[0,2]} → [0,2] [1,2] [2,2]
add{e2[0,3]} → [0,3] [1,3] [2,3]

This can, of course, be generalised to any number of dimensions, and with arbitrary lower bounds. The main disadvantage of this technique is that additional storage is required which depends on the size of the arrays declared (and hence could be quite significant).

A substantial advantage is that it is comparatively easy to add subscript bound checking to the logic of the access mechanism. Every indirect address pointer (called code word) must contain three parts. The address of the zeroth element down the chain, the largest and smallest permissible subscript value. Hardware is available on some machines to implement code word accessing. The main advantage of such hardware is that bound checking can be done in parallel with the core accessing, that is, with little or no additional overhead.

1.2.4 THE STORAGE LAYOUT

Whatever technique is used for array variable accessing, three elements of storage are involved: the "array word", the dope vector or code words, and the array itself. The compiler writer may not have complete freedom in the way these three parts are stored.

Many systems offer a mixed language facility whereby "modules" of independently compiled code from different source languages can be linked into one program. Such a system clearly necessitates agreement as to whether matrices are stored by rows or by columns (and the equivalent for higher dimensions) and indeed whether $a[1]$ is stored in the word before $a[2]$. If the intention of the mixed language system is that array parameters should be handled in a uniform way, then further restrictions arise. Under such circumstances, the compiler writer may either use the mixed language system array parameter as the "array word" or construct (when necessary) his own array word from the agreed standard. For instance, in the KDF9 Egdon system (Burns, 1966), the ALGOL compiler uses the system parameter as the "array word". For a one dimensional array, this gives the address of $a[1]$, so that the one dimensional array accessing code always subtracts one from this before adding the subscript. The system convention here is based upon FORTRAN, giving the address of the first element of the array. Since FORTRAN does not require a dope vector or codewords, the FORTRAN compiler may not produce adequate information about an array for it to be passed as a parameter to ALGOL—especially if bound checking is required. Under such circumstances, the user may have to call special routines to set up the necessary information.

1.2.5 MACHINE CODE EXAMPLES

As will be seen from 2.3.1.9, a large proportion of array accessing is for one dimensional array elements. Both the dope vector and code word techniques are the same in this case-the address of $a[0]$ is added to the subscript to obtain the required address. This addition can either be performed explicitly, or can often be achieved by use of an appropriate address mode or address modification. The address itself is, of course, wanted in many contexts, but this is much less frequent than the value which is described here.

The subscript calculation is, of course, an integer calculation, so that there is a conflict on the illustrative machine for the use of the integer registers with integer array variables. This does not happen with real expressions where the floating point accumulator is used for the main expression so the integer register is available for the subscripts.

Taking the case of real array variables first, one has

```
<calculate subscript>
ADI,1      A          A contains address a[0]
LDA        1,(0)      To fetch a[<subscript>]
```

So an expression

$$a[i] + b[j] \times x$$

would be translated as

```
LDI,1      I
ADI,1      A
LDA        1,(0)
LDI,1      J
ADI,1      B
STA        TEMP1
LDA        1,(0)
```

```
        MLA          X
        ADA          TEMP1
```

To summarise, with real one dimensional arrays, about three instructions are required for access, and temporary storage is not necessary even with general subscripts.

The conflict over the use of register 1 makes integer array accessing more difficult. It is wise to keep to the convention that integer expression always leave the result in register 1 so that the compiler logic does not become too complex (increasing the possibility of error). The simplest solution is to optimise the case when the subscript is a single variable. In this case, the working register 2 can be used for the address calculation.

So one has

$$i + i1[j]$$

generating

```
        LDI,1        I
        LDI,2        J
        ADI,2        I1
        ADI,1        2,(0)      add to index register 1 the contents of
                                the word whose address is (register 2 + 0).
```

In all other cases, register 1 is dumped into temporary storage to allow calculation of the subscript.

So $i + il[j + k]$ would give

```
        LDI,1        I
        STI,1        TEMP1
        LDI,1        J
        ADI,1        K
        ADI,1        I1
        LDI,2        1,(0)
        LDI,1        TEMP1
        ADI,1        REG2       Register 2 accessed like a core location.
```

Clearly, the subscript could have been calculated in register 2, but in this sort of optimisation, numerous checks must be applied (for instance, the subscript does not use integer divide). Using one register from subscript calculations and other for integer expressions does not solve the problem in general because of nested subscripts. Obviously nested subscripts are sufficiently rare to make this approach a very practical one. In fact, nested subscripts cause no more difficulty than ordinary nested expressions, for instance

$$i1[i2[i3[j]]]$$

would give

```
        LDI,l        J
        ADI,l        I3
        LDI,l        1,(0)
        ADI,l        I2
        LDI,l        1,(0)
        ADI,l           11
        LDI,l           1,(0)
```

Two dimensional arrays are more difficult to handle because two subscript expressions are involved. Almost always, some restriction must apply to one of the subscripts to avoid the use of temporary storage. In other words, to produce acceptable machine code the compiler is forced to do some optimisation. With $a2[i, j]$, the first subscript could be general and its result left in register 1. At this stage, with the dope vector technique the most advantageous approach would be to multiply by the number of columns and then add j—assuming j to be a simple variable and that matrices are stored by columns, i.e. $a2[i, j]$ generates

```
        LDI,l           I
        MLI,l           A2+l      part of "array word" giving number of columns
        ADI,l           J
        ADI,l           A2        address a2[0,0]
        LDA             1,(0)
```

Here it is assumed that the number of words in a column is stored in the word following that containing the address of $a2[0, 0]$.

Temporary storage would be used in the case where the second subscript expression is more complex.

$a2[i, j \times k]$ might give

```
        LDI,l           I
        MLI,l           A2+l
        STI,l           TEMP1
        LDI,l           J
        MLI,l           K
        ADI,l           TEMP
        ADI,l           A2
        LDA             1,(0)
```

Temporary storage could have been avoided with $j + k$ instead of $j \times k$, but few compilers would do this.

With integer array variables, a reasonable approach would be to use register 2 for the address calculation provided both subscripts were simple integer variables or constants.

Thus $j + i2[k, l]$ generates

```
        LDI,l           J         could be general expression
        LDI,2           K
        MLI,2           I2 + 1
        ADI,2           L
        ADI,2           I2
        ADI,l           2,(0)
```

One can see from this that a two dimensional array element can be accessed in about five instructions with simple subscripts, but that the code generated can become much larger with a slight increase in complexity of the subscripts.

The code word technique for two dimensional array element is very similar except addition only is used. The word addressed by the array identifier is assumed to be the address of the zeroth element of the vector containing the addresses of $a2[i, 0]$.

$a2[i, j]$ therefore gives

```
LDI,l      I
ADI,l      A2
ADI,l      1,(0)     reg1 contains address a2[i, 0]
ADI,l      J
LDA        1,(0)
```

Although access to array elements with constant subscripts cannot be handled in the manner of FORTRAN (i.e. as fixed storage locations) improvements can be made to the simple machine code given above. For instance $a1[3]$ would ordinarily generate

```
LDI,l      3
ADI,l      Al
LDA        1,(0)
```

which can be replaced by

```
ADI,l      Al
LDA        1,(3)
```

This is well worthwhile, since many programs have an initialisation sequence to some elements of an array using constant subscripts. The time increase may not be significant, but the saving in instruction space may be very helpful. A similar improvement may be possible with two dimensional array access with either the dope vector or codeword methods—one instruction can be saved with both on the illustrative machine.

The handling of three or more dimensional arrays can be accomplished with code sequences similar to those given already. However, the size of the generated code is likely to be prohibitive on many machines. Hence subroutines are often used at this stage. The difficulty here is that arrays of arbitrary dimension must be considered. In general, the number of subscripts will certainly exceed the capacity of the registers, so alternative storage must be found. Clearly the method used will depend on the subroutine facilities, and the conventional method of dealing with parameters. The most general technique would be to use stacked storage (see 1.9) and mimic the procedure call mechanism. Otherwise a set of consecutive temporary storage words can be used, in which case their starting address must be passed to the address calculation routine. The routine should give the address of the array element, the final fetch being done by open code. This is so that the same routine can be used on the left hand side of assignment statements (see 1.6).

With the dope vector technique, use of subroutines is often necessary because of an extra level of indirection involved in accessing the dope vector. If direct access is possible to the dope vector, say, a set of consecutive locations after the array word, then

$$a3[< s1 >, < s2 >, < s3 >]$$

would give

```
LDI,2         A3
Code for s1
MLI,I         A3+1
ADI,2         REG1
Code for s2
MLI,I         A3+2
ADI,2         REG1
Code for s3
ADI,2         REG1     Register 2 contains final address.
```

Strictly, the loading of the first address in register 2 need not be done until after the first subscript is calculated (which may be an advantage, because of nested subscripts or integer divide). Some computers have a multiply and add instruction which allows a significant saving in code space (see 9.8).

The equivalent coding with the codeword method automatically obtains access to the addressing information through the indirection. In fact the code produced would be

```
LDI,2         A3
Code for s1
ADI,2         REGl
LDI,2         2,(0)
Code for s2
ADI,2         REGl
LDI,2         2,(0)
Code for s3
ADI,2         REG1     Register 2 contains the final address.
```

Machines with an indirect addressing mode of the right form can perform the add and load in one instruction. On conventional machines with many index registers, there are substantial possibilities for more extensive optimisation than is illustrated here. This subject is considered further in chapter 10.

1.2.6 SUBSCRIPT BOUND CHECKING

Most computer programs contain an error, and using subscripts outside the bounds of the array is one of the most frequent errors. Hence every compiler should offer an option on subscript checking. Unfortunately, on most computers, quite a few instructions are necessary for this, forcing the compiler writer to use subroutines. Hence programs are likely to run substantially slower with bound checking. Quite a few modern computers provide some special instructions for rapid bound checking (ICL 4100 series, B5500, B6500, Manchester MU5 machine, Univac 1100 series). Such machines can often access array elements with bound checking at roughly the same speed (relative to the general machine speed) as more conventional machines without bound checking.

Two methods can be used for bound checking, each subscript can be checked separately, or merely a check made that the final address lies within the core area allocated to the array. With the code word technique it is usually easier to check each subscript as one progresses down the chain of addresses. This means that each "address" must contain three parts, the maximum value of the subscript, the minimum value, and the next address in the chain. Many computers have facilities whereby one word can be used as two integers (or even three on KDF9). Such facilities can be very useful for storing dope vectors, codewords and subscript information. The compiler writer may be severely constrained in the method of handling storage by an independent compilation system. In order to run programs partly compiled with bound checking and partly without, a common method of storing the array information must be used. The tendency is for this method of storage to be determined by the faster array accessing method without bound checking—perhaps making it impossible to check each subscript.

The simplest method is merely a final check on the address. If this is in register 2 (say), then it is only necessary to pass the address of the dope vector to one subroutine for the check. The subroutine does not even require to know the number of dimensions of the array. The Univac 1100 series has a convenient instruction for this; it checks that the operand lies between two integers stored in a register. Bound checking must be done on the B5500 and B6500 as the hardware does not allow access by any other means. In fact core-protection is based upon this mechanism. One defect of the Whetstone system, is that although it is very slow due to interpretation, the subscript check is an overall one rather than on each subscript, the reason being that insufficient information is preserved on a declaration to do a complete check—which in turn is determined by compatibility with the Kidsgrove compiler.

1.3 Constants

From the point of view of the ALGOL syntax there are three forms of "constants", unsigned integers, unsigned numbers and strings. Each needs to be treated somewhat differently by a compiler. Unsigned integers appear frequently in most programs. As mentioned before, integer constants appearing in real expressions should be converted at compile time—i.e. translated as if ".0" appeared after the integer. The constants 0 and 1 appear very frequently, and so advantage should be taken of any quick methods of handling these values. Many computers have a "literal" facility whereby the "address" field of an instruction can be loaded directly into a register. The maximum size of integer that can be handled this way is restricted, but the compiler should take advantage of it because of the saving in program size as well as execution speed.

Real constants are usually more difficult to handle. Firstly there is the substantial problem of reading the value accurately. Experiments on KDF9 have been conducted with two different input routines. This consisted of reading n/l00 (n = 1 to 99) as data and comparing with the computed value. Only 60% were equal with one routine whereas the other gave over 90%. The less accurate one was also shown to be biased. The literal facility may not be appropriate for real numbers in that the non zero bits are placed in the wrong part of the accumulator. It may be possible to correct this with a shift, or may be better not to use the literal instruction.

The pooling of integer and real constants can be quite important in that it may make the difference whether a large program will fit in the machine or not. This may be particularly significant if constants occupy a restricted address area. Unfortunately many independent compilation systems do not allow constants from separately compiled modules to be pooled—thus aggravating the problem. It is important that the compiler should attempt to pool constants even when its own pool area is full. For instance, to generate a new word for every constant when the pool is full would be wasteful. It is probably best to empty the existing pool and start another.

Strings are very different from integer and real constants in that the storage required varies—and can be quite large. Strings can be stored in virtually any way, since it is merely a matter of agreement between the compiler and run-time routines. Some systems use an internal numeric coding for the ALGOL basic symbols. This has the disadvantage of requiring conversion at run-time to the peripheral code—since presumably the string is required for an output procedure. Hence if a computer system has a peripheral independent character set it seems appropriate to store it in this form.

The compiled program may contain other constants generated by the compiler—for instance, as parameters to the run-time routines. These should be pooled in a similar manner. The parameters, where possible, should be specified so that a literal facility can be used—for example, if negative literals are not allowed, positive parameters should be used.

Some systems calculate subexpressions only involving constants at compile-time. In the author's opinion this is a somewhat dubious facility for the following reasons. Firstly, it is better that the programmer does this "optimisation" himself, avoiding the recalculation of the constant expression every time the source text is resubmitted to the system. Secondly there is the danger that the compiler may not calculate the expression in *exactly* the same way as compiled code (i.e. as far as round-off is concerned). On the other hand, some constant expressions may be generated by a compiler which the user cannot optimise himself. For instance, access to an array with constant subscripts, in which case the compiler should attempt the optimisation.

1.4 Boolean Expressions

The production of machine code for Boolean expressions not involving relational operators is almost identical with that of arithmetic expressions covered in 1.1. There is, however, more freedom in the internal representation of Boolean values. Integers and reals have virtually automatic machine counter-parts in fixed and floating point variables. The representation of **true** and **false** is largely determined by the nature of the logical instructions which could perform $\neg$, $\wedge$, $\vee$, $\equiv$ and $\supset$. The most common convention is to use two of the integer values -1, 0, 1. Having settled upon a representation it is rare to find that all the Boolean operators can be conveniently handled by open code—very short subroutines may be necessary in some cases.

It is interesting to note, that although all present scientific computers work in binary, Boolean values (even for arrays) use a whole word rather than a single bit. There are several reasons for this. Firstly it is substantially easier to use the same array accessing methods as for as integers and reals. Secondly on most computers accessing a single bit requires many more instructions. Lastly, when a Boolean value is in a register it occupies more than a bit, so two representations would have to be used. Hence a user faced with the storage of a large binary matrix is likely to choose to represent it as an integer array to conserve storage. An interesting feature of the X8 compiler is that Boolean array values are represented as one bit.

1.4.1 THE RELATIONAL OPERATORS

There operators are used far more frequently than the other purely Boolean operators. The usual function is to make a simple numerical test to alter the flow of control. The basic test produces a Boolean value, and in general, a jump is performed if the result of this test is **false**. With quite a few simple Algol compilers, the Boolean value produced by the test is "standardised", i.e. is converted to the internal representation of **true** or **false**. Clearly, if the value is merely to be used for a conditional jump this standardisation is unnecessary. For example, take -1 and 0 as representing **true** and **false** respectively, then

if $i = j$ **then**
could generate

```
LDI,1          I
SBI,1          J
STANDARDIZE
JIZ,1          label       corresponding to false
```

The instruction STANDARDIZE (really a compiler macro) produces -1 in register 1 if it contains a non-zero value and zero otherwise. On the other hand, the standardisation is necessary if the resulting Boolean value is to be used in an assignment. The saving gained by not standardising depends upon the number of types of conditional jumps available. On some computers the contents of a register can be compared with a core location directly. Hence if a simple variable appears as the expression on the right hand side of a relational operator further gains can be made. The right hand expression is usually much less complex than the left one so that left to right evaluation is usually optimal. Programmers almost always write $x > 0.0$ or $p + q > r$ rather than $3.0 < x + y$, etc.

The compiler writer may have a choice as to how the numerical test for equality, etc, is accomplished. If subtraction is used, then overflow could be set in certain circumstances. This is true, of course, even with $=$ and $\neq$. These operators can therefore be handled by logical operations if appropriate ones exist. KDF9, for instance, has special instructions which allow comparison without setting overflow. Advantage is taken of this by one of the ALGOL compilers for KDF9 but not by the other.

Some ALGOL compilers optimise the evaluation of Boolean expressions by not calculating subexpressions which cannot effect the result (Grau 67). So if one has $a \vee b$ then if a is true, then b is not evaluated. This assumes that b does not produce any side-effects, or that side-effects are excluded. Even if b does not involve a function call, it can well have a "side-effect" in that its execution can terminate the program. The gains thus made are usually small because the vast majority of Boolean expressions are very simple.

Certain classes of Boolean expressions often produce poor machine code. Consider the following

if $i = 4 \vee i = 6$ **then**
This would typically generate

```
LDI,l          I
SBI,1          4
STANDARDIZE
STI,l          TEMP1
LDI,1          I
SBI,1          6
STANDARDIZE
ORI,1          TEMP1
JIZ,1          label corresponding to false
```

Not calculating the subexpression $i = 6$ will only be of benefit if i is often equal to 4, which would seem unlikely in the general case. However arranging conditional jumps to avoid the second subexpression would also avoid the standardisation. Clearly it should be possible to leave i in register 1, without subtraction, giving a greater saving.

Type conversion problems arise with relational operators, in that if one of the arithmetic expressions is real and the other integer, the integer one must be converted to real and the comparison made as floating point variables. As before, such type conversion that may be required for constants should be performed by the compiler. Also, special coding may be possible and beneficial in comparisons with zero.

Although it is convenient if a complete set of conditional jump instructions is available (for reals and integers), only a very small overhead is incurred if two conditional jump orders must be combined for certain tests.

Some computers allow one to test for a particular condition, and skip the next instruction if it holds. Standardisation itself can be done with load instructions following the test, and also simple conditions can be handled without standardising by using the same initial code. So this may be marginally more convenient than having conditional jump instructions. Other computers, like the 360/370 series, have condition states set by most arithmetic orders. This gives good implementation of simple relations.

One characteristic of computers designed for ALGOL (B5500 and B6500/ 6700) is that comparison tests give a flag (or Boolean value) as the result rather than a change in the flow of control. This is discussed more fully in chapter 11.

1.5 Conditional Statements and Expressions

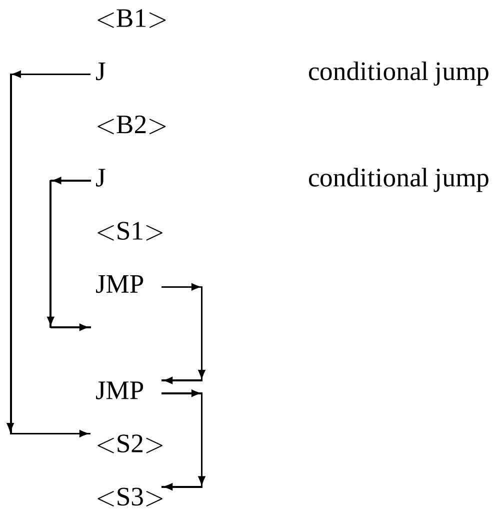
The major part of the code generation for conditional statement and expressions has been covered in 1.4. Conditional statements are somewhat easier in that at the end of a statement the main working registers are undefined, so the Boolean expression can be evaluated without any dumping. On the other hand, register dumping may be necessary with a conditional expression—which will often involve two sub-expressions with a relational operator. Consider
$x/y +$ (**if** $p + q > r + s$ **then** u **else** v)
This is likely to generate

```
      LDA    X
      DVA    Y
      STA    TEMP1
      LDA    P
      ADA    Q
      STA    TEMP2
      LDA    R
      ADA    S
      SBA    TEMP2   Subtraction used
      JAN    L1      Two jumps used since full set of conditional
      JAZ    L1      jumps not available
      LDA    U
      JMP    L2      jump over else
L1:   LDA    V
L2:   ADA    TEMP1
```

With conditional statements, it is not unusual for a **goto** to be written immediately prior to the else, in which case the internal JMP generated by the compiler to go over the else statement can be suppressed. In certain circumstances compilers generate jump instructions which lead directly to a second jump. Consider

```
if b1 then
        begin
        if b2 then
                S1
        else
                S2
        end
else
        S3
```

This would ordinarily give



The first unconditional jump can be replaced by one further down, but the second jump cannot be removed. With fast computers having instruction look-ahead, jumps are often very time consuming, but the optimisation here is probably not worthwhile.

1.6 Assignment Statements

The main difficulty with assignment statements is array variables appearing in the left part list. The left part list should be evaluated first, before the expression. Apart from the possibility of side effects (see 1.9.8), consider the statement

$$i := a[i] := i := i + 1$$

If i is 3 prior to this statement $a[3]$ and i should be assigned the value 4. It is therefore necessary for the compiler to generate the address calculation code for each left part element, storing the result before evaluation of the expression (unless, of course, the compiler can show that the above is not possible).

After evaluation of the expression, assignment to all the left part variables is entirely an internal matter since all the addresses are known. In a large number of cases, the compiler can avoid using temporary storage for the left part addresses—bearing in mind that there is usually only one element. One reasonable approach is to leave the address in a register and dump this register only if it is required for another purpose. With the illustrative machine, register 2 is used for this. So one might have

$$a[i] := b[j] := c[k]$$

generating

```
LDI,2       I
ADI,2       A
STI,2       TEMP1
LDI,2       J
ADI,2       B
LDI,1       K
ADI,1       C
LDA         1,(0)
STA         2,(0)
LDI,2       TEMP1
STA         2,(0)
```

For reasons explained in 1.1 it is convenient if the address is left in a fixed register, in this case register 2. Thus in the case of integer arrays, there is a clash on the use of register 2, resulting in poor code due to use of temporary storage. Consider

$$il[i] := i2[j] + k$$

which would give

```
LDI,2       I
ADI,2       I1
STI,2       TEMP1
LDI,2       J
ADI,2       I2
LDI,1       2,(0)
ADI,1       K
LDI,2       TEMP1
STI,1       2,(0)
```

The last two instructions can be replaced by a single instruction on some machines by using an indirect addressing mode.

With simple variables, it is not necessary to calculate an address so a simple store instruction can be generated after the expression evaluation. With multiple left part lists, the order of the final assignment is immaterial, the reverse order to the text is usually the most convenient, as is seen from the example above. Multiple assignments are very efficient since the result of the expression is in the accumulator (or register). No type conversion is necessary, since all left part variables are of the same type.

This section is concluded with a number of examples.

$$x := a[i] := b[i, j]/y$$

```
LDI,2       I
ADI,2       A           register 2 contains address of a[i]
```

```
LDI,1      I
MLI,1      B+1      calculation using dope vector
ADI,1      J
LDA        1,(0)
DVA        Y
STA        2,(0)
STA        X
```

Performing the address calculation in two different registers gives quite reasonable code in this case

$$i1[i+6] := j := i2[i-3, 4\times k]$$

This would give, using the dope vector technique

```
LDI,2      I
ADI,2      6
ADI,2      I1
LDI,1      I
SBI,l      3
ADI,l      I2
LDI,l      1,(0)
STI,l      TEMP1    Dump because of a complex second expression
LDI,l      4
MLI,l      K
ADI,l      TEMP1
LDI,l      1,(0)
STI,l      J
STI,l      2,(0)
```

An array variable on the left hand side of an assignment statement often appears on the right as well. The vast majority of ALGOL compilers do not optimise this. For instance

$$a[i] := a[i] + 1$$

would give

```
LDI,2      1
ADI,2      A
LDI,l      1
ADI,l      A
LDA        1,(0)
ADA        1.0      compile time type conversion
STA        2,(0)
```

1.7 For Loops

The generality of for loops in ALGOL 60 makes code generation quite difficult. The two major problems are the handling of real expressions with the **step-until** list element, and the handling of multiple for-list elements. Further difficulties arise with allowing array elements as control variables, and consideration of this is delayed until section 7.4.

1.7.1 MULTIPLE FOR LIST ELEMENTS

Consider the ALGOL statement

```
for $i := 5, 8, 13$ do
    controlled statement
```

After the controlled statement is executed with i = 5, the statement must be repeated with i = 8 and then i = 13. The only convenient mechanism for this is to make the controlled statement into a subroutine. So the users' ALGOL is essentially converted into

```
procedure $for\ i$;
    controlled statement;

...

$i := 5; for\ i$;
$i := 8; for\ i$;
$i := 13; for\ i$;
```

However, the compiler writer can generate better machine-code than this conversion into alternative source text would suggest. A temporary variable is allocated to the subroutine return address, which must, of course, be used for nothing else within the controlled statement. Hence most ALGOL compilers would generate the equivalent of the following

```
     LDI,1     5
     STI,1     I
     CALL,3    CS
     LDI,1     8
     STI,1     I
     CALL,3    CS
     LDI,1     13
     STI,1     I
     CALL,3    CS
     JMP       L
CS:  STI,3     TEMP1
     <controlled statement code>
     JPI       TEMP1
L:
```

No compiler known to the author does the comparatively mild optimisation of storing the control variable inside the controlled statement subroutine to conserve space. It is also possible to avoid temporary storage for the subroutine link.

For instance

```
for $i := j, 4, k + 3$ do
    $a[i] := 0.0$
```

should generate

```
        LDI,1       J
        CALL,3      CS
        LDI,1       4
        CALL,3      CS
        LDI,1       K
        ADI,1       3
        CALL,3      CS
        JMP         L
CS:     STI,1       I
                                LD1,1 I here can be omitted
        ADI,1       A
        LDA         0.0
        STA         1,(0)
        JPI         REG3
L:
```

Even this is hardly optimal code, but it does not involve any difficult task such as moving code outside the loop (as could be done with the LDA 0.0). It is important to remember that most inner program loops are extremely simple, so it is likely that the register holding the return address may not be required.

1.7.2 THE STEP-UNTIL ELEMENT

Before considering the general case, the code for single for-list elements is illustrated. The step-until element is the most common so this is given first.

The exact interpretation of the step-until element has been disputed because of some obvious difficulties in the ALGOL 60 text given in the Report (Knuth 1961). It is certainly true that the compiler writer has to make a choice on an interpretation, but having done this the implementation does not cause any difficulty. The disadvantage of the relevant section of the Report is in demanding that the increment is evaluated twice and the control variable three times per loop (except the initial and final times). Also the test depends on the current sign of the increment which makes it difficult to generate efficient code. One reasonable interpretation is that taken for the KDF9 compilers (Randell 1964) which is used here. This is

for $v := a$ **step** b **until** c **do** s

is to be interpreted as

```
        $t1 := v := a$;
        $t2 := b$;
$l1$ :  **if** $sign(t2) \times (t1 - c) > 0$ **then**
            **goto** element exhausted;
        $s$;
        $t2 := b$;
        $t1 := v := v + t2$;
        **goto** $l1$;
```

It is understood that the identifiers $t1, t2$ and $l1$ form no part of the ALGOL program but are used just to define the for-list elements. Two problems arise in code generation, if the code above is followed directly. Code for accessing v and evaluating b can be arbitrarily complex, but must be generated more than once in the source text. Some restrictions are often placed on the control variable to avoid this problem. The statement $t2 := b$ could be made into a subroutine, but in practice b is almost always a constant in which case a large part of the above can be avoided.

Consider the case when b is a positive constant, one then has

```
        $t1 := v := a$;
$l1$ :  **if** $t1 > c$ **then**
                **goto** element exhausted;
        $s$;
        $t1 := v := v + b$;
        **goto** $l1$;
```

With b a negative constant, the $>$ is changed to $<$, so in neither case is $t2$ required. Clearly at label $l1$, the accumulator (or register 1) can be assumed to contain the current value of the control variable, so that $t1$ is in fact redundant. To illustrate this, consider the following

for $i := j$ **step** 3 **until** n **do** s

```
        LDI,1         J
        STI,1         I
Ll:     SBI,I         N
        JIP,I         L2
        <code for s>
        LDI,1         I
        ADI,1         3
        STI,1         I
        JMP           Ll
L2:
```

In fact, the two store instructions can be combined by moving the label L1 up one, but this does not affect the code inside the loop.

The use of a real control variable with a step until element is somewhat dubious as the number of times round the loop may be indeterminate owing to rounding errors, but the code generation is straightforward. It is illustrated with a negative increment

for $x := y$ **step** -6.0 **until** z **do** s

```
        LDA           Y
Ll:     STA           X
        SBA           Z
        JAN           L2
        <code for s>
        LDA             X
        SBA             6.0
        JMP             Ll
L2:
```

Any further improvement to this code would involve deducing there was no assignment to the control variable and limit within the loop, so that the machine loop control instructions could be exploited. Unfortunately quite a large number of compilers do not produce as good a code as is illustrated, the average being about ten instructions in the loop. Note that if the limit is more complex than a simple variable, the code will be substantially worse.

In order to show how bad the for loop control code can be, the illustration below gives code with a complex increment and limit. A subroutine is used for the statement $t2 := b$.

for $i := 1$ **step** $2 \times j$ **until** $20 \times n$ **do** s

```
        LDI,1           1
        STI,1           I
        Sll,1           TEMP1     store t1
        CALL,3          L2
        JMP             Ll
L2:     LDI,1           2         increment subroutine
        MLI,1           J         return address not preserved
        STI,1           TEMP2
        JPI             REG3
L1:     LDI,1           TEMP2
        JIN,1           L4
        JIZ,1           L5
        LDI,1           −1
        JMP             L5
L4:     LDI,1           1
L5:     Sll,1           TEMP2     t2 := sign(t2)
        LDI,1           20        calculate (c − tl)
        MLI,1           N
        SBI,1           TEMP1
        MLI,l           TEMP2
        JIP             L6        goto element exhausted
        <code for s>
        CALL, 3         L2        do t2 := b
        ADI,1           I         add control to increment
        Sll,1           TEMP1
        STI,1           I
        JMP             L1
L6:
```

This illustrates that the for loop control mechanism is one of the weakest features of ALGOL 60.

1.7.3 THE WHILE ELEMENT

This construction is quite straightforward because the equivalent ALGOL 60 coding given in the Report can be implemented without difficulty. This coding is

```
l3 :    v := e;
        if ¬f then goto element exhausted;
        s;
        goto l3;
```

for the element e **while** f. The negation of the Boolean f is ordinarily implemented by inverting the jumps in the condition.

Hence

for $i := i + 1$ **while** $x < y$ **do** s

would give

```
L3:    LDI,1          I
       ADI,1          1
       STI,1          I            i := i + 1
       LDA            X
       SBA            Y
       JAN            L4
       JAZ            L4
       <code for s>
       JMP            L3
L4:
```

1.7.4 THE GENERAL CASE

All the techniques necessary to compile the various forms of the ALGOL for loop have now been given. With more than one for list element, the controlled statement is made into a subroutine, the step-until element may itself use a subroutine, but the while element and the single arithmetic expression presents no difficulty. The case where the control variable is an array variable is considered in 7.4.

To indicate the total system for loop control, one complex example is given

for $i := 1, 2$ **step** $j + 1$ **until** $64 + k, i + 1$ **while** $p > q$ **do** s

Although the controlled statement is made into a subroutine, assignment to the control variable cannot be done within the statement because of the requirements of the step-until element.

```
       LDI,1          1
       STI,1          I
       CALL,3         Ll           statement subroutine
```

```
        LDI,1           2
        STI,1           I
        STI,1           TEMP 1    store $t1$
        CALL,3          L2
        JMP             L3
L2:     LDI,1           J         increment subroutine
        ADI,1           1
        STI,1           TEMP2
        JPI             REG3
L3:     LDI,1           TEMP2
        JIN,1           L4
        JIZ,1           L5
        LDI,1           -1
        JMP             L5
L4:     LDI,1           1         $t2 := -sign(t2)$
L5:     STI,1           TEMP2
        LDI,1           64        calculate $(64 + k + tl)$
        ADI,1           K
        SBI,1           TEMP1
        MLI,1           TEMP2
        JIP             L6        goto next element
        CALL,3          Ll        call statement
        CALL,3          L2        do $t2 := j + 1$
        ADI,1           I         add control
        STI,1 TEMP1
        JMP             L3        jump to test
L6:     LDI,1           I
        ADI,1           1
        STI,1           I         $i := i + 1$
        LDI,1           P
        SBI,1           Q
        JIN,1           L7
        JIZ,1           L7
        CALL,3          L
        JMP L6
Ll:     STI,3           TEMP1     $tl$ not required during the loop, but
        <code for s>              register 3 may well be needed.
        JPI             TEMP1
L7:                               loop exhausted
```

Even in this case, a tacit assumption has been made that register 3 is not required in any of the internal subroutines. This will not be true if a function call or name parameter access is involved. Indeed, the control variable itself can be a name parameter, making the effective handling of the for loop virtually impossible.

1.8 Storage allocation

In this section the problem of storage allocation of the working variables of an ALGOL program is considered. Storage allocation for the program object code, constants, strings and **own** variables is excluded.

It is frequently stated that dynamic storage allocation is an expensive luxury, implying that ALGOL 60 is necessarily an inefficient language. This is basically unsound. The fact is that on most modern computers access to a word of core store with the address modified by a register is no slower than without modification. Thus, provided the necessary pointers can be maintained in registers, dynamic storage allocation need involve only a small overhead.

The basic method of allocation of internal storage can depend upon the hardware and the operating system facilities. At one extreme, some systems demand that the core store allocated should be fixed before program execution (KDF9) whereas on the B6500 all storage is allocated dynamically by the supervisor. With base address and limit address machines, a facility may be available (via the operating system) to extend the single block of core in use. Such a system must be used with care, or an intolerable burden may be placed on the operating system. On the 1900, extra core store is claimed but not returned to the system (until program termination). This has the advantage that numerous supervisor calls are not made, but has the obvious disadvantage in the case of the program requiring large amounts of core store only during the first part of its execution. One version of the ALGOL compiler for the 360 did a supervisor call on every block or procedure entry (or exit) to claim (or release) storage. This imposes far too great an overhead.

The basic storage mechanism is almost always a stack. This allows for the re-use of store of variables no longer in scope and gives access in a natural way to the variables in scope. The management of stack storage is very simple and efficient as no "garbage collection" is involved. A single stack can be used for all the working store of an ALGOL program—and this technique is almost always used for machines which allow only one block of core for internal storage. On the B5500, B65/6700, X8 at Eindhoven and the Manchester University machine (MU5) (Kilburn, 1964), arrays are stored off the stack. This can cause additional problems, although the basic mechanism is the same as for a single stack.

The problem of dynamic storage allocation is therefore one of maintaining pointers in registers so that access to variables can almost always be achieved in one instruction.

1.8.1 BLOCK LEVEL ADDRESSING

On entry to a block, all the identifiers within the scope of that block become accessible. A block is therefore the natural unit of storage allocation and variable access. Every variable within the block can be assigned a unique address, which is relative to some pointer which itself is fixed for the duration of the activation of that block. A block within a recursive procedure may be activated several times by the recursive call of the procedure. At any point, the only variables which can be accessed are those corresponding to the last activation of the block. It is therefore only necessary to maintain pointers corresponding to the last activation of a block within registers.

At one point in the source text, there will be a certain number of blocks nested in the program. Say that this is 3, and call the blocks Bl, B2 and B3. Therefore the only variables which can be accessed will be those of the blocks B1, B2 and B3. If three registers can contain the address of word 0 of the last activation of these blocks, then address modification via these registers gives access to the required variables in one instruction. For instance, if x is variable 6 of B1, y variable 8 of B2 and z variable 20 of B3, and the registers 4,5 and 6 are used for the three blocks one has that

$z := x + y$

is compiled as

```
LDA          4,(6)
ADA          5,(8)
STA          6,(20)
```

The basic requirement of the dynamic storage allocation system has been met in that the variables within scope can be accessed in a single instruction. The only overhead such a system incurs is that of setting up the registers on block entry correctly, and preserving them on procedure call (and name call) throughout the block.

The block entry mechanism is very simple. Let us suppose that Bl and B2 are already active and that B3 is about to be entered. Part of the information about B2 (stored with it) is the amount of storage it requires. Say this is 20 words. Therefore words 5(0) 5(1)...5(19) cannot be used for B3. So 5(20) is the first free word, and therefore register 6 can be assigned a value twenty more than that of register 5. The address of the first free word of core store is sometimes maintained in a register. If this is the case, then it must be updated on both block entry and block exit.

The method of block addressing is quite practical, and is used by a number of compilers. As blocks are not entered very frequently, the overhead of this technique is not high. The main difficulty is that there may be an inadequate number of index registers to support the method. Variables in the outermost block will always occupy the same locations, and so can be assigned fixed storage. Even with this improvement, more index registers may still be needed than are available. On Atlas, a machine with 90 index registers, block level addressing works very well, but for most other machines, some other method is really required.

1.8.2 PROCEDURE LEVEL ADDRESSING

Procedures give rise to a level for addressing even if the body of the procedure is not itself a block. This is necessary for access to the procedure parameters, and to define the scope of labels within the procedure body (Report 5.4.3). Procedures are therefore very similar to blocks, the essential difference being that procedures are activated by a call, which can mean that various blocks become temporarily inaccessible. The method whereby procedures are entered and left is covered in 1.9.

Procedure level addressing works by virtually removing the equivalent of blocks from the object program. Only procedures remain. Consider the procedure p in the program with the following inner block structure.

```
procedure p(a, b)          B1
          ...
          begin
                    real x, y, z    B2
                    ...
          begin
                    real u, v, w    B3
                    ...
```

```
        end
        ...
        begin
                real $r, s, t$          B4
                ...
        end
end
```

To illustrate the technique, only simple variables are considered. Using strict block level addressing, one index register would be required for Bl, another for B2, and one for B3 or B4 if entered. However, the position of variables x, y, z relative to the pointer for Bl can be determined by the compiler. This is because the amount of space required by each parameter can be calculated at compile-time. This clearly means that the additional pointer for B2 can be dispensed with. Similarly, because only simple variables are being considered, the same applies to the blocks B3 and B4. Since the storage required for B3 is returned before the block B4 can be entered, the same storage can be reused. Hence u and v will use the same location as will r and s, etc. (assuming reals and integers both occupy one "word").

Slightly more working storage may be required with procedure addressing because storage will have to be assigned for inner blocks even if they are not actually entered. On the other hand, fewer levels of storage will be set-up resulting in an economy of "link data" (see 1.8.4).

Working variables used by the object program for partially computed results in expression, etc. (TEMP1, TEMP2, etc., in the examples), are usually assigned storage like program variables declared in the innermost block in which they are used. Such variables can be re-used for several different purposes within a block. On the completion of the code for a statement or expression, such temporary variables used can be returned to the pool for re-use. One simple method of achieving this is for the compiler to maintain a stack of temporary variables. The maximum size of the stack is noted, and the stack depth for each variable. If ten working variables are required (TEMP1, ...to TEMP10), then TEMP1 is assigned the address (relative to procedure base pointer) of the first free location (i.e. not used by declared variables). Then the total space for simple variables is ten more than that required for declared variables. The B5500 ALGOL compiler uses a different technique. Although the stack of the machine can usually be used for working variables, sometimes variables are required in the global area or declared stack variable area. A bit pattern is maintained for these areas indicating whether every location is for a declared variable. If not, it can be used for a working variable (and re-used if necessary). A similar method is described by Jensen (1965).

Economy in the use of working variables in the compiled program is not very important. There are several reasons for this. Firstly, it is unlikely that many procedures will be active at one moment. Since with dynamic storage allocation only the active procedures will require storage this is a substantial economy in itself. Secondly the number of references to a temporary storage locatIon is likely to be at least two, and this should be spread over a number of instructions. Hence the storage required for the temporary variables will be small compared with the instruction space. Thus it is usually more important to reduce references to working variables in order to economise in instruction space (and object code speed) than to reduce the need for storage of these variables.

Procedure level addressing gives a very real reduction in the number of registers required for variable accessing. In fact, the majority of programs will work well with only one register. Using fixed locations for the main program variables, the one register can be for the current procedure. If no nested procedures appear in the program text, this one register is sufficient. Even if nested procedures exist, provided within the innermost procedure, access to variables in the outer procedure is not required, then one register is still sufficient. The fact that the B5500 can be so successful with only one register, shows that this is not an important restriction. With conventional machines, one can usually allow two registers for level pointers and make no restriction as is shown in 1.8.4.

Procedures which are known to be non-recursive, can be allocated core store in a similar manner to blocks with procedure level addressing. The advantage of this technique is that the procedure entry and exit mechanism can be simplified so that the coding required would be comparable with FORTRAN. The disadvantage is that procedures must be classified (see Huxtable, 1963) and that essentially different mechanisms may be involved with the call of formal procedures (see 7.2). A description of such a technique is given by Henhapl (1971).

1.8.3 FIRST AND SECOND ORDER WORKING STORE

In every block, storage is required for the simple variables, working variables and various control information. The amount of this can be calculated at compile-time, and is called first order working store. Storage for arrays can not, in general, be calculated until the block is entered and this is called second order working store.

Storage required for arrays is often very large, typically greater than the total required for simple variables, object program, strings, etc. Hence the allocation of storage for arrays must follow strictly the block structure of the program, as no wastage could be tolerated.

Consider the problem of entering a block with block level addressing. Firstly the basic block entry mechanism given in 1.8.1 must be performed, which allocates storage for the first order working variables. Then every bound pair list must be evaluated. After every evaluation, the storage required must be allocated and the addressing information for the array, calculated and stored. To do this, a pointer is used in each block giving the limit of second order working store. This is incremented for each array declaration, but is static during the course of executing the statements of the block. The pointer is initialised to the end of the first order working store on block entry.

The information giving details of the array is usually dependent only on the dimension of the array. Hence storage for all of this can be allocated by the compiler in first order working storage. With the code word technique, the additional addressing information consists of the $(n-1)$ multipliers (see 1.2.3). Usually sufficient information is stored in with the addressing information to allow subscript checking. If the information consists of the bound-pair list, then this can be set-up directly by open code when the bound-pair list is evaluated. With arrays declared in the same segment, some addressing information can be shared. Care must be taken, since it is highly undesirable that the speed of array accessing should be degraded by such sharing of information — it is preferable to have several copies. The storage of array information should be determined by three considerations, fast access to array elements without subscript checking, tolerable performance with checking and ease of handling array parameters (see 1.9.5). The requirements of array declaration are the least important since this is less frequent.

Consider, as an example entering the block

```
begin
        real $x, y, z$;
        array $a, b[1:20], c, d[1:2, 0:n]$;
        ...
        ...
end
```

Before entry assume that the first word of free store is word 500. If register 3 is to be used for this block, then the first action on block entry is to set register 3 to 500, to increment the word giving the first word of free store to allow for x, y, z, etc., and to set up the word giving the limit of first order working store to the same value. This would give (see Figure 1.1).

The array word for a will ordinarily contain the absolute address of $a[0]$ that is 517 rather than its address relative to register 3. In the case of the dynamic arrays c and d, the bound pair lists are evaluated, and then the array declaration subroutine can allocate the correct amount of store. Because of the general strategy adopted for array storage, arrays with dynamic bounds present no more difficulty than those with constant bounds. Care must be taken in placing the evaluated bounds in temporary working store. The simplest solution is to stack the values. That is, each expression is evaluated and placed in the first free word of store, and the address of this location is incremented. The address of the top of the stack can easily be reset by the subroutine (which it must do in any case to allow for the array space).

The pointer giving the end of second order working store appears at first sight to be a redundant copy of the address of the first free word (top of stack). This is so during the array declaration, but the pointer is necessary in order that storage can be correctly administered with **goto**s.

Now reconsider the problem of block entry with procedure level addressing. If the block to be entered has no array declarations, then only first order working store need be allocated. This can be done at compile-time and hence no action need be taken, only blocks containing arrays need be considered (called array blocks).

As before the space for the arrays must be added to the second order working store for the procedure. However on block exit, only the arrays declared in that block must be returned to free storage. This means that a pointer is required for each nested array block within a procedure, which gives the limit of second order working store at that nested depth.

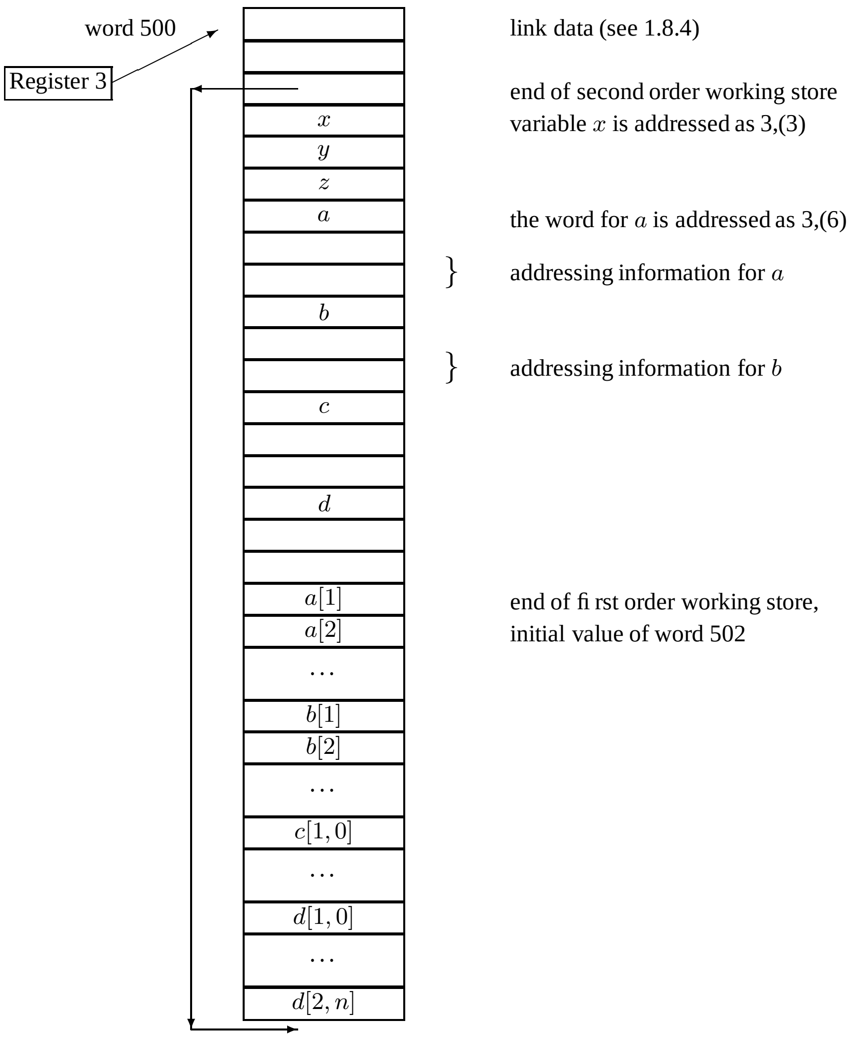

word 500
Register 3
link data (see 1.8.4)
end of second order working store
variable x is addressed as 3,(3)
x
y
z
a
the word for a is addressed as 3,(6)
addressing information for a
b
addressing information for b
c
d
$a[1]$
$a[2]$
end of first order working store,
initial value of word 502
$\cdots$
$b[1]$
$b[2]$
$\cdots$
$c[1,0]$
$\cdots$
$d[1,0]$
$\cdots$
$d[2,n]$

Figure 1.1: Diagram of storage

For example, with the block layout

```
begin
real $x, y, z$; array $a[1:50]$;                              BLOCK1
        <statements requiring three temporary variables
        TEMP1, TEMP2, TEMP3>
        begin
                integer $i, j, k$;                            BLOCK2
                <code requiring TEMP1, TEMP2>
        end
        begin
                array $b[1:40]$;                              BLOCK3
                <code requiring TEMP1, TEMP2, TEMP3, TEMP4>
                begin
                        array $c[1:20]$;                      BLOCK4
                end
        end
end
```

Storage might be assigned as follows (see Figure 1.2).

The variables WB1, WB2 and WB3 give the limits of second order working store for the three nested array blocks. In block 1, four locations at the end of the first order working store are wasted, in order to allow for the storage required by block 3. Note that the maximum requirement for first order working storage does not necessarily occur in the same block as that requiring the largest array storage. Block 2 has no effect on the storage mechanism since its storage requirements are less than that of block 3. If block 2 were an array block, it would use WB2 to give the limit of second order working storage. No code needs to be produced for entry or exit from block 2. On exit from the other blocks, the address of the first free word of core must be reset from the variables WB1 or WB2.

The illustration of the assignment of storage is represented in such a manner to demonstrate the techniques, the relative positions of the variables may be very different. The dashes indicate the array address information.

Now consider in detail the code generation required for the above example. Assume that the base register of this storage is no 4, and that the address of the first word of free storage is given by register 5. If the initial value of register 4 is 500, then register 5 would contain 519. The code for the array declaration might be

```
LDI,1       1         lower bound
STI,1       5,(0)     store in top of stack
ADI,5       1         increment top of stack
LDI,1       50        upper bound, left in register 1
LDI,2       9         address of array information relative to
                      register 4
ADI,2       REG4
CALL,3      MSFI
STI,5       WB1
```

The routine “Make Storage Function 1” (MSF1), deals with the declaration of one dimensional arrays. The array bounds are known from the setting of register 1 and the

BLOCK 1	BLOCK 2	BLOCK 3	BLOCK 4
link data			
WB1			
WB2			
WB3			
x			
y			
z			
a			
a'			
a''			
TEMP1	i	b	
TEMP2	j	b'	
TEMP3	k	b''	
—	TEMP1	TEMP1	c
—	TEMP2	TEMP2	c'
—	—	TEMP3	c''
—	—	TEMP4	—

beginning of second order working store

Figure 1.2: Procedure-level storage layout

last stacked item (word 519). The array must be positioned from words 519 to 568, and this fact must be recorded in the three words a, a' and a'' whose address is given by register 2. On exit from the subroutine, register 5 will have the value 569, and this must be assigned to WB1. If further array declarations occur the process can be repeated.

The make storage function routine tends to be quite complex because of the numerous different cases. Most compilers do not regard one dimensional arrays as a special case, so the array dimension must be given as a parameter. If different storage sizes are used for Boolean, integer or real elements then this also must be parameterised. Although own arrays must be handled quite differently (see 7.9) their declaration may be channelled through the same routine. As speed of array declarations is not critical, the compiler writer should concentrate on making the code size small. One technique is to have separate entries for the frequently used cases so that fewer parameters need to be set-up. This is illustrated above with the one-dimensional array taken as a separate entry (leading of course to the same general routine).

The stacking of the array bound information is necessary because the subsequent bounds could involve arbitrarily complex expressions. Temporary working store is not appropriate since all the bounds must be easily accessible to the make storage function routine.

The variables WB1 and WB2 are needed on exit from the appropriate blocks and also with a **goto**. For instance, on exit from block 4, the position of second order working store in block 3 must be restored, by the single instruction LDI,5 WB2 (where WB2 is of course 4,(4)). The same code needs to be generated on a corresponding **goto**.

1.8.4 LINK DATA AND THE USE OF REGISTERS

In the examples in this section, it has tacitly been assumed that a certain number of registers have been assigned to storage level pointers (at block or procedure level). Provided variables can be accessed in the vast majority of cases in a single instruction, the exact method whereby the exceptional case is handled does not really matter. There are substantial advantages in arranging that few registers are used for variable access (in a dedicated manner). These advantages being that procedure entry and exit can be more rapid if fewer registers need to be set-up (which also applies to the evaluation of call-by-name parameters). Another advantage is that the remaining registers can be used for code optimisation.

With procedure level addressing, it is certainly true that the vast majority of accesses will be at the current level or global. Thus the access to intermediatory levels can be made in a manner to economise in registers rather than produce the most optimal code. One technique is to chain the currently accessible levels via pointers in a fixed position. Backward rather than forward chains are easier to arrange, and so if the current level is 3, and access is required to variable 10 in level 1, one would generate the code

```
LDI,3        4,(1)     register 4 for current level
LDI,3        3,(1)     back one more level
LDA          3,(10)    loads variable into accumulator
```

No subroutine call need be involved at this level of generation, so register 3 can conveniently be used. Word 1 of each level gives the address of the next lowest currently active level.

An alternative method is to maintain in each level the address of the start of every active level. Thus, however deep a level is required, the base address of that level can always be found in one instruction. This obviously simplifies code generation. The method is particularly appropriate if convenient instructions exist on the machine for copying several words from one area to another. This is required on entry to a new level, where the new addressing information is largely formed from the existing level pointers.

Exactly similar techniques can be used with block addressing the only difference being that efficiency is more important. Some compilers use a mixture of both methods. For instance, although block level storage is used in the general case, the parameters and locals to a procedure form one level, and all global procedure level variables are assigned static storage. If both techniques are used many of the advantages of procedure level storage are gained.

At the start of each level, a few locations are used to store details about the level. For instance, the depth of the level, the machine code address of the return from that level (this will usually be the return address of a procedure call) and perhaps some information for diagnostic purposes. Most ALGOL compilers appear to pack such information by using partial word operations. This is quite unnecessary for storage economy since it is very unlikely that more than ten levels will be currently set-up at any moment in a program (with procedure level storage at least). So packing will only save a few words but is likely to have a detrimental effect upon set-up times.

The exact form of the link data will depend upon which procedure entry mechanism is employed. This applies even with block level addressing, since blocks are usually represented as a simple, degenerate form of procedure.

1.9 Procedure and Function Calls

The efficiency with which compilers handle procedure calls varies very widely (see 2.2.6), the main reason being that in its most general form, the handling of parameters is difficult. Unless there is some specialisation, all procedure calls will be very slow because of the difficulty with unspecified parameters, and that of parameters to formal procedures (which can never be specified). These points are considered in more detail later (see 7.2). In this section it is assumed that the compiler has resolved all the problems of the type correspondence of the formal and actual parameters.

There is very little difference between a procedure and function call. This applies not only to the syntax, but also to the machine code that needs to be generated for the call and the body of the text. With a type procedure call, the result delivered by the procedure can conveniently be left in a register. With the illustrative computer, integer or Boolean values would obviously be left in register 1 to conform to the usual conventions. Similarly real procedures will leave the result in the accumulator. Thus there is no difficulty in obtaining the value after the procedure has been left due to the absence of the addressing pointers for that procedure. Also, no special action is necessary if a type procedure is called by a procedure statement (this is a rather dubious feature of ALGOL which some systems omit, but is easy to implement).

In order to separate the various actions of the procedure entry and exit mechanism, the call of parameterless procedures is considered first.

1.9.1 PARAMETERLESS PROCEDURE ENTRY AND EXIT

The basic requirement is that registers and links must be set-up appropriately so that further procedure calls may be made as well as allowing access to all the variables in scope. Of course, the entry must be performed in such a way as to allow the return to be implemented efficiently.

The term level will be used to mean either procedure or block level according to the addressing strategy adopted. All examples will be in terms of procedure level addressing since it has already been shown that this is greatly superior in most cases.

The problem of procedure entry and exit is to set up the level pointers so that variables can be accessed correctly. If global variables are accessed using fixed locations, then no action need be taken on their account. Hence the problem lies with the access to local variables within the procedure and non-local, non-global variables.

In general, at the point of call, k levels will be active, but in the procedure itself there will be l with $l \leq k$. This is illustrated in Fig. 1.9.1 with $k = 3$ and $l = 2$. One can see that $l - 1$ levels will be shared between the position of call and the position of the procedure. The procedure body itself will therefore invoke one new level, and make inaccessible $k - l + 1$ levels. In the diagram, the two levels C and D become inaccessible during the procedure call, while a new level B is established.

The general mechanism would therefore demand that the addressing details for the old temporarily inaccessible levels are dumped and the new level established. If a display is kept in registers, then the registers corresponding to C and D in the figure are dumped on procedure entry and restored on procedure exit. Many compilers dump and restore all registers on procedure entry and exit resulting in a considerable inefficiency. This is sometimes militated by the use of a subroutine for procedure entry and exit, as it can often happen that it is quicker to dump and restore all registers rather than have additional parameters determining which registers to dump. Some computers have no

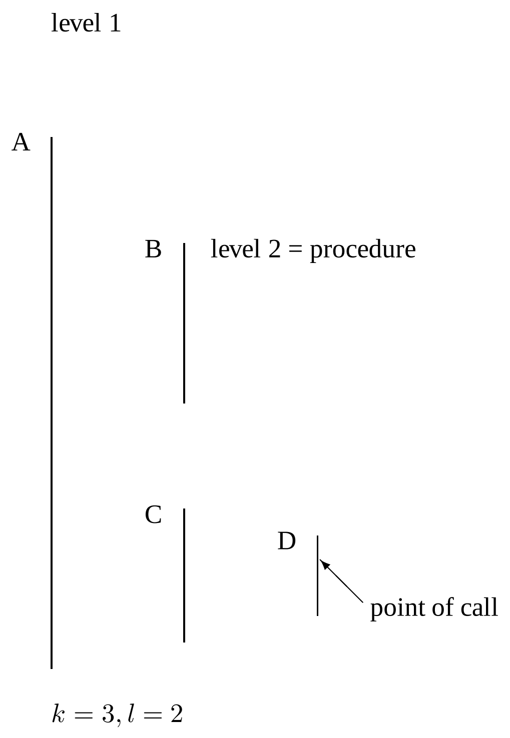


Figure 1.3: Procedure call

means of referencing the ith register, where i is data dependent, which will obviously make register dumping more awkward.

Apart from register dumping and resetting, the establishment of a new level means that additional storage is required. If this is to be checked on procedure entry, then a subroutine will most certainly be used. This is also a convenient place to add diagnostic and tracing code (see chapter 5). The additional storage is obtained from the stack by incrementing the stack pointer by the amount of first order working storage required by the procedure. So the base of addressing for the new procedure will be the top of the stack prior to the call.

Now consider the example illustrated in Fig. 1.9.1. Following the system given in 1.8.4, register 4 is used for the current level, and other levels are accessed by backward chaining. The first four words of each level are fixed and are as follows

word 0:	machine code link for the return
word 1:	chain word giving address of next lowest level
word 2:	previous value of register 4
word 3:	depth of current level

Assume also that the main program calls A, which calls C, which calls D before calling B. Assume that the storage requirements for the main program and the levels A, B, C and D are 100, 10, 20, 30 and 40 words (including the four fixed words). Then immediately prior to the call of B, one has

Address	Value	
0	0	Set up for consistency, not all required.
1	0	During the execution of the main program
2	0	code, the top of the stack will be word 100
3	0	(i.e. register 5 = 100)
100		address call of A
101	0	During the execution of A,
102	0	
103	1	register 5 = 110
110	address	(in A) of call of C
111	100	(A is the next lowest level)
112	100	(last level was also A
113	2	
		During the execution of C,
		2 register 5 is 140
140	address	(in C) of call of D
141	110	(C is next lowest level)
142	110	(C was last level)
143	3	
		During the execution of D,
		2 register 5 is 180

The new level must be set up as follows

180	address	(in D) of call of B	
181	100	(A is next lowest level)	During the
182	140	(D is last level)	execution of B,
183	2		register 5 is 200

A parameterless procedure can be called by a single CALL instruction, and the new level can be easily set up with open code in the procedure head as follows

```
PROC: STI,3        5,(0)    Store machine address
      STI,4        5,(2)    Store last value of register 4
      LDI,3        2
      STI,3        5,(3)    Store current level number
      LDI,3        4,(3)    Old level number
      SBI,3        2        Subtract current level number
      JIN,3        L2       Jump for deeper level
L1:   LDI,4        4,(1)    This is the number of chains to go
      JIZ,3        L2       through (> 0)
      SBI,3        I        Count down
      JMP          L1
L2:   STI,4        5,(1)
      LDI,4        REG5     New register 4 is old top of stack
      ADI,5        20       Increment stack pointer
```

Only the setting-up of the pointer to the last level causes any difficulty, as the action required depends upon the depth before the call. This illustrates the necessity of storing the current depth with the fixed information and why backward pointers for level addressing are used. In fact, the instructions in the loop will rarely be executed, and better code can clearly be produced when the current level is zero (since word 1 need not be set).

In contrast procedure exit is very simple. The previous values of registers 4 and 5 must be restored, and the jump made, i.e.

```
      LDI,5        REG4
      LDI,4        5,(2)
      JPI          5,(0)
```

Note that procedure exit does not use register 1 or the accumulator so the same code can be used for return from a function.

In practice almost all procedures are declared at main program level. Hence the last level address word (word 1) need never be set, reducing the number of instructions on procedure entry by 7 to 6. This means that the execution of a dummy parameterless procedure at main program level involves 10 instructions. This is about half of the typical value for most ALGOL compilers. The reason why most generated code has not been as successful as the above is the use of block level addressing and injudicious design of the link data format (for instance, by packing the information).

1.9.2 THUNKS

Thunks is the name given to the most general technique for handling procedure parameters (Ingerman, 1961). In some cases this general technique is necessary, but with most parameters it can and should be avoided. The method is to construct a subroutine out of the expression which forms the actual parameter. The procedure is then given the address of the subroutine. Whenever the actual parameter needs to be evaluated, the subroutine is called. This mechanism itself is not that inefficient, but to evaluate the subroutine correctly, the environment of the actual parameter must be established, and at the end of the subroutine the environment of the procedure must be restored. With the majority of systems, this environment control is very time consuming.

Special action can be taken, in the case when a parameter is a single identifier or constant, to avoid the construction of a thunk. A method of this type, but involving run-time parameter type checking is given in detail by Randell (1964). In fact, the scheme given does not spend any significant amount of time checking parameters, since the main overhead is in dealing with the parameters as such—they are not given to the procedure in the appropriate form but each one must be evaluated and stored in the working store of the procedure. In most cases Randell and Russell could have checked parameters at compile time, because when the address of the procedure is inserted in the interpretive code (in the DECL operation p. 378), the PARAMETER operations can be checked against the CHECK operations both of which have already been generated. This cannot be done with formal parameters.

The thunk mechanism must be used in the case where the former parameter is an integer, real or Boolean variable not specified by value. This is because of Jensen's device, where the effect of replacing the formal parameter by the text of the actual parameter must be achieved (Report 4.7.3.2). A compiler could avoid the thunk mechanism if the actual parameter is always "simple". This is considered further in 1.10.2. Because of the environment control problem, it is clearly best if procedure parameters can be evaluated before the procedure is entered.

1.9.3 VALUE PARAMETERS

Consider a procedure which has a single real value parameter such as a standard function. Within the body of the procedure, this parameter can be accessed in the same way as a local real variable. Its position relative to the current environment pointer (register 4) is fixed. Hence in the code generated just before the call of the procedure, the parameter can be evaluated and placed in exactly the right position for the called procedure. With several value parameters, this can be done with each one in turn before the actual call.

The expression forming the actual parameter can itself involve a function-call (even the same function), and so a general stacking scheme must be adopted. This is illustrated below

$proc(x, y + z, u - v)$ would generate

```
ADI,5       4        This makes space for the link data as given in 1.9.1
LDA         X
STA         5,(0)    Evaluate and stack the first parameter
ADI,5       1
LDA         Y
ADA         Z
STA         5,(0)    Second parameter
ADI,5       1
LDA         U
SBA         V
STA         5,(0)    Third parameter
ADI,5       1
CALL,3      PROC
```

With a procedure having parameters, the parameters are placed immediately after the link data in text order. On entry to the procedure, the stack pointer is just above the last parameter. It must therefore be moved back by (parameter space + 4) words before executing the code given in 1.9.1. The procedure which is called need take no other action for the parameters.

This technique is extremely simple but it is surprising how few compilers adopt it. Open code can be used for the call and the procedure body without excessive demands on instruction space.

As an example of recursive call consider
$sqrt(x + sqrt(z) \times y)$
This would give using this mechanism

```
ADI,5       4         Make space for outer call
LDA         X
STA         TEMP1
ADI,5       4         Make space for inner call
LDA         Z
STA         5,(0)
ADI,5       1
CALL,3      SQRT      Leaves result in accumulator
MLA         Y
ADA         TEMP1
STA         5,(0)
ADI,5       1
CALL,3      SQRT
```

Further improvements can be made upon this code. For instance the last parameter can be left in a register and not stacked before the CALL, the actual stacking being done in the procedure body. This saves instruction space in quite a worthwhile manner since the average number of parameters to a procedure is quite small (actually 2, see 2.3.1.6). If a procedure has only one parameter, then it is unnecessary to make space for the link data before the parameter is evaluated—in fact even with more than one parameter the instructions to make space for the link data can be combined with the stacking of the first parameter. Many computers have some automatic incrementing facility which can be used with advantage in stacking the parameters.

To illustrate all these advantages, consider the compilation of
$proc(sqrt(x), y + z, u - v)$
which would give

```
LDA         X
CALL,3      SQRT
STA         5,(4)
ADI,5       5
LDA         Y
ADA         Z
STA         5,(0)
ADI,5       1
LDA         U
SBA         V
CALL,3      PROC
```

This saves six instructions (on the call only) compared with the straightforward coding. It also means that on the call of $sqrt$, the base of storage will be four words lower.

With the assumption that the compiler knows the type of each formal parameter, the type changing that may be required can easily be incorporated in the evaluation of each parameter. For instance $sqrt(i + j)$ would generate (with the optimisation given above).

```
LDI,1       I
ADI,1       J
FLOAT
CALL,3      SQRT
```

Several compilers deal with value parameters in the manner indicated except for the specialisation with the first and last parameter.

1.9.4 STRING PARAMETERS

As stated in 1.3, the compiler writer has considerable freedom in the manner in which strings are handled. Since all strings are presumably to be used for the input/output routines it is advantageous to store them in a form convenient for these routines. For instance the enclosing string quotes can be replaced by an integer count, and the string represented as internal characters in a manner appropriate to any character handling instructions that may exist. Strings can also be validated as appropriate for the input-output routines, and additional information given about the string. This technique is used with the CDC 6000 series ALGOL compiler, where the very elaborate Knuth input/output routines have been implemented (see 12.6 and Knuth, 1964a). Hence the analysis of complex format control strings is performed only once by the compiler instead of at every reference to the string (as with KDF9). To conform strictly to the Report, strings which are not valid for the input/output system must still be processed.

However strings are stored internally, the total space they occupy may be very large. Since no alteration can be made to the string (except in code bodies), the easiest method of dealing with string parameters is to pass the address of the string to the procedure. If this formal string is again passed as a parameter, then the address can be copied using the ordinary value parameter mechanism.

For instance, a procedure process might have a string parameter and be called thus

$process(i,$‘abc’$,j)$

The internal address of the string is denoted by identifier STRI, so this would generate

```
LDI,1       I
STI,1       5,(4)
ADI,5       5
LDI,1       STRI
STI,1       5,(0)
ADI,5       1
LDI,1       J
CALL,3      PROCESS
```

If the procedure process then did a further call

$output(device, s)$

where s is the formal string parameter, one would have

```
LDI,1        DEVICE
STI,1        5,(4)
ADI,5        5
LDI,1        S          (S is 4,(5))
CALL,3       OUTPUT
```

Hence such parameter passing causes no difficulty.

1.9.5 ARRAY PARAMETERS

As explained in 1.2.4, information concerning an array is usually contained in up to three parts. First, a fixed amount of storage giving brief details of the array—for instance the address of $a[0, 0, \ldots 0]$. Secondly there is the dope vector or its equivalent. The amount of storage for this part will usually depend upon the array dimensions. Lastly the array storage itself, contained in the second order working store of the level in which it was declared.

The information passed by the parameter mechanism must be adequate to allow rapid access to the array elements. Clearly at least the first part of the array information must be passed. It is not usually convenient to pass any more than this for two reasons. Firstly copying over any substantial number of words is undesirable. Secondly, the dimensions of the actual array may be unknown (see 7.1), and so the quantity of information to be passed is unknown. If for rapid access to array elements the second part of the array information must be directly addressable, then the procedure itself can copy this over after it has been called. There need be no copying if the procedure does not access an array element.

With most conventional machines, the basic fixed information for an array is about 3 or 4 address lengths. This could be stored in successive locations, and so the passing of an array parameter consists of copying these words on to the stack. This can sometimes be managed by a MOVE instruction (as on a 360) or a block transfer instruction. Since the illustrative machine has no such instructions, quite a few operations are necessary, viz.

```
LDI,1        A          Array information assumed to be three words
STI,l        5,(0)
LDI,1        A+1
STI,l        5,(1)
LDI,l        A+2
STI,l        5,(2)
ADI,5        3          increment stack
```

With arrays called by name, merely copying over the array accessing information is all that is required. Inside the procedure, elements of the array can be accessed just as rapidly as if the array were local. Thus arrays called by name are easy to compile and efficient at run-time.

Arrays called by value cause much more difficulty. The code generated for such array parameters can be identical with the above, but special action needs to be taken when the procedure is entered (see 7.3).

1.9.6 LABEL PARAMETERS

If a formal parameter is a label, then the actual parameter can be any designational expression. If the parameter is by value, then the corresponding designational expression must be evaluated to obtain the necessary label. The evaluation is similar to that of other value parameters as given in 1.9.3 (see also 7.7). If the label parameter is by name, then the evaluation of the designational expression must be delayed until a **goto** is required to the parameter.

The difficulty with label parameters is that a general label in an ALGOL program is quite complex. It contains three parts: firstly a machine-code address (as in assembly code), secondly details of the environment to allow access to variables to continue correctly, and lastly the address of the top of stack which must, in general, be reset. With the access method given in section 1.8, this would mean a machine code address, plus the values of registers 4 and 5. This information can be packed, since unlike arrays, access to this is unlikely to be critical. Care must be taken in constructing the values of registers 4 and 5, since they will not, in general, be the same as the current values on the procedure call (see 1.11).

1.9.7 STANDARD FUNCTIONS

The ten standard functions defined in the Report (3.2.4 and 3.2.5), can be regarded as type procedures with a real value parameter. This is not stated explicitly, so presumably the compiler writer can implement them by any appropriate means if he so wishes. The report states that they have “reserved identifiers” so it may not necessarily be valid for the user to redefine their meaning.

The compiler writer certainly has a choice in the manner in which the standard functions are called—and indeed if they are called at all. Several systems generate open code for $abs, sign$ and $entier$ which not unnaturally has an appreciable effect on the time these functions take (see 2.2.6). The majority of systems appear to use a different method to call standard functions than that of user-written procedures. In some cases the calling method adopted is that of the independent compilation system (for instance Burns, 1966), which allows the user to redefine a function by providing his own independently compiled version of ln, etc. One use of such a facility would be to provide a faster less accurate version for a particular application. In general, there seems little substantial use for redefinition, so that the compiler writer can reasonably take whatever choice seems most appropriate in the circumstances. Open code for abs is particularly worthwhile owing to its frequent use (see 2.3.1.7).

Apart from the standard functions defined in the Report, the compiler writer must handle the calls of input/output routines and other pre-compiled items from a library. The conventions adopted for this will usually depend upon the operating system, and can vary substantially. The basic input-output routines can be called very frequently so it is important that the calling mechanism itself should not be too time consuming.

1.9.8 SIDE-EFFECTS

A function has a side-effect if it produces some other effect upon its environment apart from delivering a value. In ALGOL a function can have a side-effect by one of the following means:

(a) Use of own variables

(b) Use of non-local variables (including assignment to parameters)

(c) Failing

(d) Call of a procedure or function which does (a) (b) or (c).

Much discussion has centred round whether side-effects should be allowed, and if they are not allowed how they should be stopped (Knuth, 1967). As far as a user is concerned, it seems natural to write a random number generator or real number input routine as a function. Such functions, whilst they deliver a value, have very important side-effects. Hence in the author's view, removing side-effects from ALGOL seems unnecessary and unnatural.

The difficulty with side effects is that the order in which every part of an expression is evaluated becomes "public knowledge" and hence cannot reasonably be subject to change. In all cases in which an expression can be placed, a function call can be substituted which say, prints out its own name. This reveals to the programmer the order in which the expressions were evaluated. As an example, the expressions involved in subscripts on the left hand side of an assignment statement, the expressions which are parameters to procedures, etc., must all be evaluated in a given order. The alternative is not to state the order of evaluation (even though it can be found out), and say that the programmer should make no use of the order of evaluation. In other words, this remains deliberately undefined in a similar manner to the properties of integer and real variables.

In fact, failure is a side-effect which it is impossible to stop. A failure may in itself reveal the order of evaluation which would be otherwise unknown. For example, on KDF9 two compilers exist which are 'compatible' to a very large extent. Both have a retroactive trace facility (see 5.4.2) giving the last few procedures executed. Now consider $ln(abs(0.0))$. The Whetstone compiler always uses the thunk mechanism for parameters, and hence enters ln first, and then abs in order to evaluate the parameter of ln. Hence the retroactive trace is ln, abs, failure. The Kidsgrove compiler however evaluates value parameters before the call and therefore its retroactive trace is abs, ln, failure.

It has been stated in 1.1.2 and elsewhere, that there are sometimes advantages in not evaluating expressions in a strict left to right manner. The saving is of the order of two instructions required to store and fetch a temporary variable. This saving is clearly insignificant compared with a function call, so the effect of strict left to right evaluation can always be achieved by storing temporary results when functions calls are involved. On the other hand, it is clearly inconvenient for a compiler to generate strict left to right code when function calls are involved, and more optimised code when they are not. In practice, compilers adopt either a strict left to right generation, or a more optimised one which will therefore produce different results when side-effects are used.

1.10 Name Call

Within the body of a procedure, references to call-by-name parameters must have the effect achieved by a textual substitution of the actual parameter for the formal. Since textual substitution is meaningless in the compiled program, an equivalent mechanism must be used which is the "thunk" described in 1.9.2. The evaluation of the actual parameter via the thunk method is usually described as a "name call".

In the case where an expression is to be evaluated via the name call, the mechanism is very similar to that of a parameterless function call. The main difference being that instead of establishing a new environment for the name call, a return is made to the environment of the actual parameter. With most systems, the environment control overhead involved in this is enormous compared with that of the basic thunk mechanism of calling a subroutine. For this reason, some special coding is often used when the actual parameter is a simple variable (or constant).

Name call involves just the same problems as regards side effects as in procedures which was discussed in 1.9.8. and is exploited by Jensen's device (Dijkstra, 1962). When a name parameter is an expression, the actual expression evaluated could have side effects by virtue of a function call.

1.10.1 THE GENERAL CASE

In order to illustrate the methods used, both the detail of the coding of the thunk and the name call must be considered. The procedure must have access to two items, the environment to be established, and the address of the thunk. The name call must then preserve its own environment, set-up the new one, and call the thunk subroutine. The reverse process takes place at the end of the thunk.

With the conventions adopted for the illustrative machine, the current environment is given by register 4. So the value of register 4 and the address of the subroutine must be passed by the parameter mechanism. If the actual parameter is of the form $y + z$, then the code to stack the parameters will be as follows

```
        STI,4       5,(0)
        LDI,1       (L)       load address of thunk
        STI,1       5,(1)
        ADI,5       2         increment stack
        JMP         L2        jump over thunk
L:      <thunk>
L2:
```

Almost all compilers generate the thunks in-line, which means that a jump must be generated to go over these. In fact, this can represent a significant volume of code (see Chapter 9).

For the moment, assume that the thunk leaves the value of $y + z$ in the accumulator. The return mechanism is to restore the machine code address and the environment from the stack.

```
L:      LDA         Y
        ADA         Z
        SBI,5       2         decrement stack
        LDI,4       5,(1)     restore environment
        JPI         5,(0)     return
```

The name call is now fairly straightforward, assuming that the two words assigned to the corresponding parameter are P and P + 1.

```
        STI,4       5,(1)     preserve environment
        LDI,3       (R)       return address
        STI,3       5,(0)
        ADI,5       2
        LDI,3       P+1       REG3 = address of thunk
        LDI,4       P
        JPI         REG3
```

This code is substantially better than most compilers produce because of the more cumbersome environment control adopted. Hence frequent use is made of subroutines for name call and exit from a thunk. Because of the possibility of recursive name calls, the stack must be used for link data. Hence if the address of the top of the stack is not in a register, name calls cannot be very fast.

Note that if the actual parameter is itself a name parameter, then the parameter details can merely be copied over for the inner call.

1.10.2 OPTIMISATION WITH SIMPLE VARIABLES

With the code given in the last section, access to a simple variable via the name parameter mechanism would require 13 instructions. In fact, some of the complications of call-by-name are not catered for in that coding so the most general case would amount to a few more instructions. Hence the overhead in accessing a name parameter is inordinately more than a simple variable, and some optimisation is therefore desirable.

When the actual parameter is a simple or subscripted variable, then the formal parameter can appear on the left hand side of an assignment statement. In this case, the name call mechanism must evaluate the address of the actual variable. On the other hand, an attempt to assign to an actual parameter which is not a simple or subscripted variable must fail. This failure can be managed in several ways. Firstly, the name call mechanism can do a dynamic check. Secondly the compiler can do a check provided some restrictions are made. For instance, a name parameter can be divided into two cases according to whether assignment occurs or not. If assignment can occur, then every actual parameter must be a simple or subscripted variable. The check is not easy to do properly in the compiler because the single assignment to a name parameter may be only through a further call-by-name parameter. So in practice, a call-by-name argument may be regarded as an assignment to the argument for the test. This then avoids a repetitive checking process.

With a dynamic check on assignment, the thunk mechanism could deliver the "address" of the required parameter. This "address" contains a real machine address plus a marker to say if assignment is allowed. Often a sign bit can conveniently be used for such a marker. If "store negative", "load magnitude" instructions are available, these checks can be performed in two or three additional instructions.

If it is known that only valid assignments can be made and that the actual parameter is always a declared variable, value parameter or constant, then the compiler can use "call by address" instead of call by name. The author knows of no compiler which actually does this, although some compilers (Bayer, 1967) certainly do some checking of name parameter assignment at compile time.

An alternative and very common mechanism for optimising name calls is to pass in the parameter information a marker which states whether the parameter is "simple". This marker can have a number of different values interpreted as follows

(a) constant:

(b) simple variable:

(c) address thunk:

(d) value thunk: rest of as for (c) except assignment not allowed, so thunk gives value of expression.

The name call mechanism inspects the marker, and then takes action depending on its value. Assignment is not allowed in the case of (a) and (d). The method gives plenty of scope for clever coding so that the two simple cases (a) and (b) can be handled efficiently, perhaps by open code. The marker can be set up fairly quickly so that the compiler can determine which case is involved.

The Kidsgrove ALGOL compiler has a very unusual method of dealing with the assignment checking problem. For each actual parameter, two thunks are generated, one giving the value, the other the address. This removes the need for a dynamic check since the compiler can always determine the appropriate thunk for evaluation. The address thunk consists of a call of an error routine when assignment is not allowed. The substantial disadvantage is the amount of code produced for the call of procedures with name parameters.

A further method of optimising simple name call, makes the assumption that machine instructions can be placed in the data area. If this is allowed, then the stack position corresponding to the parameter contains load and store instructions in the simple case, and a subroutine call in the general case. This technique is not very "clean" since the distinction between code and data is not obvious, but it can be made very efficient. It was used by Naur in the DASK compiler (Jensen, 1961).

1.10.3 REAL-INTEGER TYPE CONVERSION

One further difficulty with the implementation of call by name is that of type conversion. It is possible that the actual parameter corresponding to a real name parameter may be either integer or real. Hence the type conversion required is dynamic and cannot be deduced from the source text. Because of the obvious difficulties involved, it is not an uncommon restriction, that in the case of assignment, the types must agree. With this restriction, an integer will be regarded as a general real expression when it is an actual parameter to a formal real variable. Hence the FLOAT instruction can be generated at the end of the thunk coding. This restriction seems very reasonable, since in most practical cases the programmer can be expected to make the parameters agree in type to avoid unnecessary checking.

If this restriction is not acceptable, then the marker given in the last section must be extended to cater for the type conversion. Also, on assignment, the marker must be inspected further to perform the rounding or floating. For instance, consider

$rp := a + b$

where rp is a real name parameter. The actions taken must be (1) evaluate thunk (checking assignment is all right), (2) evaluate expression, (3) FIX if actual parameter is integer, (4) store result in address given by the thunk. This clearly differs from the case when the evaluation of the thunk merely produces an address, which (after checking) can be used directly for storing the result. This process without dynamic checking is identical to that of storing of subscripted variables.

1.11 Switches and Gotos

Some of the difficulties of handling explicit jumps have already been mentioned in connection with storage management. The basic problem is to restore the correct environment. In this situation, the environment is given by two quantities, the local environment pointer (or display, if one is maintained), and the limit of second order working store for that environment. The working store limit will also represent the top of the stack, since it is not possible to have any half-completed activity requiring stacked storage at the point of the label.

1.11.1 LABELS

A label is really described by three parameters, the machine code address, the environment pointer, and the top of stack. If block level addressing is used, the most convenient representation to pass is the address of an environment pointer which will contain a copy of the current display and the address of the top of the stack. Hence only two dynamic variables need be used.

With procedure level addressing, the address of the top of stack is held in a word in the current environment whose displacement relative to the environment pointer depends upon the nested array block depth of the label (see 1.8.3). This displacement is static, and can be combined with the machine code address (which is also static) to give two parts to the description of the label.

The general label mechanism is quite complex, so it is worthwhile to optimise simple gotos. For instance, a goto within a block can be compiled as a single jump. With procedure level addressing, a jump within the current environment can be made in two instructions (at least in the illustrative machine), by resetting the top of stack and then executing the jump. All other jumps can use the general mechanism. This mechanism consists of resetting the current environment pointer and the top of stack from the fixed displacement from it, before executing the appropriate jump, viz.

```
LDI,2      CEP        load required environment pointer
LDI,1      DISP
LDI,5      2,(REG1)   top of stack set
LDI,3      ADDRESS
LDI,4      REG2
JPI        REG3
```

Registers 3 and 4 may be needed to address CEP, DISP and ADDRESS, hence the necessity to use register 1. With the case of label parameters, the variables CEP, DISP and ADDRESS will be passed by the parameter mechanism (1.9.6), but with explicit labels, CEP can be accessed as for variables at the same level, but DISP and ADDRESS will be constants which the compiler can insert.

1.11.2 DESIGNATIONAL EXPRESSIONS

Designational expressions appearing as statements (preceded by a goto) are straightforward. They can be compiled in the same manner as if the statement was written as a conditional statement. That is

goto if $a > b$ **then** l **else** $s[6]$

is compiled just like

if $a > b$ **then goto** l **else goto** $s[6]$

Although in the latter case the compiler may well generate an additional jump to go over the **else**.

A difficulty arises with a designational parameter as a procedure parameter or as an element of a switch list. In the former case, if the label is by value, then the designational expression must be evaluated (although one could argue otherwise from the

Report) and the resulting label presented as the parameter. Hence the action on evaluation is not to execute a goto but to produce the details of the label. For this reason, compiler writers are tempted to do gotos in two halves, evaluate label, then the actual jump. This would prohibit the optimisation given in 1.11.1. This two stage approach is used by Randell (1964).

1.11.3 SWITCHES

Two major complications arise with switches. Firstly the element of a switch can be any designational expression not merely a label. Secondly, switches can be passed as procedure parameters which means that a standard mechanism must be adopted.

No genuine user program known to the author has used anything but a label as a switch element. Hence a restriction to this effect would hardly be significant. Nevertheless, surprisingly many compilers do deal with the general case. When all the elements are labels, the switch is not used formally, and no switch element appears as a value parameter, then it is not necessary to evaluate a label of the switch separately from a goto. Under these conditions good code can be produced. A simple jump table, with bound checking will achieve all that is necessary. With labels that require more than a single jump instruction, a double jump can be made, so that a constant space is used in the jump table.

The check that the use of a switch cannot lead to a jump into a block can be made by the compiler and so does not effect code generation.

When one considers the work needed to compile $s[i]$ where s is a formal switch and $s[i]$ is a designational expression parameter called by value, it becomes clear that switches and labels are one of the weakest features of ALGOL. Perhaps this is why so many experts have explored programming without the use of labels (Wulf, 1971).

A further complication arises if an attempt is made to implement 4.3.5 of the Report. This is a very dubious feature of ALGOL, so many systems regard an out of range switch index as an error. In any case, the side effects which may have been caused by the evaluation of the subscript cannot be removed. It also means that with

goto if $a > b$ **then** $s[i]$ **else** l

a jump over the **else** must be generated.

Chapter 2

Performance Measurement

2.1 General Techniques

In this chapter the performance of the compiled code generated by a large sample of existing compilers is considered. In the first instance, this analysis merely takes the basic constituents of ALGOL, and arrives at a performance measurement based upon timing data. This does not take into account the relative frequency of use of the different operations in ALGOL, i.e. expression evaluation, subscript calculation, procedure calls, etc. Hence this measure cannot be regarded as a yardstick for performance in the ordinary sense and its main use is to pinpoint the strength and weakness of each compiling system (see 2.2.6).

One frequently used method of arriving at compiler performance is by means of "benchmarks". These are specially chosen programs whose execution speed is regarded as a figure of merit. The difficulty with this approach is that quite different relative speeds are often obtained with different benchmarks because radically opposed methods are sometimes used to compile various features of ALGOL, and thus the number of instructions executed can vary by a factor of ten or more. For instance, on one machine, procedure calls can be relatively slow, so that the execution rate of a benchmark would depend critically on the number of procedure calls involved. Most benchmarks are ordinary programs with a variety of statement types so that the reason for a poor (or good) performance is not usually apparent. To understand why a benchmark performs as it does, it is necessary to time its constituent parts by software or hardware monitors (see 6.3). An alternative approach of taking simple statements for timing is therefore much more productive.

To progress any further with performance measurement it is necessary to have accurate statistical information on program behaviour. Two sources of such information are used in this chapter, the thesis of D. S. Akka (1967) and a report of the author (Wichmann, 1970b). Such statistics fall into two classes. Static analysis, where the program source text or compiled output is considered without its execution, or dynamic analysis where the program loop structure makes itself felt. Dynamic statistics are usually more difficult to obtain since either interpretive execution, compiler modification, or hardware monitoring is required—all of which have attendant problems. On the other hand static analysis of source text can be done very easily by program.

2.2 Statement Times

The basic technique is to use the time taken to execute simple statements as a measure of computer performance. This has the substantial advantage over traditional measure of computer performance in that some of the software used by the intending programmer is taken into account, as is the effect that computer architecture may have upon the generation of machine code. On the other hand, because the times depend upon software, they are subject to change—in some cases by as much as a factor of two. ALGOL 60 makes an excellent vehicle for comparative work since the language is well defined and an implementation of it is available on most medium to large scientific computers.

2.2.1 TIMING THE SIMPLE STATEMENTS

The execution time for most of the statements considered is a few microseconds on third generation machines. Hence such a statement must be repeated many times if an accurate figure is to be obtained. One method is to place the statement in a **for** loop, provided the length of time to execute the for loop control code is subtracted. Assuming the existence of a **real procedure** *cpu time* giving the processor time in seconds the required program would be

```
$p := cpu\ time;$
for $i := 1$ step $1$ until $n$ do;
$q := cpu\ time;$
$c := q - p;$
$p := cpu\ time;$
for $i := 1$ step $1$ until $n$ do $< s >;$
$q := cpu\ time;$
$t := (q - p - c)/n;$
```

On machines without a clock, a stop watch can be used instead but this is not very satisfactory.

This method of timing assumes that the for loop control code is independent of the statement s. In practice, this has been found to be the case with the statements under test, although it is certainly not true in general (owing to some optimisation; see Grau, 1967). The value of n must be chosen to be sufficiently large so that any rounding error involved in reading the clock is small. Too large a value of n can lead to a substantial waste in computer time. With a clock giving only times in seconds, an hour or more is required to time the 41 statements involved. On the other hand, KDF9 has a 32μs clock and all the statements have been successfully timed in 20 seconds even with a multi-access system. If a low resolution clock is available, but one which is known to be accurate, a programming technique can be used to produce a finer resolution timer. This method is due to A. Sutcliffe of ICL. Assuming the clock ticks once every second, then a program loop is constructed which repeatedly reads the clock. If one thousand requests can be made between clock ticks, then this loop can be used to give a one millisecond clock. To measure any time, the integral number of seconds is measured together with the number of program loops to the next tick. This technique assumes that multiprogramming is suspended. Experiments on KDF9 gave as good results on the machine as the accurate timer, although some time was wasted waiting for the clock tick.

Several factors are likely to effect the accuracy and repeatability of the results. Firstly the clock measurement itself needs to be considered. On several machines a figure is given in small units (microseconds say) but the accuracy is lost because not all interrupts reset the clock correctly. Under these circumstances, a low interrupt rate must be arranged by running the program without multiprogramming and suspending any input or output during the timing itself. If input/output is not suspended, then apart from interrupt processing, the cycle-stealing of autonomous peripheral transfers must be considered. This is only likely to be significant with fast devices, but can again require that the program is run in isolation. On some processors, the maintenance engineer can adjust the speed—mainly required for marginal testing. If this adjustment is possible, then the engineer must be consulted so that the correct setting can be used.

In spite of the precautions mentioned above further difficulties have arisen. For instance, the time certain instructions take can depend upon the operand values—as with the floating point operations which often require variable times owing to normalising shifts. Hence it is important that the variable values used in the timing program are consistent. On some machines, the times taken can depend upon the position they occupy in relation to the word boundaries, and also upon the preceding and succeeding instructions. These difficulties arise when instruction lengths are different from the word length or if the processor does instruction look ahead. The simplest technique for overcoming most of these troubles is to repeat the statement within the loop a few times. This is only necessary with the simpler statements. A check should be made that the resulting compiled code is a repetition of the code from a simple statement. The very short statements on KDF9 gave trouble for both the above reasons, and so a short assembly code program was run to overcome this. This program took an instruction sequence (without a jump) as data, and copied it many times in the core. All these repetitions were timed. The instructions were then placed in a different position in relation to the word boundaries and the process repeated. The timing program in the report (Wichmann, 1972c) attempts to overcome some of these difficulties, at the ALGOL 60 level.

It is clearly convenient if computers can provide a consistent measure of processor usage. Atlas has an instruction counter for this purpose which was used in the timing tests. This does not represent real time, but it does make some allowance for the different time various instructions take. It is also used for accounting purposes. There seems no good reason why a computer should not have both a megacycle clock and an instruction counter. The counter and clock could be updated automatically by the interrupt hardware.

Short instruction sequences can be checked by referring to the processor specification. As a last resort, if consistent times are impossible to obtain, the processor specification times can be taken. This is obviously unsatisfactory since no independent check is made. For a machine designer there is no alternative—and indeed it is quite feasible to produce figures for a new machine provided a detailed compiler specification is available.

The compiler software can also cause difficulties with the timing. Two such points have already been mentioned, that the for loop control code should be independent of the statement, and that the code for repeated statements should be a repetition of the code from a single statement. If these conditions are not satisfied then either the source text should be altered appropriately, or estimates made upon the code produced without any optimisation. For instance if $x := y; x := y$ is optimised to LDA Y, STA X, STA X then the source text can be changed to $x := y; y := z$, etc. The aim is clearly to time a *typical* assignment statement of the form $x := y$. In the same way, if a processor has interleaved store or a cache store then a statistical distribution of address should be taken rather than the ones resulting from this simple timing program.

A further software difficulty arises with short address length machines. On the 360/System 4/Spectral 70 machines, a base address register must be set up every 4096 bytes for instruction addressing. Additional instructions to swap base registers can occur anywhere in the code, although a good compiler would attempt to place such code outside program loops. Similar situations may arise with other machines even though there is no logical reason for it, purely for compiler convenience. For instance, a compiler table may overflow forcing the production of different code.

2.2.2 CHOICE OF THE STATEMENTS

Unfortunately the choice of statements is somewhat arbitrary. The original timing program contained over 60 statements with the variables declared at main program level and also in an inner block. This program was run on Atlas, 1905 and the KDF9, but unfortunately not all the times were available. In writing a note for the ALGOL Bulletin (Wichmann, 1968), the author selected those statements for which times were known. To these statements the time taken to execute the loop code in **for** $i := 1$ **step** 1 **until** n **do**; has been added since this is produced as a by-product of the other times.

The resulting statements do embrace most of the important features of ALGOL, the most serious omissions being conditional expressions and statements, and call-by-name evaluation. A very much better choice of statements could now be made on the basis of the statistics given in section 2.3.

2.2.3 THE ACTUAL TIMES OBSERVED

The times available for those machines in current use are tabulated in Appendix 2. It is important to remember that these times are those which have been measured with a version of the stated compiler and machine. The times are not necessarily the ones which the current compilers would give on the respective machines. It is almost impossible to keep up with releases of compilers, and so in any critical cases, the timing tests should be run. Details of the timing program are given in Wichmann, 1972c.

Most of the figures have been produced by interested users who have access to the relevant machines. The accuracy of the results depends upon the machine clocking facilities, and the perseverance of the person conducting the tests in spite of frequent difficulties.

2.2.4 ANALYSIS OF THE TIMES—A MODEL

A large amount of data is involved in the complete matrix of statement times. It would be very desirable to reduce the data on one machine to a single figure of merit. This would be straightforward if there was a constant ratio between the statement times on the different machines. This is clearly not the case, but such a ratio can be constructed from a fitting process. Even though the ratios are not the same, one would expect that the time for a statement on a particular machine was roughly the product of a factor depending only on the machine, and a factor depending only on the statement. So that, if the time for a statement i (i = 1 to n) on machine j (j = 1 to m) is T_{ij} then

$$T_{ij} \approx S_i \times M_j$$

where S_i = time depending on the statement only M_j = factor for the machine, taken as unity for one machine (arbitrarily).

A fitting procedure based upon this model has been kindly calculated for the author by M. G. Cox. The method used is as follows:—
One requires

$$T_{ij} = S_i \times M_j \times R_{ij}$$

where the R_{ij} are required to be as close to 1 as possible. In order to linearise this, logarithms are taken, so let $LT_{ij} = ln(T_{ij})$ etc, then

$$LT_{ij} = LS_i + LM_j + LR_{ij}$$

where the LR_{ij} are required to be small. Using a least squares fit for the LR_{ij} one has

$$E = \sum_i \sum_j \{LR_{ij}\}^2 = \sum_i \sum_j \{LS_i + LM_j - LT_{ij}\}^2$$

One requires a minimum for E with respect to the S_i and M_j. So by differentiating with respect to these variables gives

$$\sum_j \{LS_i + LM_j - LT_{ij}\} = mLS_i + \sum_j LM_j - LT_{i*} = 0$$

$$\sum_i \{LS_i + LM_j - LT_{ij}\} = \sum_i LS_i + nLM - LT_{*j} = 0$$

where LT_{i*} and LT_{*j} are the row and column sums respectively of the matrix (LT_{ij})
By taking $M_1 = 1$, one can solve the equations explicitly giving

$$LM_j = (LT_{*j} - LT_{*1})/n, j = 1 \text{ to } m$$

$$LS_i = LT_{i*}/m + LT_{*1}/n - LT_{**}/mn, i = 1 \text{ to } n$$

where LT_{**} is the sum of all the elements of (LT_{ij})

By taking exponentials, values of the S_i and M_j can be calculated very quickly and easily.

An important application of this method is in the estimation of missing times. Invariably some values are missing or known to be suspect. Omitting a whole row or column would result in an unacceptable loss of information, so an alternative must be found. The technique is to guess a time for the missing value, perform the analysis above, and then use the product of $S_i \times M_j$ as a second estimate, and so on. The process converges very rapidly provided each row and column has a good number of known values. In some cases, one may be able to make a better estimate because of additional information—for instance, if the number of instructions executed in the statement is known. Without such additional information, one is unlikely to be able to estimate a closer value, than the one given by the iterative procedure above.

2.2.5 THE LIMITATIONS AND USES OF THE MODEL

Although the model here is only used for the analysis of statement times, it can be applied to a much wider range of performance data. Rather than statement times, the times for complete jobs could be taken. A wide variation of task times can be considered in one analysis because the difference is automatically absorbed into the statement factors.

The analysis gives a figure of merit based upon all the available data without considering any one statement to be more important than any other. This is not a true performance measure of the processing speed of ALGOL 60, since some statements are obviously more important than others. For instance, array declarations are executed much less frequently than the simple assignment statements (see 2.4).

The method used to fit the model to the available data is only appropriate in certain circumstances. Ideally, for a least squares fit one requires that the data is "random" and distributed normally. However the variations can be related to a small number of causes. In fact there is a high correlation between the data. For instance a long time for procedure entry would affect all the four calls of dummy procedures and possibly those of the standard functions. The distribution of the LR_{ij} matrix is far from normal. It appears to have the shape of two normal distributions one of small and the other of large deviation. This is hardly surprising since certain features of ALGOL are handled in very similar ways, for instance the simple assignment statements and the standard functions, whereas in other areas, particularly block and procedure entry and exit, radically different techniques have been used. A more elaborate fitting process could be used which is appropriate to this distribution, but the simplicity of the least squares fit combined with its wide acceptance makes it the obvious choice.

There are cases where it is inappropriate to use the model because the basic assumption is unsound. For instance, systems with software floating point should not be compared within the same analysis to those with hardware floating point. The reason for this is that the ratio between the expected and actual statement times depend upon the use of floating point rather than any other underlying trends. For the same reason an interpretive scheme should not be compared directly with others. The speed of statements with an interpreter is likely to depend upon the number of interpreted instructions rather than the amount of additional work to perform. A different example arises with compilation times from various compilers. Many systems take some time to compile a null program while others do not do this. One compiler might take 10 seconds plus a millisecond per input line whereas another takes just two milliseconds per line. Obviously any comparison between the compilers on the compilation of different programs must take the constant of 10 seconds into account. The method used above does not do this and so is inappropriate.

A glance at the residual matrix in Appendix 3 reveals that quite a significant number of statement times are anomalous—sometimes being as much as ten times faster or slower than expected. Such values obviously do have an effect upon the analysis as a whole. However, these values cannot be rejected on statistical grounds since they are well within the variation that would be expected from such a large data base. With the number of statements and machines involved, the net effect of the anomalous values is not significant.

The addition of a further machine (or statement) can change the results on the previous machines. This is because any new machine can make slight changes to the statement factors which will in turn affect the machine factors. However, only slight changes have been observed after five machine times were available, and from fifteen machines onwards adding a new machine did not change the existing machine factors by more than one part in a thousand. This is purely an arithmetic fact—it certainly does not imply that the machine factors have an error of less than one part in a thousand. Actually, most of the timing data is only accurate to 10%, and most performance data is usually quoted as having a 20% error.

A glance at the R_{ij} matrix or residuals reveals that each compiling system has a characteristic pattern. This pattern is usually maintained over revisions of the system—although not necessarily so. For instance, it is easy to spot whether a change of method in the evaluation of the standard functions has taken place. The pattern is also preserved when running the same compiler on a different member of a range of computers. The pattern does change slightly with an alteration in the relative speed of the core store and processor, or an enhancement in the floating point unit.

It is important to note the degree of checking performed by the system. The options chosen (if any are available) are those allowing the fastest execution without any additional restrictions on the language. In practice this means no subscript checking (at least on conventional machines) and no dynamic flow of control analysis (retroactive tracing, for instance, see 5.4.2). The problem of providing adequate checking while allowing reasonable execution rates is considered in detail in Chapter 5.

2.2.6 SOME REMARKS ON THE OBSERVED TIMES

Several of the compilers are considered in more detail in chapter 9 so only brief remarks on the times obtained will be made here. One would like to be able to give precise reasons why some of the time are anomalous—say those which are more than twice as fast or less than half as fast as expected. Only in those cases where the generated machine code is known, or contact has been made with the relevant compiler writer is such an explanation possible.

Atlas

These times were produced using an instruction counter by F. R. A. Hopgood of the Atlas Laboratory, Chilton. Although this does not represent true time, the counter is consistent, and is used for accounting purposes. Very accurate statistics are available on the average length of an instruction counter tick (3.01 microseconds). The counter is incremented by more than one for multiply and divide instructions. The compiler is considered in detail in 9.3.

1904A XALT Mk5

The times were produced by ICL and are dated 11/71. They are accurate to about one microsecond, but were produced without statement repetitions. The time for **begin real** a; **end** was zero as no code is produced. In these circumstances a time of about half an instruction time is inserted to avoid a singularity in the analysis. This does not affect the mix (see 2.4) as the corresponding weight is zero.

The statements $k := l \times m, x := abs(y), x := sign(y)$ are relatively slow. The first appears to be caused by the hardware multiply instruction and the last two use the standard external procedure calling mechanism. The statements which are relatively fast are

k := 1.0	type conversion at compile-time
e2[1,1] := 1	subscript optimisation
e3[1,1,1]:= 1	subscript optimisation

begin real a; **end**	no code
switch	simple switches optimised
$x := ln(y)$	special coding for y = 1.0

The optimisation of constant subscripts invalidates the mix figure since it is assumed that the times for these statements are representative of subscripts in general. A reduction in the mix figure (see 2.4.3) of about 5% should be made to allow for this.

1906A XALT Mk5

The times were produced by ICL and are dated 12/71. Due to problems with instruction look-ahead on the machine, the shorter times were repeated with statement repetitions, giving an observational error of at most 3%.

As the compiler is the same as for the 1904A, it is only necessary to comment on the differences which are not just due to the faster processor. The time for **begin real** a; **end** was put at 0.4 microseconds, in line with the increased power of the machine. The standard functions and $\uparrow$ are noticeably relatively slower on the 6A, whereas the open code statements are relatively faster than the 4A. This is presumably caused by the increased time for jumps with instruction overlap and interleaved store. A similar reduction of 5% should be made to the mix because of the subscript optimisation.

ALGOL 60/RRE

These times were obtained on a dual processor 1907F at the Royal Radar Establishment, Malvern by C. T. Sennett. Because of the dual processing accurate times are hard to obtain. The main purpose of the figures is to give a comparison with the same machine running ALGOL 68-R. The compiler used is the standard ICL ALGOL compiler XALT. A later version than that analysed in section 9.8 was used, as can be seen from the times. Storage is assigned at procedure level as before, so that no code was produced for **begin real** a; **end**—a time of 1.5 microseconds was inserted in line with the power of the machine.

ALGOL 68-R

This is the first of several compilers which are not true ALGOL 60. The comparison is thought to be a fair one in that the language encompasses all the constructs used in the simple statements. In such cases, the most natural coding is used for the statements. For instance, with these times, a case statement has been used instead of a switch, and *abs*, *sign* and *entier* are performed by operators. Clearly a case statement is easier to compile efficiently than a switch, but this is not thought to be too favourable due to the fact that most uses of a switch can be coded as case statements. The restrictions in ALGOL 68-R are mainly to allow one pass compilation and are not of any great practical importance (see Currie, 1971). Compile time type conversion is not performed with $x := 1$ although it is in the ICL compiler for ALGOL 60. Better coding would evidently be used for $\uparrow 2$ and $\uparrow 3$. The procedure call mechanism for ALGOL 68 does not require a "thunk" mechanism, and so is easier to manage. Although the top of the stack address is not kept in a register, the "precall" method of parameter evaluation is used to give very good times. Block level storage is used, although this is being altered in a future version of the compiler.

B5500

As is well known, this computer has special hardware to facilitate the execution of ALGOL 60. In fact there is no assembly language (in the ordinary sense) so ALGOL is used instead. For instance, the ALGOL 60 compiler is written in ALGOL 60.

This means that the design aim of the B5500 Extended ALGOL language is to provide effective access to the machine's facilities rather than provide a strict ALGOL 60 system.

The times were produced by J. Thomas of the Post Office Telecommunications Headquarters. A detailed analysis of the system appears in 9.5.

B6500 and B6700

These two computers are identical from the software viewpoint, and the main hardware difference is the processor speed. Both machines had a 1.2 microsecond core and 5 Megacycle clock rate. The computer design and that of the supporting supervisor rests heavily on the experience gained with the B5500. The pattern of instruction times is very similar to the B5500.

As expected, the procedure entry times are very fast, but not quite so fast (relatively) as the B5500. This is presumably due to maintaining a complete display for variable access unlike the B5500. All the Burroughs machines required storage allocation for arrays to be a supervisor function and hence give slow times. All the machines give faster times for $x := 1$ than $x := 1.0$ because shorter instruction sequences are generated for integer literals compared with reals.

Both these machines and the B5500 suffer from the disadvantage of only having two of the top cells of the stack as registers. This means that in terms of core cycles, the access required to evaluate a complex expression is not significantly better than a conventional one-address machine.

The times for both machines were produced for Barclays Bank Ltd and are dated 5/71. In order to avoid errors due to supervisor interrupt routines, the minimum time to execute the statements was taken. This has the advantage of giving consistent results, but does not make any allowance for the supervisor overheads which cannot be avoided during ordinary program execution. It is not at all clear whether an allowance should be made for the supervisor and if so, how this should be arranged.

X8 ALGOL

The figures were produced by J. Nederkoorn of the Stichting Rector Centre, Holland, in conjunction with Brinkhuyzen of the Mathematical Centre, Amsterdam. The computer is the Electrologica X8, a successor to the Xl computer which was used by Dijkstra in writing one of the first ALGOL compilers. The computer has a number of facilities to allow convenient implementation of certain features of ALGOL, viz:

(1) Fast instruction to allow detection of real or integer type

(2) Display handling instructions

(3) An instruction to allow efficient implementation of call-by-name.

The original compiler was produced by the Mathematical Centre, Amsterdam, but several installations have made local modifications, since the system is well documented. The Reactor Centre version contains substantial improvements in the array accessing method, which previously checked dynamically the type, dimension, round-to-integer and subscript value. This version usually does only subscript bound checking at run time, and then only on an assignment. Unfortunately, the basic statements have the subscripts more frequently on the left hand side of an assignment, which is not typical in practice (see 2.3.1.10). Hence the mix figure will be slightly pessimistic.

The figures are dated 10/71, and were measured with statement repetitions, but accurate timing appears to be difficult on this machine, so an error of 10% was obtained.

X8 Eindhoven

Although this is the same machine as the X8 ALGOL, it has a completely different compiler designed to work under the T.H.E. operating system (Dijkstra, 1968; Bron, 1971).

The main difference from the point of view of processing speed is that a virtual memory system has been implemented by software, which means that various operations *could* invoke a substantial overhead. It is not at all obvious how this overhead ought to be spread over the statement times, so the mix figure obtained should be treated with extreme caution. A further difficulty is caused by the fact that short arrays are handled differently from long ones. Short arrays are placed upon the stack, whereas long ones are placed off the stack as in the B5500. This means that in order to assess the speed of this compiler properly one would have to know the extent to which the short option would be used.

On the basis of the mix figures, the Eindhoven compiler is about 10% slower than the original Mathematical Centre compiler, but 40% slower than the Reactor Centre version. Against this must be weighed the usefulness of having a virtual memory system where the programmer does not need to be so conscious of the addressing limitations of the machine.

The times were obtained by C. Bron of Eindhoven, who also had similar difficulty obtaining accurate figures, even with statement repetitions. Owing to the virtual memory difficulties as well, the errors could be as high as 20%.

KDF9 Edgon

This compiler was produced by ICL at a rather late stage in the development of KDF9 as a result of a report on University Computing (Flowers, 1966). It was written by E. N. Hawkins, who had already produced one ALGOL compiler for KDF9. As a consequence many of the deficiencies of the previous compiler were rectified. The compiler is essentially two pass, the first pass of which is virtually identical to the only pass of the Whetstone compiler (Randell, 1964). The machine code produced does not contain any global optimisation, but a substantial amount of local optimisation is performed. For instance, Boolean expressions are not standardised if this is not necessary, the **step** 1 **for** loop code is good, and the short form of various machine instructions are used wherever reasonable. The compiler now contains extensive compile-time options to allow tracing, overflow checking and array bound checking. The most significant advantage over the Kidsgrove ALGOL compiler is that it produces substantially more compact code. The reasons for the anomalous times are that the exponential operator was handled incorrectly as (real) $\uparrow$ (real) even if the exponent was integral, and that *sign*, *entier*, and *abs* are handled by open code.

KDF9 Mk2

These figures were obtained by the author using the Kidsgrove compiler without the optimising phase (see chapter 10). The compiler is discussed in detail in 9.4, but this version contains various enhancements listed in 10.3. The original one is about 20% slower, as can be seen from the summary in Appendix 8. As has already been mentioned, the shorter statements were timed by making a large number of copies of the machine code sequences, and hence are accurate to half a microsecond.

Babel—KDF9

This language is not ALGOL 60, but contains almost all the features of ALGOL 60 and many of the extensions included in ALGOL W (see Scowen, 1969b). Naturally, most of the simple statements generate code which is identical to that of the two other KDF9 compilers. In a similar manner to the other ALGOL-like languages, a few changes are necessary to some of the statements. The switch becomes a case statement, *abs*, *sign* and *entier* are replaced by operators, and ↑ **real** uses *ln* and *exp* explicitly.

Array access is always with subscript bound checking, and on KDF9 this is necessarily slow. The **goto** is very slow indeed because the syntax of Babel demands that the label is placed directly within a block. Hence, the actual time measured includes the block entry and exit (1.4 milliseconds) as well as the actual **goto** (0.5 milliseconds). Since Babel has a richer set of control structures than ALGOL, an explicit **goto** should be less common which means that the mix figure is pessimistic. Assuming the use of **goto**s could be avoided completely in Babel, the mix figure should be increased by about 7%.

Procedure and block entry times are also slow in Babel. This is mainly due to the generality of the storage allocation system (three separate stacks are used) which is designed to support several extensions. It would be possible to reduce these overheads if some forms of extensions were precluded.

The times are dated 5/70 and are accurate to about 3%.

CDC 6600 version 2.0 and CDC 6400

These times were obtained with statement repetitions by J. Wielgosz of the University of London Computer Centre. They appear to be accurate to a few per cent although figures that are about 10% faster for the 6600 have been sent to the author by Brinkhuijsen from Amsterdam. However these faster times did not use statement repetitions and so are likely to be less accurate, especially in view of the timing problems on the 6600.

The compiler is considered in detail in 9.7. It is interesting to note that ALGOL programs appear to run only 2.2 times faster on the 6600 compared with a factor of 3 for FORTRAN. This is probably because the ALGOL compiler makes less use of the autonomous registers, which gives the 6600 its high speed.

ICL 4/70 and 4/50

Both these times are from the standard ICL ALGOL 60 compiler for the System 4 range. Both systems appear to be significantly better at processing integers (32 bits) than reals (64 bits). The standard functions are rather slow, which is thought to be due to preserving and restoring all the registers on calling an “external” procedure, i.e. one whose text is not included in the program.

In most respects the compiler is very similar to that of the KDF9-Egdon system. Storage is allocated at block level, as can be seen from the dummy block time.

ALGOL W 4/75

The compiler is identical to that of ALGOL W 360/67, but is run on hardware which is the same as the 4/70. The figures are not however strictly comparable. For instance, 32 bit reals are used in ALGOL W (one must declare variables as **long real** for 64 bits), whereas the ALGOL 60 compiler uses 64 bit reals. This could also affect the times for the standard functions. Differences in the source language must also be borne in mind, but even so, the additional speed of ALGOL W is considerable.

ALGOL W 360/67

These figures are somewhat exceptional in that several enhancements have been made to the compiler since the first analysis was produced for the author by E. Satterthwaite at Stanford. Hence the compiler has been tuned to a certain extent to do well on these tests. A further improvement could be made to increase the speed of the procedure calls, by evaluating value parameters in the precall sequence. This was not done owing to **real—long real** type conversions which may be involved. This possibility can be removed by only performing the optimisation when a procedure cannot be called formally (see 7.2). A formal call can be ruled out for most procedures on a single pass (the remaining procedures may be called formally).

IBM 360/65

Since these figures have been produced on virtually identical hardware to that of ALGOL W, the differences are somewhat surprising. The main trouble arises from the block and procedure entry figures which include about 370 microseconds for two supervisor calls to claim and release core-store. A semi-official modification is available which removes these calls. The times with this improvement have been sent to the author by J. W. van Wingen and appear in the summary in Appendix 8. The original figures were produced by P. Samet of University College London before the improvement package was available. On the other hand the compiler evidently does subscript optimisation, as the times taken for one, two or three dimensional array access are the same. As with the 1900 XALT compiler, this really invalidates the basic statement technique, since the intention is to measure the speed of typical array accesses.

Apart from the improvement package, an option exists for 32 or 64-bit reals. The difference this can make for both the 360/65 and 360/50 can be seen from the additional times in the summary in Appendix 8. The slowness of the procedure call had been noted by Sundblad (1971) in the calculation of Ackermann's function.

IBM 370/165

Times for this machine were received from R. Brinkhuijsen of the Mathematisch Centrum, Amsterdam. The program used did not repeat the statements, so it is thought that the times may not be particularly accurate, although the loop code was timed several different ways to avoid consistent errors from this.

The mix figure obtained must be regarded with extreme caution because of the cache store. The program loops which do the timing are very small so that all accesses are likely to remain in the cache store. Hence the times obtained will probably be on the low side because the hit rate on the cache store is higher than would be usual with most programs. To overcome this defect, it would be necessary to know what hit rate is to be expected from average scientific programs written in ALGOL and then repeat the statements so as to achieve this. For instance, instead of writing $x := y$, one might write $x := y, n := v, w := u$, etc., each statement being of the same form but using different variables to fill the cache store.

ALCOR 7094/1

The overall design of the compiler is that of the ALCOR group, as described by Grau (1967). The techniques used in this particular system are given by Bayer *et al*. (1967) and Gries (1965). The times were produced by E. Hansen of Copenhagen.

The weakest feature is the procedure call times. Although some advantage is taken of the specification of the parameters, this is clearly not adequate to compete with those systems which demand complete specification. The times for the other statements are very consistent, which suggests that significant improvements in the compiler is unlikely. Some optimisation is done with $\uparrow 2$ and $\uparrow 3$. It is known that storage is assigned at block level, although code is generated on entry or exit to a block, but this may be to allow for array storage, or for diagnostic information.

As with the IBM 360 compiler, the optimisation of array subscript calculation invalidates the corresponding statements. In fact an option is available to remove this optimisation, but bound checking is also included which would be unduly pessimistic.

NU 1108

These times were produced by P. T. Cameron of Univac Ltd for the compiler written in Trondheim, Norway, which now replaces the previous compiler. The new one is a very substantial improvement both in terms of performance and language level. The old compiler appears in the summary in Appendix 8.

Statement repetitions were not used, so the shorter statements may be somewhat in error. They are dated 4/71. A detailed analysis of the performance of this system appears in 9.6.

PDP10

This compiler has been written by Carnegie Mellon University for the Digital Equipment Corporation. The times were produced using statement repetitions. The machine used had the KA10 processor and one microsecond core-store, and is dated 11/71.

Theoretical times for the shorter statements are about 30% faster than the observed times. The observed times are self-consistent to about 5%, so it is clear that the additional overheads (cycle-stealing, monitor interrupts, etc.) are fairly uniform.

The following points about the times should be noted. Type conversion is slow except where it has been performed at compile time ($x := 1, k := 1.0$). The operation $\uparrow$ 2 has been optimised but $\uparrow$ 3 has not. There is almost certainly some exceptional coding with $x := y \uparrow z(z = 1.0)$ and $x := ln(y)(y = 1.0)$. Procedure calls are rather slow, which appears to be caused by the handling of parameters. The function *sign* is slow although *abs* is fast (which is unusual). The **for** loop control code has been optimised, but this may be because of an ALGOL extension which allows the **step** 1 to be omitted. It is not clear if the optimisation would have been performed if **step** 1 had been included.

The following machines are thought to be of less general interest so that detailed times are not given, the overall performance only being listed in Appendix 8.

4130 (2 μs and 6 μs versions)

This computer was available with two different core-stores. The figures therefore give a good indication of the changes that can be expected from hardware alone. The additional speed is largely absorbed into the machine factors so that the residuals for these are very similar. The increase in speed with the two microsecond machine is clearly not so marked with the statements involving substantial processing. Storage is allocated at block level since the time for **begin real** a; **end** is quite substantial. The overall speed ratio between these machines is almost exactly two, and this can be found from the machine factors, the ALGOL mix (Appendix 8) and also the Gibson Mix. A purely hardware measure of performance like the Gibson Mix can only be regarded as satisfactory when software factors have been removed. This, of course, happens when considering different machines in a single range running the same software.

H. S. P. Jones of the University of Warwick kindly produced the times for the six microsecond version and Thompson of Leicester University for the two microsecond version.

Elliott (ICL) 503

This compiler was designed on the same lines as the 4130 system (see Hoare, 1964). Subscript checking is always performed and this does incur a significant penalty because, unlike the 4100, no special hardware assistance is available. Procedure calls are fast, but this is thought to be due to a language restriction which allows parameters and local variables to be placed in fixed storage.

The times were produced by E. Carter of the Rocket Propulsion Establishment, Wescott.

CDC 3600

These figures were produced by Per M. Kjeldas of the Kjeller Computer Installation, Norway. The compiler is not that provided by CDC but one written at Kjeller. The figures are apparently accurate to one microsecond (which is the speed of the core store). It can be seen to be a different compiler since the pattern of the residual matrix is quite distinct from that of the CDC 6600. No details are available concerning the compiler, but the slow times for procedure entry have been attributed to some optimisation when the actual parameter is a simple variable with call-by-name parameters. It is unfortunate that such optimisation should affect value parameters, which as has been shown, can be implemented very efficiently.

Optimisation of $\uparrow 2$ and $\uparrow 3$ and simple **gotos** would appear to be very worthwhile. The times for entry to a dummy block suggest that procedure level addressing is used.

2.3 Program Analysis—General Techniques

Performance measurement must necessarily rely upon an accurate description of the workload involved in a test. Hence statistics on the use of ALGOL 60 are fundamental to any work in this field. Apart from its obvious use in assessing the execution rate of compiled code, details on the nature of the source text could be an advantage to compiler writers in the optimisation of the compiling process itself.

Static analysis of programs is particularly easy, and it is therefore surprising that very little information is available in this area. If an operating system provides facilities for storage of source text which is widely used within an installation, then scanning such text by program will provide a good sample for static analysis. A similar scan can be made of binary programs, especially if they are stored in a ‘relocatable’ form where the references to external routines are apparent. Such data can give the relative occurrence of ALGOL constructs which require different subroutines to handle them at run-time.

Dynamic analysis is obviously more important, but usually harder to obtain. With a true compiler, the resulting binary program cannot easily be related to the source text. Hence, although it may be possible to find the number of STA instructions, for instance, that are executed in a program, it is not necessarily possible to find the number of assignment statements executed. If options exist for tracing, then it may be possible to replace the trace routines by ones that merely count assignment statements, array variable fetches (from the bound-checking subroutine), etc. Facilities of this nature should be available with compiling systems, if only to provide the user with a method of analysing his own programs (see chapter 6).

Two further techniques have been used very successfully for dynamic analysis. D. S. Akka was able to insert a modification into the Atlas Autocode compiler at Manchester in order to accumulate counts which could be related to the source text. This was largely possible because the compiler was constructed by using the compiler-compiler and hence had a particular structure. The hardware of Atlas meant that the overhead in accumulating these counts was not excessive. This is important, since otherwise a very restricted sample of programs must be used.

The second method which was used by the author, consisted of a modification to an interpretive scheme. This scheme is used extensively on KDF9 for program testing because the compilation speeds are very high. The slowness of the interpreter meant that the degradation due to the capture of the necessary statistics was very small. Hence every program under test was monitored, avoiding any selection process (production programs are executed using the true compiler). The main advantage of this technique is that the interpretive instructions can, with a small analysis, be related to the source text.

In many cases, statistics of one form or another are gathered as a result of routine computer operations. Such statistics can be very helpful and should be studied with a view to gaining a deeper understanding of user programs, the compiler and operating system, etc.

2.3.1 WHETSTONE ALGOL DYNAMIC ANALYSIS

In this section, the analysis depends critically upon the internal structure of the implementation of ALGOL given by Randell and Russell (1964). The book describes the system in substantial detail so that it is not necessary to give more than an outline here. Any interested user can make their own analysis by referring to the book and the statistics given in Appendix 6.

The method used was to make a modification of only a few instructions to the Whetstone interpreter. The modification consisted of accumulating in a table the total number of times each interpretive instruction was executed. At program termination (either successful or on failure), the accumulated counts were output. Initially these counts were punched on paper tape, but later they were accumulated on the disc store of the machine. C. Phelps of Oxford University made a similar modification which allows an independent check to be made.

Repeated references to Randell and Russell's book is inevitable in this analysis; they are abbreviated to the form RR 2.6.2 where the digits refer to the relevant section. Also, the interpretive instructions themselves, which play such a key role are written in the abbreviated form with capital letters, e.g. MSF or when the context is not clear expanded in the form Make Storage Function.

The interpreter is extremely slow, taking from 300 to 800 microseconds per instruction, so the overhead of accumulating the counts of about 40 microseconds is not significant. As noted before, all programs submitted to the system were monitored, the only selection being the monitoring period.

In interpreting these results, one must bear in mind the environment within which these figures were obtained. The National Physical Laboratory is a research establishment in which, in general, the scientists do their own program writing and debugging. Program development takes a large share of the available computer time, almost all of which is in ALGOL. Although some of the programmers are very experienced, it is thought that a fairly high proportion have little programming experience, and that this is reflected in the programs they write.

2.3.1.1 Outline of the statistics

For the purposes of the calculation of variance, the figures have been divided into seven groups as follows

(1) Three days separated by a fortnight at NPL, collected late in 1968. The separation was an attempt to get different programs rather, than a rerun of a few programs.

(2) Figures from Oxford University obtained in April 1969.

(3 to 7) Figures collected automatically during three months in 1970. The four groups represented about 7 to 10 days use of Whetstone ALGOL. Re-runs of the same program would be quite common, but a substantial volume of data was collected.

Sometimes it is appropriate to give all seven figures, in which case they are written as "total (f1,2,f3,f4,f5,f6,f7)". Where appropriate the values are rounded to three places. Usually one is interested in the frequency of execution of one of the interpreted instructions, in which case the average of the frequency from the seven samples is given (as operations per million). The variance of the seven frequencies as a percentage is then given in brackets as an indication of the reliability of the information. For instance, the frequency of execution of Block Entry is 13400 (26), meaning 13400 operations per million with a variance of about a quarter. As expected, the more frequent operations have a smaller variance (typically 10%) as opposed to the less frequent ones with a variance of often 100%. In general, the number of programs in the seven samples is an indication of the comprehensiveness of the data, except that rather more care was taken in selecting the programs in the first set.

The number of programs executed was 949 (93, 120, 122, 349, 71, 121, 120). The number of elementary operations in each set was 155 (12, 6.3, 16.3, 67, 12, 25.4, 15.8) millions. Hence the average number of operations in each program was 152000 (165000, 63500, 133000, 192000, 170000, 210000, 132000), so that the Oxford University programs were significantly shorter. The average time taken to execute one instruction was 664 (738, 761, 598, 593, 616, 655, 687) microseconds. The percentage of programs to terminate without error was 54 (58, 55, 51, 55, 41, 49, 63).

The figures give very accurate details of some aspects of the use of ALGOL. For instance, the number of times each standard function was used is known. But in other cases the operation counts give little information. For example, all arithmetic operations performed upon items in the stack give no indication of the type of the operands. There is only one + operation, which is integer plus if both operands are integer and real plus otherwise. Another difficulty is that the array element accessing instructions (INDex Address and INDex Result) do not indicate the array dimension—this is deduced from the current state of the stack (RR 2.1.3). In such cases, information from a different source can be used to give an estimate of the missing data.

2.3.1.2 The operators

The frequency of the arithmetic and Boolean operators can be found directly from the corresponding elementary instruction counts. These are summarised in Table 2.1.

As has already been stated, the types of the operand are not known. A very rough guess is attempted on the ratio of real to integer working based upon other instructions in section 2.3.1.10.

The asymmetry between = and $\neq$ is not surprising, since it would be more natural to write **if** $i = j$ **then** 81 **else** 82 rather than **if** $i \neq j$ **then** 82 **else** 81. The other

Operator	Frequency	Variance
$\times$	36 800	(9)
$+$	31 000	(14)
$-$	26 300	(11)
/	11 000	(27)
$=$	6 980	(26)
NEG	5 230	(21)
$>$	3 600	(24)
$\uparrow$	3 530	(59)
$<$	3 290	(38)
$\neq$	2 230	(56)
$\vee$	2 180	(32)
$\leq$	1 030	(70)
$\geq$	914	(86)
$\wedge$	884	(74)
$\div$	481	(70)
$\neg$	115	(107)
$\equiv$	1	(241)
$\supset$	0	(∞)

Table 2.1: Use of the ALGOL operators

relational operations are symmetrically used to within the statistical variation of the data. Note that the total of all the relational operations (11 100) greatly exceeds that of the Boolean operations (3200). This implies that the non-standardisation of Booleans recommended in 1.4.1 is certainly very worthwhile.

The fact that multiply appears higher in the table than add does not mean that at machine-code level, multiplication is more common than addition. Addition is implied in a large number of the other operations, for instance, in the array accessing operations (INDex Address, INDex Result), storage allocation (Make Storage Function, Procedure Entry, Block Entry, Call Block) and the for loop control code (FORS2). It is, however, possible to make a rough calculation on the usage of actual machine code instructions. The table illustrates that the less frequent instructions do have a very much larger variance.

2.3.1.3 Use of constants

Whetstone ALGOL inserts constants within the instruction sequence, that is, there is no constant pool. So it is not possible to say if any storage would be saved if different uses of the same constant made use of the same storage cell. With the technique used by Randell and Russell, it is never necessary to “address” a constant, since the constant is part of the operation.

In analysing these (Table 2.2), allowance must be made for the fact that constants as actual parameters to procedures do not give rise to these operations (RR 2.5.4.2). The use of special operation-codes for 0 and 1 is clearly fully justified. In a similar way, compiler writers should attempt optimisation for the coding of these two important constants. It is thought that a large proportion of integer constants not 0 or 1 are small, and so could for instance be accommodated in one byte.

Operation	Frequency	Variance
Take Integer Constant 1	43 200	(4)
Take Integer Constant	22 500	(33)
Take Integer Constant 0	6630	(25)
Take Real Constant	5450	(44)
Take Boolean Constant False	124	(68)
Take Boo1ean Constant True	78	(126)

Table 2.2: The constant operations

2.3.1.4 Labels, switches and jumps

In ALGOL 60, the majority of transfers of the flow of control are implicit in the language constructs. Conditional statements and expressions, for loops, and procedure calls all enable the programmer to achieve the necessary program flow without explicit transfers. In many people's opinion (including the author) this adds greatly to the power of ALGOL in comparison to FORTRAN.

The frequency of **goto**s can be found from the operation GoTo Accumulator and is 2010 (51), whereas the use of switches is very much less 94 (105) which can be found from the operation Take Switch Address. In Whetstone ALGOL, the correct switch element is found by going down a chain of Decrement Switch Index operations (RR 2.4.2). Hence the average value of a switch index can be found from the ratio DSI/TSA, which is about fifteen (with a very large variation).

On passing a label or entering a procedure the operation TRACE is called to preserve the appropriate identifier in a cyclic buffer (RR Appendix 5). Hence the number of times a label is passed can be found from TRACE count minus Procedure Entry count. This is 3370 (44). The percentage of **goto**s to passing of labels is 58, with a very small variance of 8. So, roughly speaking, a label will be passed once, and be reached equally often by a **goto**.

No statistics are available on the extent to which a **goto** leads out of blocks or procedures. However, formal labels are certainly very rare as can be seen from Check Label 59 (91) and Check and Store Label 2 (208) which correspond to the number (dynamically) of labels and labels by value respectively. Access to a label by name was less frequent than one operation per million (Take Formal Label) indicating that label parameters are infrequently used in relation to their specification. This is probably due to their use in error conditions.

Implicit program jumps can be indicated by the relevant operations as follows

Procedure call	Procedure Entry	20 100 (26)
For loop	For Return	19 300 (18)
Conditional statements and expressions	If False Jump	15 400 (16)

Apart from these operations, the "thunks" start with Block Entry and end with End Implicit Subroutine. With a true compiler, jumps should only be involved with the call by name mechanism. An analysis of parameters is considered in 2.3.1.6.

The Unconditional Jump operation is generated in a number of contexts, mainly as an indirect result of the source text. These contexts are as follows, with an estimate of each frequency.

(i) After Call Block (RR 2.2.7) and so is executed with exactly the same frequency as CBL.

(ii) After Block Entry (RR 2.2.8) in order to jump round procedure and switch bodies. For two or more consecutive procedures there is only one jump, providing no array declarations intervene. Allow about half the CBL count for this.

(iii) Jump round else in conditional statements or expressions (RR 2.1.5 and 2.1.6). The count for this is estimated below, but one would expect a frequency of about half of If False Jump.

(iv) Jump to end of switch declaration after evaluating a switch designator (RR 2.4.2). Executed with the same frequency as Take Switch Address.

(v) Jump round the actual parameter code of a procedure or function call. This is executed with the same frequency as Call Function and Call Formal Function.

(vi) Jump round the address calculation of the control variable in a for loop (RR 2.6.1). This is executed with the same frequency as Call Function Zero in this context. As Call Function Zero can arise in the call of a procedure without parameters, this must be taken into account (see 2.3.1.6).

The table of the use of UJ is now

(i)	197	(51)
(ii)	98	(51)
(iii)	2067	(34)
(iv)	94	(105)
(v)	19 528	(29)
(vi)	2594	(23)
Total	24 578	(20)

The count for (iii) is now seen to be significantly less than half of IFJ. Hence it would appear that conditional statements are more common than expressions and that the **else** clause is often not present. The only alternative explanation is that the condition is usually false, which seems unlikely. Independent evidence of the frequent omission of an **else** clause is given in Appendix 7.

2.3.1.5 For loops

The one pass compilation of for loops is rather difficult because of multiple for list elements and the generality of the **step-until** element as noted in 1.7. In the Whetstone system, the sections of code for the address of the control variable and the constituent parts of the for lists are generated as subroutines terminated with the LINK operation (see RR 2.6).

On initialising the **for** loop, Unconditional Jump, Call Function Zero and For Block Entry are executed. On completion of the controlled statement For Return is executed and on exhaustion of the for list elements For Statement End is obeyed. For each arithmetic element of a for list, two LINK and one FOR Arithmetic operations are executed. For each **while** element three LINKs and one FOR While is obeyed. The **step until** elements are more complex requiring four LINKs and FORS1, on the first execution of the loop and four LINKs and FORS2 subsequently.

The figures reveal as expected that the **step until** element is much more frequently used than the others. Various other facts can be deduced.

The percentage of **for** loops from which an abnormal exit is made can be found from (For Block Entry—For Statement End)/FBE, which is 3.6 (106). It is not possible to find out anything about the structure of the for list elements—this is considered in section 2.3.3.1.1. The average number of times round a **step until** element is (FORS2/FORS1)+1 which is 8.9 (12). This is a number which one would expect to increase substantially for production runs as opposed to the program testing which was monitored. This effect has been observed by D. S. Akka, who found that the number of times round a loop for four classes of programs (divided according to processor time) rose from 5 to 14.

2.3.1.6 Procedure entry

The procedure calling mechanism of ALGOL 60 is quite complex, which has resulted in a substantial difference in the relative speeds of the procedure calling times found in 2.2. In the Whetstone system, the difficulty of the procedure mechanism is reflected in the fact that 56 of the 125 elementary operations are necessary to cater for this mechanism. The wealth of these operations means that the frequency figures give a very clear picture of the use of procedures.

The correspondence between the actual and formal parameters is checked at run-time (RR 2.5.6). Associated with each actual parameter is a “Parameter” operation, and to each formal parameter there is a “Check” operation. The procedure entry mechanism consists of executing each Check operation in turn, which accesses the corresponding Parameter operation and performs the check and takes any necessary further action. For instance, the operation Check and Store Real, checks that the actual parameter is of the correct type (evaluating any thunk that may be involved) and then stores the real value in the address corresponding to the formal parameter. The “Parameter” operation itself is not executed directly and so does not appear in the frequency counts. This means that the best picture of parameters is to be obtained from the Check operations.

The table 2.3, below gives a list of all the Check operations and their frequency. Also tabulated is the “average” number of formal parameters of that check-type; this is given by frequency/procedure entry frequency.

The sum of all the check operations shows that the average number of parameters to procedures is 1.88 (19).

The frequencies of these operations shows a very rapid exponential decline which is typical of much of this data. The operation Check and Store Real accounts for nearly half of these operations whereas five operations were not used in a total of 155 million.

2.3.1.7 Special procedures

Calls of particular procedures can be distinguished even though the Procedure Entry operation is always used. For instance, procedures without parameters are called by Call Function Zero and Call Formal Function Zero instead of Call Function and Call Formal Function. Unfortunately CFZ is also used in for statements, but the call of parameterless procedures can be calculated from (PE$-$CF$-$CFF). This has a mean of 788 (104) operations per million. Expressed as a percentage of all procedure calls, this is 4.33 (108).

The standard functions are handled by the Whetstone System as ordinary procedures, whose bodies are single syllable operations which perform the necessary cal-

Operation	Meaning	Frequency	Variance in brackets	Average number of parameters	
Procedure Entry	Procedure or function call	20 100	(26)		
Check and Store Real	real by value	18 100	(26)	0.922	(26)
Check and Store Integer	integer by value	13 100	(26)	0.644	(17)
Check Arithmetic	integer/real by name	2480	(79)	0.140	(89)
Check STring	string	1790	(76)	0.089	(54)
Check Array Real	real array by name	656	(34)	0.036	(45)
Check Array Integer	integer array by name	471	(180)	0.018	(160)
Check and Store Boolean	Boolean by value	113	(93)	0.006	large
Check Label	label by name	59	(91)	0.003	large
Copy Real Formal Array	real array by value	43	(85)	0.002	large
Check Boolean	Boolean by name	24	(170)	0.001	large
Check PRocedure	procedure	9	(153)	small	
Check Function Real	real procedure	3	(169)	small	
Check and Store Label	label by value	2	(208)	small	
Copy Integer Formal Array	integer array by value	0	(223)	small	
Check Function Boolean	Boolean procedure	not used			
Check Function Integer	integer procedure	not used			
Check SWitch	switch	not used			
Check Array Boolean	Boolean array by name	not used			
Copy Boolean Formal Array	Boolean array by value	not used			

Table 2.3: Procedure Entry and the Check Operations

SQRT	1750	(49)
COS	1490	(101)
ABS	1390	(40)
SIN	1020	(108)
ENTIER	909	(122)
EXP	831	(92)
LN	644	(83)
ARCTAN	591	(149)
SIGN	82	(82)

Table 2.4: Use of the standard functions

culation. Hence the use of each function can be found directly from the frequencies, shown in Table 2.4.

The total for all the standard functions is 8718 (56) which represents 41.8 (37)% of the call of all procedures.

Another class of procedures are those written in KDF9 machine-code. Although such procedures can be written by the user, the most common are undoubtedly the main input-output ones. These are *write* (**integer**, **integer**, **real**), *read* (**integer**), *write text* (**integer**, **string**) where the type are as indicated and are call by value except the string. The Check String operation is probably largely accounted for by the *write text* procedure. At Oxford University, the Whetstone system is somewhat different so that these input-output procedures are treated as standard functions. Hence it is possible to determine the fraction of procedure calls for input-output which was 59%.

Machine code procedures are entered via the operations R DOWN, I DOWN or B DOWN, depending on the type of function or R DOWN in the case of an ordinary procedure. The percentage of these operations in relation to Procedure Entry was

R DOWN	40.7	(38)
I DOWN	4.2	(large)
B DOWN	0.0	(very large)

The use of code procedures and standard functions was 86.8 (9)% of all procedures. This fraction can also be determined from (RETURN−Call BLock)/PE, which gives proportion of procedures in ALGOL (assuming procedures were left by the ordinary mechanism, i.e. not a goto).

2.3.1.8 Use of formal parameters

The information that can be obtained from the instruction counts on the use of formal parameters is uneven and not always easy to interpret. If a parameter is by value, then within the procedure, access to the formal parameter is identical to access of variables declared in the procedure body itself (RR 2.5.5). Hence no statistics can be given on the frequency of access to value parameters. This absence of detail is unlikely to be significant, since access to value parameters is almost always as efficient as access to any declared variable, and so is not performance critical.

Access to name parameters is via a 'Take Formal' operation. The operation Take Formal Address is used for all accesses to a name array, assignments to Booleans (RR 2.5.5.2), and switches (RR 2.5.5.5). In practice the last two cases are so rare in relation to the first that they can be safely ignored. This assumption yields Table 2.5.

Operation	Context	Frequency	
Take Formal Address	all accesses to arrays	24400	(39)
Take Formal Real	real in expression	2690	(40)
Take Formal Address Integer	assignment to integer	1890	(42)
Take Formal Integer	integer in expression	1820	(109)
Take Formal Address Real	assignment to real	1700	(93)
Take Formal Boolean	Boolean in expression	85	(143)
Take Formal Label	label in expression	0	(151)

Table 2.5: Use of formal parameters

The Whetstone system excludes type conversion on assignment to name parameters. Hence the operations TFAI and TFAR, apart from calculating the relevant address, check that the corresponding actual parameter is of the correct type (i.e. as specified for the formal).

It is possible to see how many accesses are made on average to each formal parameter. For instance, the operations TFI, TFAI, TFR and TFAR correspond to a formal parameter which must have been validated by a Check Arithmetic operation. Hence the average number of accesses in this class is 4.46 (19). Ignoring the access to formal Booleans and switches, the ratio TFA/(Check Array Real + Check Array Integer + Check Array Boolean) gives the number of accesses to arrays. This is much higher at 29.4 (52). These figures indicate that the programmers are aware of the crucial difference between name arrays (which are efficient) and simple variables by name (which can be very inefficient because of the thunk mechanism).

Only particular formal parameters can have actual parameters which are expressions and the proportion can be calculated. Each thunk consists of a Block Entry operation terminated with a corresponding End Implicit Subroutine (RR 2.5.4.3). BE does occur in blocks (RR 2.2.2) and both operations occur in switch declarations (RR 2.4.2), but the relative frequency of these occurrences is very small and so can be safely ignored. The number of parameters involved can be found from the sum of Check and Store Integer, Check and Store Real, Check and Store Boolean and Check and Store Label for the value parameters and the sum of TFI, TFR, TFB, TFL, TFAI and TFAR for the name parameters. The ratio of this sum to BE gives the fraction (on the basis of execution) of parameters which are expressions as opposed to a simple variable or constant. This is 35.2 (16)%. This percentage shows how worthwhile it is evaluating such expressions in the pre-call sequence rather than using a thunk.

2.3.1.9 Average array dimension

The access to every array variable involves the evaluation of each subscript. This is followed by one of the two operations INDex Address or INDex Result (RR 2.3.2). Unfortunately these operations do not indicate the array dimension, since this is determined by the state of the stack. However, each operation does have an effect upon the run-time stack in Whetstone Algol in the manner indicated by the table in Appendix 6. The only operations to have an indeterminate effect on the stack are INDA, INDR, Make Storage Function and Make Own Storage Function. Assuming the average array dimension for these operations is the same, and that users did not regularly jump out of functions, etc., this average dimension can be calculated by equating the net effect on the stack of all the operations to zero. This gives an average dimension of 1.2. A static analysis of array declarations gives an average dimension of 1.26 (see 2.3.3.1.2). D. S. Akka found an average dimension of between 1.25 and 1.44.

2.3.1.10 Arithmetic variable store and fetch

Of particular importance is the proportion of real to integer fetches and stores and also the proportion of array element accessing to simple variables. A very approximate analysis of this is possible, by assuming that the “average” program has a simple structure.

Since Boolean variables (as opposed to relational operators) are used comparatively rarely, only integer and real quantities are considered. The operation Take Formal Address is not considered explicitly, but an allowance is made for it by increasing the Take Real Address and Take Integer Address counts in this analysis. The fact that INDex Result is used in switches is ignored.

The use of the elementary store and fetch operations in the **for** loop control code is rather different from elsewhere, so an explicit allowance is made for this use, by assuming that almost all for loops are of the form **for** $i := 1$ **step** 1 **until** n **do**. In such cases, TIA and TIR are used once for every FORS2 operation.

The use of integers in subscript expressions can be calculated on the basis of an average array dimension of 1.26 (see last section) and an assumption that each subscript involves a TIR. The use of TIA and TRA to fetch array words can be removed by subtracting the counts for INDA and INDR.

This crude analysis gives a use of simple and array variables as 329 000 operations per million which can be broken as in Table 2.6.

	Use	Frequency and Variance	
	Integer	57	(7)
	Rea	43	(9)
	Fetch	77.3	(2)
	Store	22.7	(6)
(1)	Integers as subscripts	24.5	(6)
(2)	Integers in loop control	16.0	(17)
(3)	Integer array fetch	2.8	(20)
(4)	Integer array store	1.8	(29)
(5)	Real array fetch	9.2	(9)
(6)	Real array store	5.6	(13)
(7)	Integer fetch excluding 1 and 2	9.4	(24)
(8)	Integer store excluding 2	2.3	(18)
(9)	Real fetch	20.7	(17)
(10)	Real store	7.6	(16)

Table 2.6: Variable usage

2.3.1.11 Average length of an expression

The average length of an expression can be calculated by an analysis of the stack depth of the interpretive code. This is worked out in a very similar way to the average array dimension calculation given in 2.3.1.9. The sum of the effect on the stack is calculated by removing the operators and store instructions. This total sum is then divided by the total for the store instructions. A slight adjustment is made so that a program consisting entirely of $x := y$ gives the answer 1 (this is done by removing the total for STA from the load operations total). The average calculated this way comes out at 22, so a "typical" expression contains only two identifiers or constants as in $a[6], x + y$ or $i \div 30$. This strongly suggests that very clever optimisation by some compilers is not worthwhile (see also Knuth, 1971a).

2.3.2 WORK OF D. S. AKKA

D. S. Akka has produced a similar dynamic analysis to that given above for Whetstone ALGOL[1], and this is summarised below.

The dynamic analysis was produced from Atlas Autocode (Brooker, 1966), which is sufficiently close to ALGOL to allow reasonable comparisons to be drawn. Counts were obtained for 26 different language features by incrementing various registers in the monitored program. This required an additional instruction to be inserted in the compiled program for each of the language features monitored, a fairly simple operation since the compiler contained a routine for each such language feature. These routines were modified by the compiler loading program in order to generate the additional instructions.

The programs analysed were split into four groups according to the execution time of the program. This allows one to note whether there is any consistent trend in the counts depending on program length. One such trend has been mentioned in that the

[1]This analysis has not been published although the thesis containing all the material is available from the University of Manchester (Akka, 1967).

average number of times round a loop increases from 5 to 14. The other trend is that the frequency of use of the facility to output strings and to declare arrays decreases with the length of the program.

Apart from the output of strings and the loop control code, a number of other language features have close parallels. Array declarations, procedure calls, standard function calls, array accessing, assignment statements and the use of switches can all be compared with the Whetstone figures. Apart from the three trends given above, the only consistent differences with the ALGOL figures is a higher use of procedure calls and switches. The higher use of switches in Atlas Autocode is not surprising because of the greater use of labels and the less powerful conditional statement facility. An interesting additional statistic was the proportion of subscript expressions which were not linear, which varied from 2.7% to nearly zero.

Together with the collection and analysis of these counts, Akka produced a test program which used the language features in the same proportion as was found from the 200 programs that were monitored. The program was produced in a long and short form, and the short form was run on a number of different computers as a benchmark (see 2.5).

2.3.3 STATIC ANALYSIS OF SOURCE TEXT

In any system where the source text of programs is held within the system, analysis of source text by program is particularly fruitful. Although one is ordinarily interested in reducing the execution time, in many applications the length of the object code is very critical. Compiler writers could also make use of a static analysis to attempt to reduce compilation times.

On KDF9, source text is held on a fixed disc store in an internal ALGOL basic symbol code. For this purpose, the 116 ALGOL basic symbols are extended by the editing characters, space, tab and newline and also by some additional compound symbols needed for the system namely, **KDF9**, **ALGOL**, **EXIT**, **segment**, **library** and $\rightarrow$. The symbols **KDF9** and **ALGOL** are used to bracket the body of procedures written in machine-code (whose text is including in-line) **library** is used for a special macro facility for library text, and $\rightarrow$ terminates a file. The other symbols are not relevant to the present note.

Appendix 7 lists the frequency of KDF9 ALGOL basic symbols as found from 200 programs. The program ignored symbols appearing between **KDF9** and **ALGOL** and between **comment** and ;. The other two types of comment were not ignored (**end** comment and parameter comments). Almost any sequence of ALGOL basic symbols can occur in strings, but in practice only the simple characters are found.

A number of points are worth noting from the figures. The digit frequency follows the logarithmic distribution function (Knuth, 1969, p. 220), whereas the alphabet follows a modified lexicographic distribution. Although the programs are not necessarily correct, they are ‘nearly’ so, but the difference between the counts for ‘(’ and ‘)’ may be a program error. All the operators +, $-$, $\times$ occur with much the same frequency as in the dynamic case (remember unary minus counts as NEGate). In fact, in many cases in ALGOL basic symbol is uniquely related to an elementary operation or set of operations. Hence it is possible to calculate a ratio between the dynamic and static uses of a particular language feature. This allows one to assess which features are worth optimising for code space as opposed to execution time.

Other points to note are that **else** is much less common than **if**, and that the relational operators are more common than **Boolean**.

Symbol	Static count	Dynamic count	Operation	Ratio
:=	20 600	49 100	ST, STA	2.4
[	15 400	64 900	INDR, INDA	4.2
$-$	7400	31 500	$-$, NEG	4.3
$\times$	7000	36 200	$\times$	5.2
+	7000	31 000	+	4.4
if	4140	15 400	IFJ	3.7
for	3500	2700	FBE	0.8
step	3400	2330	FORS1	0.7
=	2800	6980	=	2.5
/	2500	11 000	/	4.4
goto	1460	2010	GTA	1.4
$<$	770	3290	$<$	4.3
$>$	740	3600	$>$	4.9
$\neq$	720	2230	$\neq$	3.1
$\uparrow$	480	3530	$\uparrow$	7.4
$\wedge$	360	884	$\wedge$	2.5
$\vee$	260	2180	vee	8.4
$\leq$	230	1030	$\leq$	4.5
$\geq$	220	914	$\geq$	4.2
$\div$	200	481	DIV	2.4
string	106	1790	CST	17.0
true	102	78	TBCT	0.8
false	100	124	TBCF	1.2
$\neg$	85	115	$\neg$	1.3
while	31	377	FORW	12.0

Table 2.7: Dynamic to static ratio

	Number	Percentage
One **step-until** element	780	94.7
list of expressions	26	3.2
containing **while**	7	0.8
other forms	11	1.3

Table 2.8: Overall structure of loop

A further program was written to analyse the length of identifiers and digit sequences. The length of identifiers (on use, including declaration) is distributed roughly on a negative exponential so that there are half as many identifiers of length $n + 1$ as n. The digit sequences were similarly distributed but with a local maximum around the machine accuracy of 11 decimal places.

2.3.3.1 The use of particular basic symbols

Some basic symbols occur much less frequently than once per line. In order to analyse the use of such symbols, a program was written to scan a number of program texts, printing any line containing the given symbol. The program was used with the four symbols, **for**, :, $\uparrow$ and $\div$.

2.3.3.1.1 For loops A total of 824 **for** loops were examined from 37 programs. The results are most easily summarised by means of a series of tables.

Since only one line of text was printed, in some cases not all the symbols between **for** and **do** were output. This happened only with lists of expressions which were too long for a line, in which case it was assumed that the list continued without a **step-until** or **while** element. For similar reasons, the types of variables used in the for loop were not known although the simple variables were thought to be integers except in two or three cases.

The only type of loop worth further consideration is the **step-until** element. The most important characteristic is the nature of the step and limit (see 1.7.2). The 803 **step-until** elements were classified into groups by the complexity of the three expressions. Three degrees of complexity were used, viz.

(i) <constant> = explicit constant, possibly with sign

(ii) <sv> = simple variable

(iii) <other> = expression but not (i) or (ii).

The following classification is shown in Table 2.9.

These figures emphasise the necessity of optimising **step** 1 and -1.

2.3.3.1.2 Array bounds Lines containing the colon were printed out to determine the values of the array bound pairs in array declarations. Labels were also printed (found to be very largely of the form l digit. . .) as were parameter comments.

A total of 272 bound pairs were found from 47 programs. Of these 137 had both bounds constant (possibly signed). The lower bounds were as follows:

lower bound = 0	57 times
= 1	208 times
other constant	6 times (in fact, always -1)
other	once

		Number	Percentage
(A)	<constant> **step** 1 **until** <sv>	414	51.6
(B)	<sv> **step** 1 **until** <sv>	69	8.6
(C)	<other> **step** 1 **until** <sv>	69	8.6
(D)	<constant> **step** 1 **until** <other>	66	8.2
(E)	<constant> **step** 1 **until** <constant>	55	6.8
(F)	other loops with **step** 1	9	1.1
(G)	<sv> **step** -1 **until** <constant>	20	2.5
(H)	<other> **step** -1 **until** <constant>	34	4.2
(I)	other forms with **step** -1	36	4.5
(J)	other forms with **step** constant	8	1.0
(K)	loops without constant **step**	22	2.7
	(A) to (F) that is, with **step** 1	682	84.9
	(G) to (I) that is with **step** -1	90	11.2

Table 2.9: For loops

The dimension distribution was 161, 54 and 1 for one, two and three dimensions, giving an average dimension of 1.26. The number of arrays declared using the same bound-pair list was distributed as follows:

Number in bound pair	Count
1	127
2	31
3	24
4	10
5	4
6	5
7	4
8	6
9	1
13	1

2.3.3.1.3 The exponential operator A total of 524 $\uparrow$ symbols were found in 76 programs. 416 of these symbols did not appear to be in comments, or after the final **end** (this denotes that data for the program follows the source text). These operators were used as follows:

$\uparrow 2$	269 occurrences
$\uparrow 3$	21
$\uparrow 4$	3
$\uparrow 5$	7
$\uparrow$ other	101 occurrences

Unfortunately it is not possible to tell with the 101 other cases the type of the exponent, i.e. integer or real, or that of the base. This is very significant in view of the difference in the algorithm used (see 1.1.4.2).

2.3.3.1.4 Integer divide This was comparatively rare compared with the last three symbols. Only 63 occurrences were found in 146 programs. These can be categorised as follows:

division by constant	29
test to see if variable is divisible by a constant, as with $i = i \div 4 \times 4$	25
general	9

This clearly indicates the necessity of having a further operator giving the remainder on integer division directly. Open code could often be generated for integer division by a constant since the test for the sign of the arguments is unnecessary (see 1.1.4.1). Also, arithmetic shifts could be used for division by powers of two, but this seems hardly worthwhile in view of its rarity.

2.3.3.2 Array subscripts

Because of the importance of array subscripts, a special program was written to analyse them. The program was written using a version of the Compiler-Compiler (Brooker, 1967). This did a left to right parse of a subscript list classifying it into one of nine different categories. Each individual subscript was classified as either an integer constant (C), a simple variable (V, presumably an integer) or something more complex (?). At the same time, the number of array segments was counted, as was the total number of ALGOL basic symbols.

30 programs were scanned, containing 117 array segments and about 86000 ALGOL basic symbols. The counts for each class was as follows:

	Class	Count	Note
1	[V]	865	
2	[C]	470	
3	[?]	132	anything not type 1, 2 or 4 to 9.
4	[V, V]	120	
5	[C, ?]	51	two or more dimensions but excluding types 8 and 9.
6	[V, C]	39	
7	[V, ?]	33	two or more dimensions but excluding types 4 and 6.
8	[C, C]	20	
9	[C, V]	17	
	Total	1747	

The program did not analyse the actual dimension frequency because on meeting a complex subscript of more than 2 dimensions the parsing stopped. Assuming that each individual subscript is independently distributed between V, C, and more complex forms, then the ratios are roughly 8:4:1. To this should be added the figure of Akka that between 2.7% and nearly 0% of subscripts involved a non-linear expression (see 2.3.2). This analysis of the non-linear nature of subscripts was necessary in the Atlas Autocode compiler because of the register instructions of Atlas (see 9.3).

2.3.4 STATIC ANALYSIS OF COMPILED CODE

The difficulty with the analysis of the source text is that many of its relevant characteristics cannot be determined without a process similar to compilation. The type of variables, the use of subscripts, formal-actual correspondence of parameters cannot be analysed without a name list. Writing half a compiler merely to get such information is very time consuming. However, the binary program produced by the compiler can be analysed to yield much useful information. The binary code usually has a simple structure which means that an analysis program is correspondingly easy to write. With the KDF9 system, the output from both the compilers has been analysed, and this is described in the next two sections.

2.3.4.1 The Whetstone compiled code

One of the Whetstone ALGOL systems used at NPL writes the interpretive code to backing store for later execution. This code can be read off the backing store with the aid of a small machine-code subroutine. The interpretive code can then be "anti-compiled" into the form corresponding to the printed code as given in Randell and Russell's book. That, is the statement $x := y$ appears as Take Real Address n, Take Real Result m, STore, where n and m are the addresses of x and y respectively. Totals for the number of uses of each elementary operation were produced on each anti-compilation.

40 programs were analysed using the anti-compiler, but only the total number of operations for each program was printed. The anti-compiler worked at about 120 elementary operations per second or about one third of the speed of the compiler itself. This meant that this static analysis was substantially more expensive in computer time than the dynamic counts used in section 2.3.1. Hence a much smaller sample of programs was taken. The main result was the static frequency per thousand of the elementary operations which is listed in Appendix 6.

The anti-compiler may be of interest in itself. It was written in ALGOL, and makes a single pass over the code. Each elementary operation is of the form

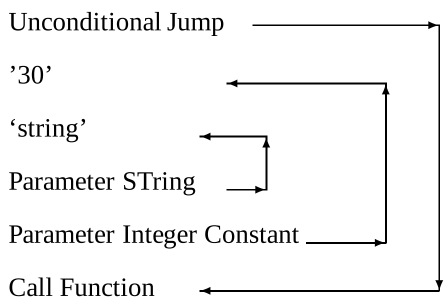
<function> <operand>

where the function is 8 bits and the operand is a multiple of 8 bits which is determined by the function. So a decoding table used by the anti-compiler consists of the size of the operand and the text for each function. The complication comes with procedure parameters where explicit constants and strings are stored with the code and so must be avoided. This can be arranged by inspecting the relevant parameter operations which precede the Call Function or Call Formal Function operations.

Example

The ALGOL *write text*(30, 'string')

is compiled as



So the action of the anti-compiler is as follows. Each operation is decoded in a simple manner until an Unconditional Jump operation is met. Simple decoding continues unless this points to a Call Function or Call Formal Function. In the case of a function call a backward scan is made of the parameter operations (the number of parameters is in the operand part of CF or CFF, so the length of the scan is known). When the parameter operation is a Parameter Boolean Constant, Parameter Real Constant or Parameter Integer Constant the six syllables of the constant are skipped over. If the operation is Parameter STring, then a skip is made to the next whole word, the string is printed in ALGOL basic symbol form, and another skip to the next whole word is made to get the anti-compiler back in step. The most complex case is when the parameter operation is Parameter SubRoutine. Then a sequence beginning with Block Entry and ending with End Implicit Subroutine must be decoded. Since this can include another function call, various variables used in decoding the outer call must be stacked, and then unstacked by the End Implicit Subroutine. Note that End Implicit Subroutine can occur in switches and in this case no unstacking is necessary (as the stack depth is then zero!).

Apart from the basic frequencies given in the appendix, the following information was produced. An average program consisted of: 6.05 disc blocks or about 23 000 ALGOL basic symbols; Interpretive code was 1030 words (of 48 bits), containing 1934 operations of 3.3 syllables (on average).

The program contained

2.5	blocks
2	standard functions
4	ALGOL procedures
30	machine-code procedures

(containing about 1100 words of code).

2.3.4.2 The Kidsgrove compiled code

The output from the Kidsgrove ALGOL compiler is binary machine-code, except that calls on the permanent control routines are put in a special form. To run a program, the machine-code is relocated into core by adjusting the internal addresses and those of the entries to the control routine, as well as references to the stack.

Although anti-compilation of the binary machine-code is possible, the code cannot in general be easily related to the structure of the source text. Entries to the control routine are an exception in that the correspondence to various program constructs is clear. More than half of the total number of entries relate to input-output and are not listed here (see Table 2.10). 200 programs were scanned on magnetic tape, and the total number of calls of each entry to the control routine was listed. A “program” can also be a “segment” which is an independently compiled procedure, having a slightly different structure in that no initialisation or termination code is produced.

2.4 An ALGOL Performance Measure

The analysis of the statement times given in section 2.2.4 gave a speed factor for various implementations. This was not a true performance measure because the relative frequency of use of the various parts of ALGOL was not taken into account. The statistics given in section 2.3 now enables the individual statements to be weighted to give

Total use	Meaning
20 759	Assignment overflow check, produced for each left part list.
9847	Float, integer to real conversion. This is often called unnecessarily (see 9.4).
5905	Two dimensional array access; this has subsequently been replaced by open code in most cases.
3296	Thunk end, used to return control after the evaluation of a call by name parameter.
3009	Evaluate thunk. Used to calculate the address or value of the call-by-name parameter.
1918	(real) ↑ (integer). Type checking is performed at compile time so that the various cases of this operator are distinguished.
1796	Procedure entry and exit. Used when the procedure is not "simple" (see 9.4).
1796	(ditto)
1501	Make storage function. Used once for each array segment, including **own** arrays.
1381	Fix, real to integer conversion. Probably the majority of uses are for reading integers, since the standard procedure is a **real** function.
879	Assignment trace routine. Alternative to the first entry in this table, now the alternative is set at load time.
742	Evaluate formal thunk.
559	3 or more dimensional array access.
373	Increment calculator. Used to set up a register for subscript optimisation. Only called if the optimiser is requested (for about 10% of programs).
268	(real) ↑ (real).
257	Calculate number of times round for loop (used only when optimiser is requested).
212	Integer divide.
201	(integer) ↑ (integer).
164	**goto** l where l is a label parameter.
157	Invalid address thunk. This is called if an attempt is made to assign to a call-by-name parameter which is actually a constant or an expression.
56	Evaluate switch, used for each **goto** $s[l]$.
52	Switch declaration.
50	Copy value array. Called once for each array by value.
28	Call formal procedure.
12	Fix and check integer. Used when dynamic type checking cannot be avoided (see 1.1.4.2.).
6	(integer) ↑ (real)
0	Three entries are concerned with access to variables in nested procedures. A call would only appear if the outer procedure were recursive and an inner one accessed variables (see 9.4.).

Table 2.10: Static frequency of control routine entries

a more realistic measure. The Whetstone ALGOL dynamic counts are used to give the weights for each statement, but other figures are used where appropriate. Since the Whetstone figures are divided into seven sets, this is repeated here (giving an average and variance).

The basic technique is to expand the simple statements into their constituent elementary operations. Weights must how be assigned so that the resulting frequency of each elementary operation matches that given in Appendix 6. Although the method of arriving at the weights appears to depend on a particular implementation, it is clear that the basic statistics reflect the use of ALGOL 60, rather than the idiosyncrasies of Whetstone ALGOL. Fitting the weights to the observed frequencies could be regarded as a simple minimisation problem to be solved by linear algebra but a few difficulties arise with this method. Unfortunately many elementary operations do not appear in the statements, and in other cases insufficient information is available to assign values to the weights. Hence direct calculation of the weights used in section 2.4.2.

2.4.1 EXPANSION OF THE SIMPLE STATEMENTS

Listed in Table 2.11 is each simple statement expanded into the elementary instructions as given in Randell (1964). Only the short form of the instructions is given. The code in brackets represents executed instructions placed apart for the statement itself.

2.4.2 CALCULATION OF THE WEIGHTS

For the purposes of this section, the 42 weights are denoted by an **array** $wt[1 : 42]$. Many of the weights can be determined from single operations, for instance

wt[6] = 11000	(27)	from /
wt[11] = 481	(70)	from DIV
wt[27] = 2010	(51)	from GoTo Accumulator
wt[28] = 94	(105)	from Take Switch Address
wt[29] = 1020	(108)	from SIN
wt[30] = 1490	(101)	from COS
wt[31] = 1390	(40)	from ABS
wt[32] = 831	(92)	from EXP
wt[33] = 633	(83)	from LN
wt[34] = 1750	(49)	from SQRT
wt[35] = 591	(149)	from ARCTAN
wt[36] = 82	(82)	from SIGN
wt[37] = 909	(122)	from ENTIER
wt[42] = 17800	(21)	from FORS2

The $\uparrow$ operator appears in the three statements numbered 15, 16 and 17. Hence the count for $\uparrow$ must be split between these three. Figures for arriving at a split are available from 2.3.3.1.2 and 2.3.4.2, although these are static counts. These give a factor of seven between an integer exponent and a real one. Also $\uparrow 2$ appears to account for about 90% of the use of $\uparrow$ (integer). Hence a reasonable split between the three is 315:35:50 giving

wt[15] = 2780	(59)
wt[16] = 309	(59)
wt[17] = 442	(59)

Statement number	Statement	Code
1.	x := 1.0	TRA x, TRC‘1.O’, ST.
2.	x := 1	TRA x, TIC1, ST.
3.	x := y	TRA x, TRR y, ST.
4.	x := $y + z$	TRA x, TRR y, TRR z, +, ST.
5.	x := $y \times z$	TRA x, TRR y, TRR z, $\times$, ST.
6.	x := y/z	TRA x, TRR y, TRR z, /, ST.
7.	k := 1	TIA k, TIC1, ST.
8.	k := 1.0	TIA k, TRC‘1.O’, ST.
9.	k := $l + m$	TIA k, TIR l, TIR m, +, ST.
10.	k := $l \times m$	TIA k, TIR l, TIR m, $\times$, ST.
11.	k := $l \div m$	TIA k, TIR l, TIR m, DIV, ST.
12.	k := l	TIA k, TIR l, ST.
13.	x := l	TRA x, TIR l, ST.
14.	l := y	TIA l, TRR y, ST.
15.	x := $y \uparrow$ 2	TRA x, TRR y, TIC‘2’, $\uparrow$, ST.
16.	x := $y \uparrow$ 3	TRA x, TRR y, TIC‘3’, $\uparrow$, ST.
17.	x := $y \uparrow z$	TRA x, TRR y, TRR z, $\uparrow$, ST.
18.	$e1[1]$:= 1	TIA $e1$, TIC1, INDA, TIC1, ST.
19.	$e2[1, 1]$:= 1	TIA $e2$, TIC1, TIC1, INDA, TIC1, ST.
20.	$e3[1, 1, 1]$:= 1	TIA $e3$, TIC1, TIC1, TIC1, INDA, TIC1, ST.
21.	l := $e1[1]$	TIA l, TIA $e1$, TIC1, INDR, ST.
22.	**begin real** a; **end**	CBL, UJ, BE, RETURN.
23.	**begin array** $a[1 : 1]$; **end**	CBL, UJ, BE, TIC1, TIC1, MSF, RETURN.
24.	**begin array** $a[1 : 500]$; **end**	CBL, UJ, BE, TIC1, TIC’500’, MSF, RETURN.
25.	**begin array** $a[1 : 1, 1 : 1]$; **end**	CBL, UJ, BE, TIC1, TIC1, TIC1 , TIC1, MSF, RETURN.
26.	**begin array** $a[1 : 1, 1 : 1, 1 : 1]$; **end**	CBL, UJ, BE, TICI, TIC1, TICI, TICI, TIC1, TIC I , MSF, RETURN.
27.	**begin goto** $abcd$; $abcd$: **end**	TL $abcd$ GTA, TRACE $abcd$.
28.	**begin switch** s := q; **goto** $s[1]$; q : **end**	CBL, UJ, BE, UJ, BE, DSI, TL pq, UJ, ESL, EIS, TSA ss, TIC1, INDR, GTA, TRACE pq, RETURN.
29 to 37.	$x := standard\ function(y)$	TRA x, UJ, PR y, CF $standard\ function$, ST. (TRACE $standard\ function$, PE, CSR, STANDARD FUNCTION CODE)
29.	x := $sin(y)$	
30.	x := $cos(y)$	
31.	x := $abs(y)$	
32.	x := $exp(y)$	
33.	x := $ln(y)$	
34.	x := $sqrt(y)$	
35.	x := $arctan(y)$	
36.	x := $sign(y)$	
37.	x := $entier(y)$	
38.	$p0$	CFZ $p0$, REJECT. (TRACE $p0$, PE, RETURN)
39.	$p1(x)$	UJ, PR x, CF $p1$, REJECT. (TRACE $p1$, PE, CSR, RETURN)
40.	$p2(x, y)$	UJ, PR x, PR y, CF $p2$, REJECT. (TRACE $p2$, PE, CSR, CSR, RETURN)
41.	$p2(x, y, z)$	UJ, PR x, PR y, PR z, CF $p3$, REJECT. (TRACE $p3$, PE, CSR, CSR, CSR, RETURN)
42.	loop code in **for** $i := 1$ **step** 1 **until** n **do**	FORS2, TIC1, LINK, TIA, LINK, TIR, LINK, TIA, LINK, FR.

Table 2.11: Expansion of the simple statements

In section 2.3.3.1.2 the observed ratio of array declarations between one, two and three dimensional arrays was 161:54:1. So the weights for the array declaration statements can be found by splitting the Make Storage Function frequency between these, viz.

wt[23] = 59 (49) one dimensional ones
wt[24] = 59 (49) divided equally
wt[25] = 39 (49)
wt[26] = 0.73(49)

Unfortunately this means that Call Block is oversubscribed since the ratio of MSF to CBL is 1.25:1, which cannot be preserved by the statements. For this reason it would seem best to put the weight for statement 22 to zero.

wt[22] = 0 (0).

The weight for statement 38 is determined from the Call Function Zero that arises from parameterless procedure calls. As CPZ also occurs in for loops, this must be taken into account (see 2.3.1.7). This gives

wt[38] = 788 (104).

The weights for the other procedure call statements can be obtained by matching the Procedure Entry and Check operations. Since only Check and Store Real appears in the statements, it seems reasonable to match the total count of all the check operations, that is, the average number of parameters. Of course, PE and CSR have already appeared in the standard function calls and so only the remaining weight need be assigned. This gives two conditions on the three weights, so arbitrarily take the weights for statements 39 and 40 to be the same. This gives

wt[39] = 2316 (87)
wt[40] = 2316 (87)
wt[41] = 6053 (28).

The statements involving INDex Address and INDex Result (18 to 21) cannot be handled very well, since it is impossible to keep both the INDA/INDR and dimension ratio's correct. So the total INDA and INDR count is split amongst the four statements to keep the dimension ratios correct. Taking the same weight for statements 18 and 21, one has

wt[18] = 23 796 (7)
wt[19] = 15962 (7)
wt[20] = 296 (7)
wt[21] = 23796 (7).

To assign weights to the statements 4, 5, 9 and 10 one requires an estimate of the proportion of real and integer use of the operators + and $\times$. The estimate used is based upon the analysis given in 2.3.1.10. The categories 3, 4 and 8 in Table 2.6 are taken as an estimate of the use of integers, and 5, 6, 9 and 10 as that for reals. This gives

wt[4] = 26694 (16)
wt[5] = 31 212(12)
wt[9] = 4300 (23)
wt[10] = 4978 (17).

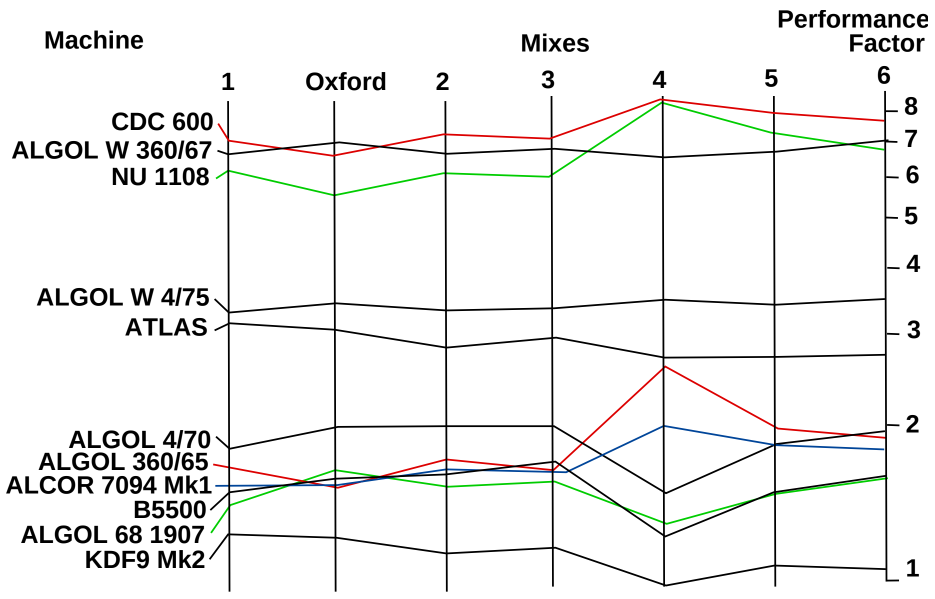


Figure 2.1: Different ALGOL mixes

The eight remaining statements are not characterised by any of the elementary operations since they only involve store and fetch operations. In fact, the store and fetch instructions are already overweighted, because the 42 statements are somewhat shorter than is typical of most programs. On the other hand, about 10% of the elementary operations remain unaccounted for. The total discrepancy of about 40 000 is therefore divided amongst the remaining statements.

From the static counts of FIX and FLOAT subroutines in Kidsgrove ALGOL (2.3.4.2) it is clear that integers appear fairly frequently in real expressions, whereas the converse is much less common. So arbitrarily take the following

wt[l] = 10000
wt[2] = 7000
wt[3] = 10000
wt[7] = 3000
wt[8] = 500
wt[12] = 5000
wt[13] = 4000
wt[14] = 500.

The statement weights and the corresponding mix values are listed in Appendix 8. These figures are most easily appreciated by means of a graph in Fig. 2.1. Each mix is represented by a vertical logarithmic scale. The position of each computer on the scales is joined by a broken line. If each mix were in complete agreement, then the broken lines would be horizontal. One can see at a glance that Mix 4 is somewhat anomalous. This is caused by the smallness of the sample and the fact that the standard functions are used very heavily in comparison with user defined procedures. Systems which bypass the standard procedure calling mechanism for the standard functions therefore do very well on Mix 4.

2.4.3 COMPARISON OF THE MIX FIGURES

The mix figure can be compared directly with the machine factors given in 2.2.4. The differences are due to unevenness in the implementations. Some systems have been notably successful in the ALGOL organisational statements—principally block and procedure entry, whereas others have not. If the procedure entry figures are poor, and the standard procedure entry mechanism is not used for input-output, then the mix figure could be unfavourable. However the central processor time spent in input-output cannot be meaningfully compared because there is no widely used standard for input-output. This is unfortunate since this is by far the most important feature not measured by the mix. On KDF9, many small programs spend over half of the processor time executing the basic input-output subroutines.

In those cases where there is a substantial difference between the two ALGOL measures, the residual matrix can be used to locate the reasons. For instance, the IBM 360/65 level F compiler is degraded from 1.03 to 0.65 times Atlas. The residual matrix reveals that the procedure and block entry times are the major statements which are markedly worse than the values predicted by the model.

The mix figure can be compared with other measures of processor performance that are in common use. A highly appropriate one is the Gibson mix (Central Computer Agency 1971). This measure is not precisely defined, but various consistent sets of figures are available. This basic method is to assign weights to each type of machine instruction, fixed point add, floating point multiply, etc. The rate at which the “average” instruction can be processed on the machine is taken as the Gibson mix figure. A major difficulty is that of allowing for the difference in machine architecture—for instance, no credit is given for having more registers in the processor. Nevertheless, it does represent a tolerable measure of processor performance, and is a substantial improvement on just taking the cycle-time, or the like. One might expect the ratio between the Gibson mix and the ALGOL mix to be about constant—or rather to represent a measure of the efficiency of the compiler. The ratio is, in fact, far from constant, mainly due to the varying amounts of effort put into the ALGOL systems. Although it is certainly true that some computers are much more suitable for the production of good code than others, the effect of machine architecture seems very slight. A similar study with FORTRAN would almost certainly be more in line with the basic processor speeds, and would no doubt show the effects of the computer architecture (see Bryant, 1968).

The disparities between the three computer measures are illustrated by the Fig. 2.2. Each measure is on a vertical logarithmic scale in a similar manner to the previous figure. There is no fixed correspondence between the scales—so they have been arranged to minimise the apparent discrepancies. Naturally a computer like the B5500 is better at ALGOL than the Gibson mix would indicate, whereas the CDC6600 is worse (see 9.5 and 9.7).

Given a hardware specification, it would be possible to work out an ALGOL mix figure without a compiler. It is merely necessary to work out the machine-code which one would expect the compiler to produce from the 42 statements. The hardware specification can be used to calculate the statement times, from which the mix figure can be obtained. Obviously no extensive optimisation should be assumed of the compiler, and the code production must be done with sufficient thoroughness to be confident that the code from the 42 statements will be as stated. This technique could be used to validate computer design studies.

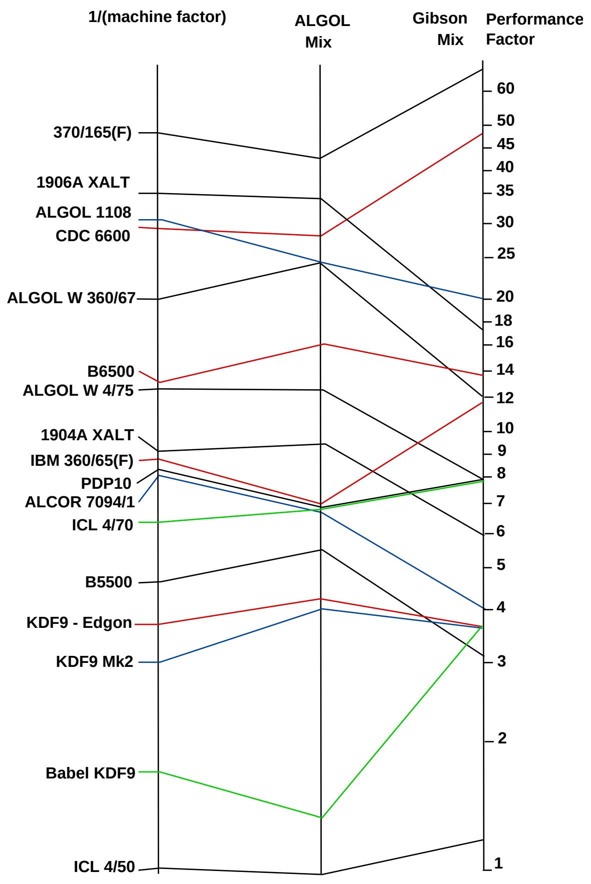


Figure 2.2: Three performance measures compared

2.5 Benchmarks

Benchmarks are a very popular method of measuring computer performance. The essential technique is to use the performance of a single program (or fixed set of programs) as a measure of overall performance. This may be satisfactory with substantial production programs, but is unlikely to take into account the large variation encountered in many major installations. Important system parameters like the number of jobs processed per day, ratio of compilation to execution, degree of multiprogramming and timesharing are likely to have a very critical effect on overall performance. Even if one restricts ones attention to the processing speed of ALGOL 60, a benchmark always gives a one-sided picture. The rate of calling procedures, entering blocks, accessing array variables, etc., in a single program cannot be expected to be typical of the whole job mix in a system.

The difficulties and dangers of using a single benchmark are very well illustrated with the GAMM measure (Heinhold, 1962). This measure is the length of time taken to perform simple numerical calculations-adding a vector, multiplying two vectors, finding the largest element of a vector, calculating a polynomial by Horner's method, and calculating a square root by Newton's method. This algorithm can be programmed in any language, and the ALGOL 60 version is considered in detail in section 9.1. It is clear that this program makes very heavy use of one dimensional arrays and simple for loops. Substantial gains can be made with simple optimisation of such loops. However the test makes no use of procedures or blocks, two dimensional arrays, integer arithmetic or the standard functions. Hence the test is obviously not even typical of numerical work and so it would be most unwise to use it as a performance measure.

One method of overcoming the above deficiency of benchmarks is to write it in a number of modules, and weight each module according to some known statistics. The ALGOL mix given in 2.4 could have been constructed this way. D.S. Akka produced such a benchmark using his own statistics from Atlas Autocode. This seems a very reasonable approach although great care must be taken with the construction of the benchmark to ensure that extensive optimisation of the program is not any easier than is likely to be the case with user programs. For instance, the constant subscript optimisation which invalidated the array accessing figures in the ALGOL mix, illustrates this difficult.

Chapter 3

Estimating the Execution Speed of an ALGOL Program

Resulting from the analysis of the timing data are very good estimates of the relative execution times of certain simple statements (see 2.2.4 and Appendix 5). These averages allow one to construct a machine-independent estimate of the time that a particular program or algorithm should take. The estimates can be made by hand with reasonable ease, so that one can determine which of a number of possible alternatives is likely to be the fastest. In practice, this method need only be applied to the innermost loop of a program.

The basic method is to give a weight to each identifier, constant or delimiter of the program. This weight is machine/compiler independent and represents an average based upon a number of different implementations. The units used correspond very roughly with one machine instruction time, which is about 3μs on Atlas, the times for other machines being in proportion to the ALGOL mix figure (Appendix 8)

Each rule which assigns a weight is numbered (e.g. R7). Examples are then given which refer to this number so that the reader may verify the examples himself. In changing a program to make it execute faster, it is important to remember that the clarity of the text should not suffer inordinately. A clear program which can be easily understood is better than a faster obscure one.

Throughout this chapter examples are given of short pieces of ALGOL coding together with the execution weights. Unless otherwise stated, the declaration of the variables is as follows:-

real	$x, y, z;$
integer	$i, j, k, n;$
array	$x1[1:10], x2[1:10, 1:10], x3[1:10, 1:10, 1:10];$
integer array	$i1[1:10], i2[1:10, 1:10], i3[1:10, 1:10, 1:10];$

3.1 Arithmetic expressions

The technique is easily illustrated by considering arithmetic expressions.

operator	integer operands	real operands
unary minus[*] $+, -$	1 . . . R3	1 . . . R4
$\times$	3 . . . R5	2 . . . R6
/	not applicable	4 . . . R7
$\div$	6 . . . R8	not applicable
$\uparrow$(integer) $n = abs$(integer)	11+5n . . . R9	11+5n . . . R10
$\uparrow$(real)	not applicable	70 . . . R11
[*] A unary + can obviously be ignored.		

Table 3.1: Operator times

Access to simple and array variables (but excluding some formal parameters: see section 3.7)	1 unit	R1
Use of constants	1 unit	R2

The operators are most conveniently given in Table 3.1.

In the course of the execution of an arithmetic expression type conversion is sometimes implied. The allowance for this is as follows:

real to integer	4 units	R12
integer to real	1 unit	R13

Array variable access is allowed for by weighting square brackets and the commas separating the subscript expressions. With all brackets, only the opening one is assigned a weight.

[	3 units	R14
, (separating subscript expressions)	3 units	R15

Round brackets in expressions are given a zero weight. Although compilers may be forced to generate code for storing partial results, but this is not simply related to the bracket structure (see 1.1.5). In practice the number of instructions used to preserve partial results is not a significant percentage and can reasonably be ignored in this simple estimation process.

Some examples can now be given

- $x + y \times z$
  (R1, R4, R1, R6, R1) = 6 units.
- $x/y - i$
  (R1, R7, R1, R4, R13, R1) = 9 units, note the type conversion
- $x \uparrow 2 + y \uparrow 2$
  (R1, R10 (n=2), R2, R4, R1, R10 (n=2), R2) = 47 units.

whereas $x \times x + y \times y$ gives 9 units, although the use of $\uparrow$ is much to be preferred from the point of view of program clarity. $x +_{10} -6 - 3.8_{10} - 8$ is 5 units as the sign in the exponent is part of the constant.

3.2 Assignment statements

A zero weight is given to the := symbol ...R16

$x := x + x1[i + j]$
(R1, R16, R1, R4, R1, R14, R1, R3, R1) = 10 units

and

$y := i + j \times k$
(R1, R16, R13, R1, R3, R1, R5, R1) = 9 units

Note the addition for the type conversion.

It is easy to see from this that if a subscripted variable is accessed more than once without changing its value, then it is worthwhile doing $xi := x1[i]$ and then referencing xi. However the saving with only two references is marginal and so as a general rule 3 or more references to a subscripted variable are worth replacing. The above assumes that the compiler does not perform the equivalent optimisation.

Similar remarks apply to reducing the dimension of array accessing. The assignments to subscripted variables can sometimes be moved outside the **for** loop resulting in an even greater saving.

For instance

```
for $i$ := 1 step 1 until $n$ do
for $j$ := 1 step 1 until $n$ do
for $k$ := 1 step 1 until $n$ do
    $x3[i, j, k] = x2[i, j] \times x1[i]$;
```

takes $43n^3 + 20n^2 + 20n + 6$ units whereas

```
for $i$ := 1 step 1 until $n$ do
    begin
    $xi := x1[i]$;
    for $j$ := 1 step 1 until $n$ do
        begin
        $xijk := x2[i, j] \times xi$;
        for $k$ := 1 step 1 until $n$ do
            $x3[i, j, k] = xijk$;
        end
    end
```

takes $28n^3 + 33n^2 + 26n + 6$ units (see 3.5 for details).

3.3 Relational operators and Boolean expressions

For the symbols

=, $\neq$, $>$, $<$, $\geq$, $\leq$	allow 2 units	…	R17
for **true false**	allow 1 unit	…	R18
for $\neg$, $\wedge$, $\vee$	allow 1 unit	…	R19
for $\equiv$, $\supset$	allow 2 units	…	R20

In conditional statements and conditional expressions

if	is given zero weight	…	R21
then	is given 2 units	…	R22
else	is given 1 unit	…	R23

Hence one has

$i < j$
(R1, R17, R1) = 4 units
$x1[i] \leq x1[j]$
(R1, R14, R1, R17, R1, R14, R1) = 12 units
$max :=$ **if** $x > y$ **then** x **else** y
(R1, R21, R1, R17, R1, **then** R22, R1, **else** R23, R1.
= **if true then** 8 **else** 7 units.

In calculating these weights the tacit assumption is made that the compiler does not perform extensive optimisation; for instance, in the above example neither x nor y is kept in a register. A similar effect to that optimisation can be achieved by the use of slave stores.

3.4 Declarations

In general, declarations require no execution time, but entry (and exit) to a block takes 10 units … R24

An array declaration does, of course, require time. The expressions giving the bounds must be evaluated and the time this takes can be calculated from the rules given in 3.1. Apart from this, the following must be added:

[after list of array identifiers	35 units	R25
: separating the bounds	30 units	R26
, separating bound pairs	0 units	R27
, separating array identifiers	10 units	R28

Own arrays are excluded from these weightings since they are more complex and not executed sufficiently frequently to be worth detailed analysis. In any analysis, the interpretation given to own arrays with dynamic bounds is not clear (see 7.9).

The weights given imply that it is much more efficient to place arrays with the same bounds in the same array list. There is also a corresponding economy in code space which may well be more significant. The time saving is illustrated by

array $a, b[1 : n]$ takes 79 units whereas
array $a[1 : n], b[1 : n]$ takes 136 units.

Of course, the **begin** — **end** pairs a compound statement and the semicolon separating declarations and blocks generate no code and are assigned a zero weight. . . . R29

3.5 For loops

The complexity of the ALGOL 60 for loop mechanism is naturally reflected in the time taken to execute the more elaborate forms. In practice, the following two cases are usually adequate:

> **for** $i := 1$ **step** 1 **until** n **do**; requires 6 + 14n units and
> **for** $i := 1, 2, 6, 8, 19$ **do**; requires 10 units for each assignment.

In the more complex cases, each type of the for list element must be calculated separately. For this, it is assumed that the control variable is a simple variable which is not a name parameter. If a subscripted variable is used as the control variable, the number of times the address calculation is done could be significant. This can vary significantly with the interpretation put upon this part of the ALGOL report—in any case, it is of little practical significance (see 7.4).

For each type if for list element we have:

(A) <a>
For this allow expression <b> + 8 units R33

(B) <a> **while** <b>
For this allow for each time round the loop expression <a> + expression <b> + 8 units . . . R34

(C) <a> **step** <b> **until** <c>
Initial assignment and test expression <a> + 4 units . . . R31

For each time round the loop expression <b> + expression <b> + 12 units . . . R32

No explicit allowance need be made for the ‘,’ separating the for list elements.

As an example, consider two apparently equivalent ways of writing a loop:

> **for** $i := i \div j \times j$ **step** -1 **until** 0 **do**;

takes 17 + 15 (no. of times round loop).

> **for** $i := 0$ **step** 1 **until** $i \div j \times j$ **do**

takes = 16 + 25 (no. of times round loop).

These simple formulae do not assume any extensive optimisation of loop control, and are, of course, based upon an average of existing compilers most of which do some optimisation for **step** 1 or **step**-1.

3.6 Procedure and function calls

In this section the time taken to call and exit from a procedure or function is considered. The body of the procedure is a statement and so can be estimated using the rules given elsewhere. The rules giving estimates for the access to formal parameters are given in the next section.

The estimate is made up as follows

procedure identifier 25 units	R35
parameter bracket (12 units	R36
“,” or “) letters : (” separating parameters 8 units	R37

This covers the basic calling mechanism, but an additional allowance must be made for the actual parameters as follows:

Actual parameter	Estimate	
string	1 unit	R38
expression	if parameter is called by value then calculate as for expression; otherwise allow 3 units	R39
array identifier	1 unit if called by name If called by value allow 100+2n units where n is the total number of elements in the array	R40
switch identifier	3 units	R42
procedure identifier	3 units	R43

In the case of the standard functions, it is possible to give an allowance for time which includes the procedure body. The time required for the execution of the expression parameter is excluded.

sin( )	65	R44
cos( )	65	R45
abs( )	3	R46
exp( )	65	R47
ln( )	58	R48
sqrt( )	50	R49
arctan( )	83	R50
sign( )	7	R51
entier( )	15	R52

3.6.1 EXAMPLES

- $x := sin(y) + cos(y + z)$ is 136 units.
- $x := sum(x1, i)$, where the first parameter is by name, the second by value. This comes to 48 units plus an allowance for the body of the procedure *sum*.

The important points to note are that value arrays are very slow in the call, and that there is a significant overhead for each parameter. Simple variables called by name are handled quite quickly in the entry to the procedure but access to the formal parameter is very slow (see below).

3.7 Access via Formal Parameters

Parameters which are specified as being by value are accessed as ordinary variables. In the compiled code, value parameters and variables local to the procedure are usually indistinguishable.

Parameters which are called by name count as follows:

array	1 unit	R55
string	1 unit	R56
procedure (ie, a call)	25 units	R57

These are the same as for ordinary access. However, in the following parameters, the actual parameter must be evaluated by the thunk mechanism (at least in general, see 1.10).

real	25	R58
integer	25	R59
Boolean	25	R60
label	12	R61
switch	12	R62

The use of the function designator on the left hand side of an assignment statement is given one unit. . . . R63

Note that assignment to the procedure identifier is substantially faster than assigning to a name parameter. This is because the procedure identifier assignment is handled as an assignment to a local variable (which is write-only). So for procedures which produce a result, it is substantially more efficient, and often as convenient to make it into a function designator. Also, with one less parameter, the call of the function will be quicker.

It is clearly important to remove access to name parameters involving a thunk from the inner loop of the program.

3.8 Labels, designational expressions and switches

A **goto** statement should be allowed 2 units if the resulting label is local to the block in which the statement appears. In other cases, including formal labels, 12 units should be allowed. . . . R53.

For a switch, allow 30 units for the switch identifier and subscript brackets. If the resulting element of the switch is more complex than a simple label, then an additional allowance must be made in the same way as for a designational expression. . . . R54.

In calculating the estimates, allow zero for the **goto** symbol and give the weight to the corresponding table. Hence:

goto if $x > y$ **then** *exit* **else** $ss[i + j]$;

= 8 units if true and or 38 otherwise (assuming all labels are local).

3.9 Use and accuracy of the estimation

Given a reasonable mix of instruction types, the estimate of the time taken can be expected to be within 20% of the actual time. Larger discrepancies will occur if the statements being timed consist entirely of procedure calls or array subscripting etc. In such cases, the residual matrix given in Appendix 3 will indicate whether the particular implementation in question is relatively strong or weak in the relevant language feature. If a recoding of a particular sequence gives an estimated saving of 10% or more, then an actual saving should be assumed.

For this method of estimation to be practical tool with large programs, it must be combined with a knowledge of the frequency of execution of the various parts. This problem of dynamic flow analysis is considered in section 6.3. Assuming such information is available, the inner loops of a program can be coded in different ways to obtain a gain in processor time. Because of the way the weights are constructed, the gains should be general to most ALGOL systems.

It would obviously be convenient if the estimation was automated by a computer program which worked in a similar method to a compiler. A side-by-side listing of the source text, the processor estimates, and the dynamic flow analysis would also be a very powerful tool in program tuning. This has been produced for FORTRAN (Ingalls, 1971). Such tools are really essential for vetting of algorithms for publication. Performance details of published algorithms are very scarce and brief. To say that a sort took 3 seconds on “X” is often more informative about the ALGOL compiler for X than the particular sorting algorithm in use. Unfortunately, the structure of the software industry is such that very few significant items of software are published and are subject to such scrutiny. Nevertheless such analysis of programs and systems is imperative if software engineering is to become comparable with other branches of engineering.

3.10 Tabular Summary of the Estimates

Weights are given to the basic symbols of the ALGOL text, or to identifiers and numbers in the case of letters and digits. In general, an identifier has a weight of one unit (rule number one: R1). The weights summarised as in tables 3.2 and 3.3.

The other cases are summarised as follows:

	Weight	Rule
procedure identifier	25	R35
(as actual parameter)	3	R43
(for standard functions see R44 to R52)		
array identifier	1	R40
(as actual parameter, called by value)		
	100+2(number of words in array)	R41
switch identifier		
(followed by simple subscript in designational expression)	30	R54
(as actual parameter)	3	R42

Identifiers appearing in the specification part of a procedure body can be ignored. For formal parameters by name see R55 to R62. <number> has a weight of one unit (R2) and may contain a sign in the exponent which is not an operator.

Although declarations can, in general, be ignored, allowance must be made for both array declarations and the consequential entry and exit from the block (R24= 10 units).

The other weights can be determined from the basic symbols. The only exception being the real-integer type conversion that is implied by some expressions.

Basic symbol	Weight		Rule
+	as unary operator	0	
	with integers	1	R3
	with reals	1	R4
$-$	all cases	1	R3,R4
$\times$	with integers	3	R5
	with reals	2	R6
/		4	R7
$\div$		6	R8
$\uparrow$ (integer = n)		11+5n	R9,R10
$\uparrow$ (real)		70	R11
$<\ \leq\ >\ \geq\ =\ \neq$		2	R17
$\neg \wedge \vee$		1	R19
$\equiv\ \supset$		2	R20
goto	see 3.8	0	
if		0	R21
then		2	R22
else		1	R23
for	see 3.5	0	
do	see 3.5	0	
,	(separating subscript expressions)	3	R15
,	(separating bound pairs)	0	R27
,	(separating array identifiers in declarations)	10	R28
,	(separating for list elements)	0	
,	(separating parameters in procedure calls)	8	R37
,	(other uses in declarations, procedure heading and specification)	0	

Table 3.2: Basic symbol weights, Part 1

Basic symbol	Weight		Rule
$\cdot_{10}$	only appear as part of a real number		
:	(after label)	0	
:	(separating bound pairs)	30	R26
;		0	R30
:=		0	R16
_	Only appear in strings		
step while until	see 3.5		
comment	(and symbols of comment)	0	
(	(in expressions)	0	
(	(in procedure call)	12	R36
(	(in procedure heading)	0	
)		0	
[	(in subscripted variable)	3	
[	(in array declaration)	35	
]		0	
‘	(plus symbols of string)	1	R38
‘		0	
begin	(of block)	10	R24
begin	(of compound statement)	0	R29
end	(ignore **end** comments)	0	
own	not considered		
Boolean integer real		0	
array switch procedure		0	
string label value		0	

Table 3.3: Basic symbol weights, Part 2

Chapter 4

Syntax Analysis

This is undoubtedly the most highly developed area in compiler construction. Detailed theoretical work on the description and analysis of grammars has had a very beneficial effect upon the definition of ALGOL 60, which has rarely been equalled. Fortunately many compiler-writing systems have provided a practical realisation of much of this work (see Feldman, 1968). This means that errors in syntax analysis of ALGOL 60 should not occur, because semi-automatic methods exist for coding a recogniser directly from the description given in the Report (see Foster, 1968; 1970, Cohen, 1970). Unfortunately direct coding from the Report is not common-usually because some restriction in the syntax analysis method precludes this. When hand coding is used, the chances of error are quite high because of the size of the syntax and the fact that it contains a number of peculiarities.

This book concentrates mainly on the semantics of ALGOL 60, and this chapter does not attempt to cover the syntax completely. The next two sections illustrate the common pitfalls in syntax of ALGOL 60, which have been found in a number of implementations. This only serves to emphasise the necessity of direct coding from the Report, supported by more thorough validation tests (see chapter 8).

4.1 Common Syntactic Errors

Syntactic errors can broadly classified into two types. Those which do not effect the semantics, and the more serious ones which do. Even with these more serious errors, the effect that the error will have upon the code generated by the compiler is rarely in doubt.

4.1.1 ERRORS IN SYNTAX ONLY

One peculiarity of ALGOL 60, is that comments are not easy to spot and remove and this is unfortunate, since any program, apart from a compiler, which analyses ALGOL 60 source text will almost certainly want to skip over comments. The difficulty arises because there are three different forms of comment all of which are defined syntactically. Consequently unless a complete syntax analysis is performed it is hard to be sure that the comments have been successfully handled. One cannot help thinking that this is a weakness in ALGOL 60, especially since there are so many places where comments are desirable but specifically excluded.

The three forms of comment are the explicit form starting with **comment** and finishing with ;, the parameter comment which is really another way of writing a comma separating formal or actual parameters, and the **end** comment. They all have peculiarities. The explicit form can only follow **begin** or ;, and so *label*: **comment** …; is invalid. The parameter comment, apart from being of such limited context, can only contain letters (for no obvious reason). The **end** comment is the most dangerous form, since it is the least explicit. It is terminated by **end**, **else** or; and in consequence it is rather easy for a statement to be swallowed up in an **end** comment through the omission of a semicolon.

A further point to remember about comments is that virtually any sequence of ALGOL basic symbols can appear in a string—which must obviously be preserved by the compiler. Nevertheless comments can be removed fairly simply without a complete syntax check provided parameter comments are replaced by a special basic symbol 'parameter comment', so that a check can be performed later when the complete syntax analysis is made.

Another minor pitfall is that there is a dummy statement but no dummy declaration in ALGOL 60. If, for instance a semicolon immediately follows a **begin**, then that constitutes a dummy statement and hence no declarations can follow.

One oddity is concerned with own arrays. With real arrays, one can omit the **real** on the declaration, but this is not permissible with own real arrays. This is one of the few cases where a syntax error is sufficiently precise for the compiler writer to contemplate correcting the source text automatically.

4.1.2 ERRORS EFFECTING SEMANTICS

There are a number of complications in the syntax of ALGOL 60 which are required to avoid certain grammatical ambiguities. Several of these difficulties are due to the optional **else** with conditional statements (see Naur, 1960; Abrahams, 1966). If, for instance, a conditional statement appears after **then**, then it would not be clear which **if** was matched to a single **else**. The syntax prohibits this construction (Report 4.5.1), but several ALGOL 60 compilers allow it (B5500 and B6500). So with

```
if $b$l then
    if $b2$ then $s1$
    else $s2$;
```

it is not clear if $b1$ is true and $b2$ false whether or not $s2$ will be executed. In practice every compiler which allows this, matches the **else** with the immediately preceding **if** (so in the above $b1$ is true and $b2$ false to execute $s2$).

A similar, slightly more complex case arises with the **for** loop. Consider the following

```
if $b1$ then
    for $i := 1$ do
        if $b2$ then $s1$
            else $s2$
```

This is excluded because only an unconditional statement can follow **then** which does not include a **for** statement (Report 4.5.1). A few compilers have allowed this even though the simpler error was trapped (Egdon-KDF9, System 4).

A very wide range of "syntax" errors arises indirectly through the application of the copy rule (Report 4.7.3.2). If a name parameter appears on the left-hand side of an assignment statement, then the corresponding actual parameter can only be a simple or subscripted variable. If the actual parameter is a simple variable preceded by a + sign, or is enclosed in some redundant brackets, then it is invalid. Even with these minor changes, the actual parameter becomes an expression, and so the name parameter can only appear in contexts requiring an expression. For the reasons stated in 1.10, few compilers spot these errors, so that the necessary checking is performed at run-time.

Syntax errors tend to persist in those parts of the language which are rarely used. Few programmers realise that labels can appear immediately after **then**, **else** and **do**. Also, the **goto** statement is very general in the sense one can jump into a compound statement (but not a **for** statement). Checking for the restriction with the **for** statement is rather awkward since this is not simply a matter of scope. With the Whetstone ALGOL system, the authors overcame this by making the **for** statement into a block even if it contained no declarations.

Even though the syntax analysis of the Atlas ALGOL compiler used the Compiler-Compiler it contains an error. A dummy statement cannot appear between **then** and **else**. Again, in most programs this sort of error can remain dormant for years, unless systematic checks are made as suggested in chapter 8, or the coding is done directly from the Report.

4.2 Output from the Syntax Analysis

Depending upon the design of the compiler, the output from the syntax analysis phase can be very similar to, or a long way from the target machine-code. Several one-pass compilers for ALGOL 60 exist, and these, of course, must produce the machine code almost immediately. With ALGOL 60, one pass translation is quite a severe restriction which obviously precludes various forms of optimisation. In this section multi-pass compilation is considered leaving to section 7.10 the discussion of one-pass compilation.

Given that the output of the main syntax analysis pass is not machine-code or a very near derivative, then an important design problem arises. The output will necessarily reflect upon both the syntax analysis and the subsequent pass. One criteria may be that the output is to be machine-independent in order to have a portable first pass. This seems a very reasonable aim which has been achieved in at least one system. In the production of anyone particular compiler there is a great temptation to put additional information in the output to avoid complications in the subsequent pass or passes. This may make the output machine dependent, or at least more so.

One very practical form of output is a reverse polish string. This is easy to produce, but not necessarily ideal for code generation. One possibility is to output the interpretive code used by Randell and Russell, a substantial advantage being that the required processor is very well defined in their book. Also, apart from a complete syntax check, the translator does a large number of other checks, almost all that is reasonable in one pass. The major disadvantage is that it leaves unresolved many questions of type-checking. For instance, the assignment of **true** to an integer is not trapped until execution. More critical is the question of type-checking of procedure parameters. It has already been seen that this checking must be performed at compile-time if reasonable rapid procedure calls are to be made.

So, apart from the reverse polish string or its equivalent, various tables must be produced to allow type checking and related issues to be resolved. Identifiers and line numbers (or an equivalent) must also be preserved for diagnostic purposes. The line numbers can be inserted into the program string with some suitable convention, while the tables are most reasonably produced for each block upon its completion.

A complete syntax analysis is often represented pictorially by a tree structure. Some compilers produce this tree structure in a direct form—for instance, the Compiler-Compiler (Brooker 1967). A substantial difficulty of this technique is that a large part of the storage of the structure is taken with pointers to its remaining parts. Such pointers are implied in the case of reverse polish string as they are in the original source text. Hence methods have to be devised to condense the information if it is to have anything but a transitory existence. This can make the technique both inconvenient and slow. With the Atlas ALGOL compiler Kerr and Clegg obtained a tree structure for the major program parts but used a condensed form of the text at basic statement level (Brooker 1967). The condensed text was then re-expanded in bits as required for code generation.

Allocation of storage within the syntax analysis program can produce problems. The identifier information grows and contracts with the static block structure of the program. This is very convenient since a simple stack is adequate. There is difficulty with storage of information concerned with each identifier. With a procedure, the specification of each parameter needs to be stored so that efficient code can be generated for each call. This information is not necessarily available on the first use of the procedure and even the amount of storage that it requires may not be known.

When any degree of optimisation is attempted then various auxiliary tables are required. This can all too often result in arbitrary limits upon the complexity of source text acceptable to the compiler. For instance, the Kidsgrove compiler limits the number of procedures to 96, and the length of a switch list to 64 elements. To use a list storage scheme to overcome these problems is not necessary and can have other disadvantages. The simplest solution is probably that given by Knuth (1968) for sharing a single core area between a number of arrays. In many cases, it is important that complex language constructs should at least be compiled even if the code is not optimal. To arrange this, special action must be taken when a table overflows, or when a restriction required for optimisation is violated.

4.2.1 INFORMATION REQUIRED FOR CODE GENERATION

Apart from the program string and identifier tables, various additional information is required for code generation. The amount of this information will depend on the degree of optimisation attempted, and its nature will reflect the machine architecture. Some of these points have been mentioned above but a list is given here of the major requirements. Extensive optimisation will certainly require more than the items listed below (see chapter 10).

For each block

(1) Does it contain arrays?
(2) Does it contain procedures?
(3) Extent of first order working store.
(4) Number of anonymous variables required.
(5) Does it contain a label (or switch) used formally?
(6) Extent of own storage.

For each identifier

(1) Block level.

(2) Address.
(3) Type.
(4) Dimension or number of subscripts.
(5) Position of first use for diagnostics (warning if not used).
(6) Assignment made (if appropriate, print warning if none made).

For each **for** loop
(1) Number of for-list elements.
(2) Control variable is a name parameter or subscripted variable.
(3) Control variable is real or integer.
(4) For each **step** element, an indication if the increment is constant.

For an outer-most expression or inner-most **for** loop:
Does the construct involve a name or function call?

Additional information is required with some types, for instance

For each procedure
(1) Can it be called formally?
(2) Parameter specification.
(3) Array dimensions if known (see 7.1).
(4) Assignment made to name parameter?
(5) Extent of first order working store.
(6) Procedure global?
(7) Procedure only called at declared procedure level?

For each array
(1) Dimension.
(2) Bounds constant? If so, size of array and bounds.

For each label and switch
(1) Used formally?
(2) Used other than local to block?
(3) Switch “simple”?

Chapter 5

Diagnostic and Debugging Aids

5.1 General Techniques

Almost all programs contain errors, the only difficulty is to locate them. The degree of checking performed by ALGOL systems varies enormously, and in virtually no case can a system be called ideal. An interesting intercomparison of the debugging capability of a number of existing systems is given in Scowen (1972). This chapter is divided into three main sub-headings corresponding to the principle methods or error detection. Firstly, a compiler should check the syntax and part of the semantics of the source text. It is important that intelligible error messages should be produced at every stage, in order that the programmer can make the necessary correction. Secondly, errors can be detected at program execution time. On an execution error, much valuable information about the program state is available in the main memory. The third subheading describes methods of producing good crash-time error information.

5.2 Compile-time Diagnostics

A compiler should be an excellent example of how to handle data-input. Data should be completely validated, and in all erroneous cases an intelligible message produced. The final arbiter as to the intelligibility of the message is the programmer and not the compiler-writer. As the average ALGOL 60 programmer is certainly not very familiar with the Report, technical jargon should be avoided.

One piece of information is of paramount importance and that is the position of the "error" in the source text, which should be given in terms readily understood by the programmer. If a line-number based editor is used, then obviously line numbers are appropriate, otherwise printing out the offending line marking the position of error within the line is a reasonable method. Sometimes two positions in the text are relevant to an error, for instance, with an inconsistent use of an identifier the declaration or previous use may be at fault rather than the current line under suspicion. It is most important that positional information should not be "lost" in subsequent passes of a multi-pass compiler. Hence the error printing routine must be carefully designed to produce just the information that is required. In many cases, compilers print such things as "current identifier", "current delimiter", etc., regardless of whether they are relevant to the actual error which has been detected. A manual must be consulted if only an error number is given, as is usually the case for non-overlaid compilers working in a minimum of core-store. In most other cases, text can be produced at very little overhead. A better technique is to put the text in coded form where one particular code might be interpreted as "print current identifier" and another code is a shorthand for "the current delimiter is", etc. In this way, a small piece of additional coding in the error routine can lead to a compact form of the error messages which are themselves self-explanatory.

One difficult problem is recovery from an error during the syntax analysis stage. A fairly simple strategy is to skip to the next semicolon and attempt to continue. During the skipping one can miss a **begin** and consequently get badly out of step. Also, if the compiler has found an error in a declaration it may miss further declarations making it virtually impossible to recover. The Whetstone compiler, being essentially one pass, makes declarations on each use of a variable within a block. This has the advantage that if a declaration is missed it will not get out of step, in fact the error will not even be detected until the syntax analysis is complete. Also, only one error message is produced, not one for each use of the offending identifier. Almost all compilers seem to get hopelessly out of step in certain circumstances even though the relevant piece of text is "locally" good ALGOL 60. This suggests that radically different techniques than those used currently should be employed on error recovery. A line-by-line syntax checker for ALGOL 60 described by Haddon (1971) appears to be very much more successful with programs containing several errors. The importance of recovery from an error depends upon the turn-round of the user. With on-line use, detecting only the first error is probably adequate. With batch execution, every attempt should be made to recover, otherwise development of large programs becomes virtually impossible.

Unfortunately it is very much easier to give examples of bad error handling by compilers than good ones. One system, although it always gives a textual message, on an error of any complexity merely says "trouble near line n". The fact that this proves to be adequate in most cases shows the importance of positional information. The Kidsgrove ALGOL compiler produces one of two different types of message. If it fails in the first pass, accurate positional information is given by printing the neighbouring text. Unfortunately failures on subsequent passes—which are necessarily more difficult for the programmer to detect, give no indication of position. This situation is only tolerable because almost all testing is done with the Whetstone interpretive system. The Elliott 4100 series ALGOL compiler has a rather unsatisfactory method of giving positional information by printing a few characters of the source after the actual error. This is obviously easy to implement but is certainly not so convenient as far as the programmer is concerned.

It is not easy for the compiler-writer to devise succinct phrases for error messages. Even if the programmers could be expected to know what a "left part list" or a "switch designator" is, it is hard to avoid some technicalities. Some systems make few concessions to the user with such phrases as "comma matches wrong push down configuration" or "generator nest with more than one result". Clearly, technical terms not in the Report should be avoided, especially those introduced merely because of the compiling method.

The volume of printed output produced by compilers differs greatly and is in most cases, controlled by various parameters. Often error messages are linked to a source text listing produced by the compiler, so if this listing is not required, then the positional indication of error may not be adequate. This may seem a small point, but it can prohibit the use of a compiler in a multi-access environment when slow printing devices are being used. The Whetstone compiler has proved to be ideal for multi-access use not only because of the high compiling speed but also because so little printed output is produced.

Type-checking errors found during subsequent passes of a compiler can be very complex. Devising good messages is difficult because the “error” may be due to a number of causes. Take the case where integer-real type conversion is not allowed with call-by-name parameters if an assignment is made. It is likely that the programmer does not realise that his program violates this restriction or perhaps he does not know what the restriction means. To alter his program, an actual parameter, a formal specification or an assignment could be changed. Ideally all three positions in the program text should be included in the error message, together with the two identifiers. The requirement to give good error indications in these cases imposes a substantial burden on the compiler. All identifiers and positions of source text code must be preserved merely for the handling of errors, although such information can be kept on secondary storage. One interesting possibility is if file input of the source text is used. The source text can then be rescanned upon an error, and the offending lines printed out.

The printing of error messages other than the first is not so important since there is a high probability that the compiler may have got out of step, or missed a declaration, etc. Hence there is a good case for giving the first error preferential treatment—although all compilers known to the author treat them identically. It is quite important that compilers should give up completely at some stage. The Whetstone compiler stops after 20 errors which seems reasonable—although a smaller limit for multi-access use would be appropriate. Cases have been reported of a compiler attempting to translate a data file, the result being a 100 pages or more of diagnostic output.

Some features of ALGOL 60 are particularly error-prone and deserve special handling by a compiler. For instance, if a closing string quote is missed from a source text, a large part of an ALGOL program will be ignored. Printing a warning if certain symbols like **begin** or) etc. appear in a string, or if the string exceeds a certain length, are possible solutions. In fact, some implementations restrict the length of a string which has the very beneficial effect of getting the compiler back in step under such circumstances. Another dangerous feature of ALGOL is the **end** comment. A missing semicolon can mean that an entire statement is skipped by the compiler. In an attempt to avoid this, the Whetstone compiler prints a warning if any delimiter appears in an **end** comment. This has proved to be very effective, although on a few occasions the warning has not been heeded by a programmer. It is a most unfortunate fact that warnings such as these, and advice of a similar nature is all too often ignored by inexperienced programmers.

The Eindhoven X8 ALGOL compiler makes some very interesting attempts to overcome the above problems. The symbols **begin** and **end** are not allowed in comments and strings, making it easier for the compiler to recover from a syntax error. Also, the program text must be terminated by an explicit symbol **progend**, so that there is no risk of reading beyond the source text. This symbol must appear at the end of the program ensuring that no source text has been omitted due to too many **end**s. This technique was also used in the Whetstone compiler (Randell, 1964, p. 364). Positional information with the Eindhoven compiler also follows Whetstone in giving an absolute line number and one relative to the last label or procedure passed. A delimiter count within the line pinpoints the error very precisely. A good textual indication of the error is also given (in Dutch).

Warning messages are produced by compilers in a variety of circumstances. For instance, on request, the Whetstone compiler produces a warning for the use of each overlength identifier, the reasoning behind this being that if a programmer declares two variables *autocorrelation 1* and *autocorrelation 2* in nested blocks, then the program would compile but execute incorrectly. Unfortunately the compiler produces a message for every overlength identifier, not just those which are the same on the significant characters. Hence the output is too voluminous to be effective. The Trondheim 1108 compiler produces a warning if constants are compared, i.e. **if** 2 = 2 **then**, etc. This seems very astute, although one suspects that if a programmer writes this, then he has good reason, although it could be caused by a simple typing error.

5.2.1 COMPILING OPTIONS

Almost all compilers have several different options which control the two main aspects of the compiling process. Firstly options control the information about the program which is printed by the compiler. Typically this covers such things as a source listing, a memory map, and machine code listing of the generated code. These options and similar facilities are considered in section 5.5. Secondly there are usually options which control the nature of the machine code generated. This varies from code to perform elaborate checks—subscripts, overflow, stack space, etc., to code which may even take liberties with the program in order to be very efficient.

The main difficulty with options for testing at compile-time is that recompilation may be necessary. Even if the compiler is fast, separate binary programs will be required for production running. In a large number of cases, options can be delayed until load time. This alternative is largely dependent on the operating system facilities and is discussed in section 5.5.

Additional run-time checking may require recompilation because the code generated is radically different. Examples of this are:-

(1) array bound checking of subscripts

(2) overflow

(3) stack space checking

(4) use before assignment

(5) tracing.

Subscript checking is quite awkward unless hardware assistance is available (see 1.2.6), and thus on the majority of third generation machines subroutines must be used.

Overflow is often an interrupt condition, but some machines do require explicit code for testing overflow (KDF9). In such cases, an option must be provided, otherwise the volume of checking code produced is likely to be prohibitive.

Stack space checking is performed in most systems by the control routines on entry to a block or a procedure. However, it should be possible to enter blocks and procedures without the use of subroutines (see 1.8). In such cases, it is necessary to check that there is enough stack space. If a violation of the stack leads to a known recoverable condition, then checking may not be necessary, otherwise the checking code may have to be inserted as a compiling option. There are a few difficulties in this, as an allowance must be made for in-line use of the stack. For instance, if parameters are planted directly where required as recommended in 1.9, then calling a procedure with a large number of parameters could violate the stack unless special care is taken. On KDF9, the stack always has at least 16 cells free on every procedure call and array declaration. Nevertheless, an occasional program does violate the stack causing unpredictable effects.

One very useful check requiring special coding is used before assignment, which is difficult to achieve without something amounting to interpretation, unless special hardware is available. This is really a run-time check and is discussed in the next section. Tracing can be done at various levels, and in almost all cases requires different codes to be generated.

5.2.2 COMPILE-TIME CHECKING

In the first chapter, many cases were mentioned where the computer should resolve various issues at compile time in order that efficient code should be produced. It is also important from the point of view of debugging. An infrequently used part of a program may contain an error, and if this can be detected at compile time it can be corrected rapidly while the programmer is still actively involved with the program. With a run-time check, the error may be dormant for some time, causing substantial difficulty when it becomes apparent.

The issues which should be resolved at compile time are:-

(1) Type checking of integer, real and Boolean variables.

Statement such as i := **true** (i integer) should be rejected, $x := i+j+y$ should be compiled to produce optimal calls of the type conversion subroutines or macros. This really demands complete specification of parameters, and suitable conventions with exponentiate (see 1.1.4.2).

(2) The number of parameters to procedures and subscripts to arrays should be checked.

The difficulty with some obscure constructions is considered in 7.1.

(3) Formal-actual parameter checking should be performed at compile-time, so that open code can be produced for value parameters (see 1.9).

Again some obscure features may have to be deferred to run-time (see 7.1 and 7.2).

(4) If an assignment is made to a name parameter, then it should be checked that at least one actual parameter is consistent with this (and produce a warning for any others?).

The checking of an invalid assignment to a name parameter cannot be avoided at run-time, but if it is virtually certain that such an assignment will be attempted this should be noted. Compilers which do not allow integer-real type conversion with assignment to name parameters should check this at compile-time. This check is usually such that if there is any assignment (or possible assignment via a name parameter), then the formal and actual parameters must agree in type.

(5) If possible, the formal-actual correspondence of parameters to formal procedures should be checked. In its full generality, it is virtually impossible to check this at compile time, but practical uses of formal procedures are often sufficiently simple for it to be accomplished (see 7.2).

5.3 Run-time Checking

With a perfectly designed language, a correctness proof could be established for a computer program by a compiler. Under such circumstances, run-time checking could be avoided altogether (except for hardware checking). No programming language has reached this level of perfection, so run-time checking is necessary for as many as possible of those errors which cannot be checked at compile time. There is no very rigid distinction between an error and bad programming practice—in some cases the Report explicitly states something to be an error, whereas in other cases no reasonable meaning can be attached to a language construction. It is important that as many aspects of a program should be checked as possible so that the programmer can concentrate on the major logical problems of program construction. The main areas on run-time checking are surveyed below.

5.3.1 SUBSCRIPT CHECKING

This is a difficult problem because of the various methods of checking and the substantial effect it has upon execution speed. If no special hardware is available for bound checking then programs can run at half the speed if bound checking is performed. This issue radically affects the methods used to generate code for array handling and was considered in detail in 1.2.6. It will be seen that the code generated for subscript checking is very different from that without, so this must usually be covered by a compiling option. Even within one system, several methods may be feasible as follows

(1) checking each subscript separately
(2) checking on store but not fetch
(3) no checking
(4) use of open-code or subroutines.

The Electrologica X8 ALGOL compiler was modified at Petten to do subscript checking only on store (as an option). This reduces the overhead of the check substantially but ensures that overwriting of the core does not result in the program going "wild". The Univac 1108 ALGOL compiler offers three options, checking by subroutine, access by subroutines, or access by open code. This allows the user to trade checking against space or time. In practice, it may be best for the compiler writer to make most of the decisions leaving relatively few options open to the user. The programmer with a large program may, however, welcome an option giving compact code for a speed penalty.

In very few cases do compilers deal with subscripting in a perfectly rigorous manner. For instance, on KDF9 an integer is 48 bits whereas the address modification registers are 16 bits. Needless to say, a subscript is placed in the 16 bit register for array accessing without checking for 16 bit capacity. The same happens on Atlas with 24 bit address registers and 48 bit integers. The author has never heard of a genuine error being missed because of this although some programmers have exploited this deficiency (using the top bits for markers).

Checking the subscript of a switch is slightly different. The 'lower bound' is always 1 and only one subscript is involved. Also, a minority of systems implement true ALGOL 60 in that a designational expression to a switch out of range is regarded as a dummy. This is usually handled by making the evaluation of the label into a subroutine. If the **goto** is successful, no return is made, otherwise the subroutine call acts as a dummy (apart from side-effects).

5.3.2 OVERFLOW AND UNDERFLOW

The methods adopted to handle these conditions are likely to be very machine dependent. They may be both an interrupt condition (B5500), masked interrupts (360) or detection only of overflow (KDF9). Apart from the basic hardware system, the operating system or conventions of the machine may determine the way these conditions are handled. The general problem was discussed in 1.1. Providing the hardware permits, it is very convenient if the programmer can monitor overflow or underflow by a procedure. This allows, for instance, repeated rescaling of a calculation. To achieve this, one does not want the overflow condition to be signalled as an error before the procedure is called. More rigorous testing can be made by a system if no assignment is allowed until overflow/underflow has been checked.

Overflow is checked by explicit code with the Kidsgrove compiler. Apart from not allowing rescaling conveniently, this has the disadvantage that a substantial amount of code is generated to check overflow, even though not all assignments are checked (value and for loop control variable assignments are omitted). To overcome some of these difficulties, the overflow checking code was made optional, but an additional overflow check was inserted in the procedure exit subroutine. Since the main program is regarded as a procedure, a final check is assured. Also, the overflow register is set by an **or** operation, so there is no danger of losing an overflow setting.

Even though a satisfactory method of dealing with overflow and underflow may be used, it must be consistently and thoroughly applied. Control routines, the standard functions and other external procedures should all use a uniform method. Particularly vulnerable is the exponentiation routine, and the standard functions *ln*, *exp* and *arctan*.

5.3.3 STACK OVERFLOW

The checking of adequate stack space is usually performed by the control routines used for procedure or block entry and array declaration. If procedure entry is accomplished by open code, then stack checking may be rather more difficult. In some operating systems it is not necessary to check, since upon violation sufficient information is available to take whatever action is required. On many computer systems one is required to give the store requirements of a program before loading and this cannot be changed during execution. Obviously under such circumstances, stack overflow is an error. In other cases, requests for more core-store can be made by a running program. On most systems in which this is allowed, it is an extremely expensive facility which must be used with caution by the control routines. Hence procedure or block entry is far too frequent to allow its use in this, the most natural, way. In any case, the dynamic behaviour of most ALGOL programs does not warrant such a close control of storage allocation. Programs invariably claim a substantial amount of array storage initially and the only dynamic element is a very much smaller amount required for working storage. Since the depth of procedure calls is rarely more than three or four, it is array and program storage which make the major demands on space. Array space is claimed very much less frequently than procedure working space (see PE and MSF in Appendix 6). Hence it is only reasonable to claim and release stack space at rates and amounts comparable with array declarations. This is exactly the system adopted with the B5500 computer where array declarations are a system call, but procedure calls and claiming of working space on the stack is a hardware function.

The problem on a conventional machine is to devise a control mechanism for storage allocation with adequate performance. On the 1900 series, the ALGOL system claims but never returns storage. This means one claims space very often when the stack is expanding, but has the obvious disadvantage of letting programs acquire store which may not be fully utilised. A further disadvantage, shared with all systems allowing expansion, is that a “wild” program may cause substantial difficulties. A reasonable compromise is to claim space in some fixed unit, say 100 words, and then to return 100 words if 200 words or more are free. This would avoid claiming and restoring space at too rapid a rate.

A more satisfactory system is to allocate and release in units over which the user has more direct control, which really requires a separate mechanism for dealing with arrays and the stack. This can be done on a conventional machine, but not without a penalty in delaying some of the storage mapping until execution time. When arrays are stored separately, then the return of array storage may have to be explicit, i.e. it is not adequate merely to reset stack pointers. On the B5500, on a jump out of a block (containing arrays), the control program searches for array descriptors in the stack, so that their space can be returned to free storage. The Eindhoven X8 compiler works in a similar manner which is discussed by Bron (1971).

5.3.4 USE BEFORE ASSIGNMENT

This is a common form of programming error which is very rarely checked. The value of a variable before assignment is not defined in the Report, so compiler writers are presumably free to give it any value. In view of this, a programmer certainly cannot expect to get consistent answers if unassigned values are used. Therefore it is highly desirable or systems to make a check on unassigned values if any reasonable means is available. The Univac 1100 series compiler initialises all simple variables and array variables to zero (and **false**). In the author's view this is a very dubious practice, since it makes Univac programs less portable, and also precludes any logical checks on use before assignment and, of course, there is a penalty in the initialisation if zero is not the required starting value.

The ease and the extent to which checks are possible depends upon the hardware. Ideally one would like an interrupt when a particular store location is accessed. This is not possible on the majority of machines although a close approximation can sometimes be achieved. On the 7090, a parity feature was used for exactly this purpose, although it somewhat defeated the object of the parity check. On some computers with a separate floating point accumulator a load instruction can cause standardisation shifts. These in turn can cause overflow. So an overflow interrupt can mean that an unassigned value has been fetched. This method is used on the 1900 series compiler. Unfortunately, in almost all cases it is not possible to do the same with integer values.

On KDF9, several binary patterns are not valid integers or floating point numbers, because the ALGOL system restricts integers to using 39 out of the 48 bits (although this is checked very rarely). The purpose of this is to allow floating point calculations (with a 39-bit mantissa) to be exact with integers, in order that real arithmetic could be used even when the true type of an expression was unknown at compile time (see 1.1.4.2). The author made a modification to the Whetstone system which initialised the stack before program execution with an undefined value. The interpretive instructions TIR, TRR, TBR and INDR were modified to fail if this bit pattern was found. An additional 10% of programs failed, including many that were apparently working (even for years!). This method does not necessarily check every use before assignment, since the stack is only set once, not on every block entry.

A similar modification was made to the Kidsgrove ALGOL compiler with less satisfactory results. It is not possible to check on every load instruction since no standardisation shifts are involved. However the bit pattern does overflow very easily—for instance if it is multiplied by 1.0. Hence in most cases use of an unassigned value does lead to a failure even if this is not immediate.

Basically similar methods to these are used on a number of systems for instance, X8 ALGOL at Eindhoven and ALGOL 68-R at Malvern.

An interesting description of trapping the use of “undefined” variables in FORTRAN is given by Moulton (1967).

5.3.5 ADDITIONAL CHECKS

Almost all subroutines used to perform the more complex functions in ALGOL have some error conditions. For instance, on array declaration, the lower bounds must be less than or equal to the corresponding upper bound. Invalid assignment to a name parameter usually has to be checked dynamically. The subroutines used for exponentiation in 1.1.4.2 have two errors.

Unfortunately many compilers leave virtually all parameter checking until execution time although this is only necessary in a minority of cases. Apart from the obvious execution overhead, program checking is more difficult and the run-time system is longer and more complex.

In practice, the largest part of most run-time systems is the input-output subroutines, which are likely to contain the majority of error conditions. Such routines must be very robust, since they will be required to report errors, and hence usually have numerous checks.

5.3.6 TRACING

This is a very popular method of providing diagnostic features in a system. It is easy to implement, but the information given is not always what the user requires. Basically additional code is compiled at various parts of the program so that when these parts are executed, extra diagnostic information is printed.

Three program constructions are obvious candidates for tracing. Calling, a procedure, passing a label and performing an assignment. In the first two cases the corresponding identifier can be printed, so that the user is given the information in source text terms. An assignment is more difficult in that there is no obvious identification to give. Usually a method is adopted similar to that of the compiler, but unfortunately rather more is required in order to give the flow of control in the program. Procedure exit must be included with procedure entry to give the correct recursion level. Also, a call by name evaluation can cause control to pass to a completely different part of the program. In most programs, the most significant omission is that of conditional statements and for loops.

The more information that is gathered, the more difficult it becomes to print it in a succinct manner. All too often the programmer becomes inundated with the tracing output so that the important information is hard to find. Special procedures may be provided to control the output dynamically, but this tacitly assumes that the programmer can anticipate his failure correctly. Multi-access systems with powerful interactive facilities can obviously enable the user to exploit some debugging aids. In some systems, tracing information is written to a file, which can be displayed in a number of ways. One system actually allows one to run the program backwards from the point of failure!

Very elaborate tracing facilities have been added by Satterthwaite to the ALGOL W system (Satterthwaite, 1971). At its most detailed level, each source text line is printed upon execution, together with a summary of the value of assignments, etc. Each procedure call execution listing shows the depth of recursion, which is reprinted on return from another call. An interesting method is used to control the volume of tracing output. Each source text statement is traced a specified number of times—the default is twice. After this, only the number of executions is accounted, and the processing rate is virtually full speed. The final output consists of the source text and flow summary (see 6.3)

Very detailed levels of tracing tend to be used when all other methods have failed. Such long-stop methods are important because the user need never despair in the face of a particularly awkward bug. They can also be used to demonstrate compiler faults—although in some cases, errors disappear when tracing is done because optimisation is suppressed. Apart from the lack of selectivity in the output, the other disadvantage of tracing is the processing overhead. A not infrequent problem is a program which fails in an inexplicable manner after half an hour of calculation. Tracing is out of the question in such cases since the overhead is too great.

One method of selecting diagnostic output is to wait until program failure, which is the subject of the next section.

5.4 Failure Time Diagnostics

Having detected a failure at run-time, it is necessary to inform the programmer of all the relevant details. Existing ALGOL systems vary to an enormous extent in the assistance that they give programmers in this crucial area. At one extreme, two characters are printed indicating the nature of the failure with no other details, whereas the ALCOR 7090 compiler indicates the nature and position of failure together with the value of variables. There is little excuse for not giving good information on program failure because the overhead is very small. A program overlay can be used, so no additional core-store is necessary, and the extra peripheral transfers that may be required are insignificant compared with the benefits to the user.

Information produced on failure can conveniently be divided into three parts; firstly that relating directly to the failure, secondly that relating to any failure and lastly other information which may be of value to the user.

5.4.1 FAILURE INFORMATION

The most important information concerning the failure is usually its exact nature and the position in source text terms. Additional information should be available which if printed appropriately would be of great assistance but its nature varies substantially with the failure, and is not easily parameterised. A subscript bound violation is a common error, but no compiler known to the author actually prints the array concerned and the values of the subscripts.

In many cases the position of the error in the source text is not easy to give. The run-time system usually has available only a machine-code address and even this may be difficult to find, as the failure could be detected inside an internal run-time subroutine. Obviously these routines must be carefully designed so that all the relevant information is conveniently available. Relating the machine-code address to the source text requires a table for a conversion. In many systems, provision is not made for procedural access to these tables. The relevant information is merely printed by the compiler, relocating loader, and failure routine. This is adequate, but hardly a convenient method as far as the user is concerned. Ideally, the high level language programmer should not be aware of machine code addresses, which are always converted to source text terms automatically by the computer system. An intermediatory process of a relocating loader can make the problem more difficult than one simple table look-up. One system for the KDF9, while always giving the necessary machine addresses, required one to add and subtract in octal to find the corresponding source text position.

Faced with the difficulty of not being able to relate a machine address to the source text, some compiler-writers have adopted other techniques. At one extreme, the current line number of the source is held in a register. This is a substantial overhead, both in terms of the number of instructions executed and the size of the compiled program. A less accurate indication of position is therefore more usual. The Kidsgrove compiler for KDF9 gives the last ALGOL label passed and last procedure called. Unfortunately both of these are given in terms of numbers generated by the compiler, and so must be related to the source by use of the tables produced by the compiler. The CDC 6600 compiler generates code to preserve the source text line every ten lines, and on procedure calls.

The additional information concerned with the failure is equally hard to handle as even the number of parameters can vary. Ideally, such parameters should be printed as real or integer as appropriate, with text explaining their significance in this particular context. This should be done for the most common errors, but some simpler process is necessary for straightforward cases, many of which may arise from user-produced code. An ALGOL procedure which calls the error handling routine should be available in every system. The Kidsgrove and Whetstone compilers have a compatible error system which has three parameters, a descriptive string, an error number, and a value, which seems adequate.

5.4.2 INFORMATION RELATED TO THE FAILURE

In the previous section, one instance was given of information which is maintained dynamically purely for diagnostic purposes. This was the name of the last procedure called and label passed. A much more powerful and useful technique is the retroactive trace. The idea behind this is simple. One difficulty with ordinary tracing methods is that the volume of output can easily be excessive. The retro-active trace technique involves preserving this information in a cyclic buffer rather than printing it immediately. If the program fails, then the information is printed, the amount depending upon the size of the buffer.

The Whetstone ALGOL system maintains one cyclic buffer of 16 elements for procedures and labels. One enhancement has been added to avoid simple loops for overfilling the buffer with multiple copies of one identifier. Each identifier has a count, so if one is repeated immediately, the count is incremented. This is only partly successful since only very simple loops have one procedure call or label. The table in Appendix 6 shows that procedure calls account for 90% of the entries in the dynamic trace. Experience with the use of a retro-active trace shows that the information given is about as useful as an accurate indication of the position of failure. The strong looping characteristics of most programs means that the values of control loop variables are more important than a dynamic trace of procedure calls. During the first execution of a program, it is not unusual for failure to occur before the buffer is full. In this case, the whole program has been traced, but on other occasions the buffer could with advantage be slightly longer than 16 elements.

It is possible to combine tracing and retro-active tracing. For loops or blocks or procedures one can arrange to trace the first so many uses and thereafter to place the information in a buffer. This corresponds very closely to the way programmers "dry-run" programs. When the pattern of behaviour of one program unit becomes clear, the dry running of that part is omitted (it being assumed that it performs satisfactorily in all cases). The ideal solution would be to combine compiling, tracing and retro-active tracing into an interactive system.

5.4.3 PROGRAM STATUS INFORMATION

With few exceptions, a point of failure is not very different from any other point in the program. The status of the program at the failure can be made available to the programmer so that he can determine the error more readily. Techniques currently used vary greatly, both in the quantity of information produced and the ease with which it can be interpreted. The majority of systems produce no information at all—this is a real deficiency since it is fairly easy to do.

One of the first compilers to give good failure status information was the ALCOR Illinois compiler (Bayer, 1967). Provided a compile and run had been used, the status information is printed in source text terms. The values of all simple and array variables are printed for all procedures or blocks which are currently active. The printing of array variables is probably not worthwhile, since the amount of printing required is substantial. The most crucial variables tend to be for loop control variables and a small number of working variables. The difficulty with the ALCOR method is that no choice is exercised over the variables listed.

The Atlas ALGOL compiler has a similar system which has some advantages and disadvantages over the ALCOR method. On entry to every block, code is executed to place in the link data for the block the addresses of simple variables to be printed. This does entail an execution overhead, but a slight choice can be exercised over the variables listed. The chosen variables are those appearing in the first type list of reals, integers and Booleans in the block. So by reordering the declaration of simple variables appropriately, a selection can be made. Each block or procedure is identified by a number which can be related to the source text from the (short) compiler tables. No identifiers are printed against the variables either, but these can easily be found from the relevant block (with the source listing).

Implicit in both of these systems is a listing of the dynamic chain of activation of procedures. This is the only status information printed by the Eindhoven X8 compiler. Although useful—it at least gives the point of failure to within a procedure, it may miss some of the more significant information. Apart from the procedure chain, the evaluation of call-by-name parameters could have an effect on the listing. It is not easy to arrange a satisfactory result, because the thunk mechanism implies a return to the old environment. Hence if no special care is taken, information about the procedure with the name parameter will be lost. One method is to have a forward as well as a backward procedure chain. This can be set up easily on procedure entry and cleared on exit, so that the current procedure activation can be determined and checked from any environment.

Experience with a post-mortem facility shows that it is rare for more than 3 or 4 procedures to be active at a point of failure. Hence the printing of all simple variables is quite reasonable, at least on a line-printer. It is the author's view that every ALGOL compiler should list such variables on failure, even if it is not possible to print their identifiers.

5.4.4 THE KIDSGROVE ALGOL POST-MORTEM FACILITY

A program status print-out has been added by the author to the Kidsgrove ALGOL run-time system. Only one very minor modification was required to the compiler, and, since the techniques could probably be applied to the majority of systems, a description is given here.

The system consists of a binary program which is called as an overlay if an ALGOL program fails. The overlay area is part of the run time routines which are of no significance once the program has failed. The post-mortem assumes that two index registers, representing the current environment pointer and stack pointer are still set correctly. In order to check this, certain tests are performed upon the registers, and the dynamic chain of procedure calls is inspected, validating the link data at each level. If these tests fail, then only the first 200 variables of the stack are printed.

Provided the link data at each procedure activation is correct, the package prints the following for each level. Firstly, the number of the procedure. This can be determined from the link data by inspecting the machine-code of the call (which can be found from the return address stored in the link data). The address of the call is also printed, although this information is rather difficult for the user to relate to the source text. Next, the first 200 variables are printed of that procedure. If no arrays have been declared in that procedure, then almost certainly the space in that environment will be smaller than 100, and so few variables will be printed. Fortunately, it is possible at a binary level to determine the type of a variable. This can either be (0, 0.0, **false**) or (-1, **true**) or **real** or **integer** for user declared variables. Certain other information is stored on the stack which may have a bit pattern not corresponding to a real, integer or Boolean value. One of these bit patterns is used to initialise the stack, so if this is found, the message ‘not assigned’ is printed. The remaining invalid bit patterns are printed in octal, although it is unlikely that the user will be able to make any sense of them.

The basic information given is therefore the procedures currently active, and the values of variables in these procedures. Variable 6 of procedure 8 might be listed as 3.14. By consulting the compiler tables, the user can find the identifier of procedure 8, and that of variable 6 within the procedure. One complication arises because first order working store is assigned at procedure level (see 1.8.2). This means that variable 6 (say) could correspond to several different entities in separate parallel blocks of the procedure. Fortunately the block structure of procedures is rarely complex enough to cause confusion.

Experience with the system has shown a number of advantages and deficiencies. Firstly, the correct printing of the values according to type, and whether or not they have been assigned is a substantial assistance. Secondly, in many cases, the value of a variable is crucial to a failure, and so this is of great benefit. For instance, when a number has been mis-read, the value of the control variable in the reading loop is the only significant piece of information required, which is not given by other methods. Thirdly, it is very rare for the procedure depth to be greater than 4, which means that only about 100 lines are printed. It is not necessary, therefore, to make the post-mortem an option. One deficiency is that the control routines were not, of course, written without the post-mortem in mind. Hence quite a few failures detected in the run-time package leave the crucial registers in an incorrect state. Also, the call-by-name method used does not allow one to detect that a thunk is currently being processed and this means that the listing of some environments can be lost.

A major disadvantage of the system is that no identifiers are produced with the post-mortem. In fact, a small modification was made to the compiler so that the variable

Percentage	Reason
50.70	time limit expired
7.50	subscript overflow
6.10	call read at end of data
5.40	variable used before assignment
5.40	real overflow on /
4.50	array variable used before assignment
4.20	various input-output failures
2.70	error in machine-code
2.50	actual-formal incompatibility
2.10	*sqrt(x)*, $x < 0$
2.10	program needs too much space
1.50	real overflow not / or $\uparrow$
1.20	error in $\uparrow$
1.10	dynamic type check
0.81	integer overflow not $\uparrow$
0.79	lower bound $>$ upper bound on array declaration
0.74	*ln(x)*, $x \leq 0$
0.27	*exp(x)*, x too large
0.10	out of range switch index

Table 5.1: ALGOL errors

memory map was listed with the other tables. Ideally this information should be written to a file which the post-mortem routine could use for a listing but unfortunately, the operating system would not support such a facility. Although interpretation of the post-mortem printing with the aid of the compiler tables is straightforward, it does mean that users now have to be familiar with the significance of the compiler tables which was not previously necessary.

Although being able to determine the type of a location from the bit pattern is useful, it is not essential. Clearly all interpretations of the bit patterns could be listed, but such excess of information is likely to confuse the programmer. In practice, integers are almost always limited to small values—much less than the address length of the machine. This can be used to give an interpretation which is very likely to be correct (certainly better than printing in octal or hexadecimal).

An interesting defect was found some time after the first release of the post-mortem. In calculating Ackermann's function (Sundblad, 1971), the program failed time limit with over 300 nested activations in the stack. The post-mortem was designed only to print 3 or 4 pages, but in this instance produced 100 pages before failing owing to excessive output. This was rectified by printing only the first 20 active procedures.

5.4.5 FAILURE STATISTICS

In Scowen (1972), details are given of a modification made at NPL to the run-time routines of Whetstone ALGOL to accumulate statistics on program failure. A total of 8902 'failures' were detected as listed in Table 5.1.

Note that some of these are not strictly errors, but are treated as such by the system. Only 3.7% extra would have failed at compile time by placing type checking there.

The errors in machine-code include any errors detected external to the ALGOL system itself.

5.5 Operating System Features

In many cases, the diagnostic features that are offered in an ALGOL system depend upon the operating system of the machine. In order to provide good testing facilities, certain operating system functions are necessary. If these functions are absent, a special subsystem may have to be written.

It is certainly of great assistance to be able to recover from a program failure in such a manner that a post-mortem can be produced. Very exceptional circumstances may preclude this, but it ought to be possible in the vast majority of cases. As mentioned above, it is convenient if stack checking is not necessary, which can be arranged on most machines by using the limit register which is set for core-protection. When a violation occurs, the recovery coding can either request more store (if this is allowed) or terminate the program with the appropriate error message. Similarly invalid addresses can be passed for name parameters to which assignment is not allowed. An attempt to make an assignment will then be trapped as a hardware error, obviating the need for a special software check.

Much depends upon the system for linking independently compiled routines. In practice, most programmers compile their program as one module, although not with large programs. Hence the most critical area is the way in which the run-time routines link to the ALGOL program. Link-editing, although it has a number of disadvantages, does allow some decisions to be delayed until program execution. About half of most ALGOL run-time system relate to functions which are very unlikely to occur in most programs. Link-editing will ordinarily avoid these being loaded. Many systems use the link-editing stage as a method of assembling routines written in different languages, and may even allow the ALGOL program to call upon a large FORTRAN library. There are substantial difficulties in arranging this, since the calling sequences for independently compiled modules must be compatible. The problem can be simplified by demanding that the language of the called procedure is specified.

If link-editing is performed then the implementation of some facilities becomes very easy. One can achieve a lot merely by loading a different version of the run-time routines. Two examples of this are given in section 6.3. For instance, a retroactive trace can be kept for any language feature which uses the run-time routines. The difficulty in this approach is that it may not be possible to relate the subroutine calls to the source text. This is a fundamental problem with most link-editing systems. They are designed merely as a method of collecting together independently compiled pieces of code, so that a complete binary program can be constructed. Usually this consists of inserting addresses in instruction and data fields corresponding to each external identifier and almost always exclude type-checking, or any other validation. Hence the onus is on the user to check that the specification of independently compiled modules is correct—or even that a procedure is not used as a piece of data. Hence, if any validation is to be done, it must be performed dynamically with a space and time overhead.

Ideally, a link-editing system should allow the storage of the specification of procedures so that this can be checked at load time. Also, additional information concerned with the storage of internal variables should be preserved so that a post-mortem routine can produce listings in source text terms. This is not easy, since the method of allocating storage varies from language to language. A system for FORTRAN would not be adequate for the stack allocation of ALGOL or the controlled storage facilities of PL/I For this reason it may be necessary to store information for the post-mortem system on a separate file to that of the binary modules. A module in some systems can consist merely of the specification of a data area and values for its variables. This should be extended to ALGOL by allowing independent compilation of array declarations, preferably with a method of initialising values (which is not available in ALGOL).

With a link-editing system, the loading of a program becomes a process in its own right, so that one might reasonably ask what actions it can take to assist program development. Clearly many debugging options should be capable of specification at load time. This is particularly important if compilation speeds are slow, and in any case is more natural from the programmer's view-point (the probability of program error is data dependent). Programs frequently produce erroneous results after having "worked" for some time.

Load time diagnostics can often be used in such cases more conveniently than any other method. Load time options can only be really effective if different code can be loaded depending upon the options. Merely substituting one module for another is too crude a method. For instance, if it is required to make array subscript checking a load option, then short sequences of code for array accessing must be changed from open code to longer ones involving subroutines calls. Therefore, the loadable code should allow conditional generation of code. A system like this has been devised for the ALGOL 68-R compiler (Currie, 1970), and was the main reason why the manufacturer's standard form of loadable code was not used. If global optimisation is performed, similar to the ALCOR compilers then it may be impossible to produce diagnostic code which is only locally different from the optimised form. In such cases, a compiling option must be used, which is in any case, the most usual method.

The most elaborate debugging systems at present seem to be those for machine-code programs. However, many of the techniques used are applicable at a higher level. Such things as test points, interrogation of variable values, resetting of values and controlled execution are not automatically ruled out because the source text is ALGOL 60. Undoubtedly, dynamic storage allocation means that any system must be constructed with more care. Also, one cannot 'anti-compile' ALGOL 60 in the same way as assembly code, so methods must be devised of linking to the source text. Again, the loadable code format can help here by allowing storage of line numbers or similar information with the code.

Most systems seem to assume that any external name, i.e. one not declared locally, is global to everything that the program has access. Logically one would like as many global levels as necessary. An overlay could be a level of nomenclature, which automatically ensures that overlays always communicate via the fixed code—a very convenient restriction from the operating system viewpoint. The absence of these facilities, and the structure of most link-editing schemes suggests a strong reliance on FORTRAN and assembly code.

Chapter 6

Program Documentation Aids

It has been stated many times that a piece of software should, with its accompanying comments and documentation, be as lucid as a text book. Unfortunately it is all too easy to produce a routine which apparently works, but to create from this a well-documented piece of software worthy of publication or release to the computing community is very much harder. If automatic or semi-automatic methods can be provided to enhance the standards of existing software, then these deserve substantial attention.

The eventual aim in this area is to provide, as part of the compilation process, a formal proof that the algorithm meets its specification. Unfortunately most practical software is beyond the capacity of current techniques (see London, 1970). Even a formal proof of correctness may not help a person who wishes to modify an algorithm for another purpose. Indeed, one of the disturbing facts about modern software is the lack of ease with which it may be altered because the consequences of any change cannot be safely determined.

6.1 Source Text Methods

The most obvious source of documentation is the program listing itself. This can either be re-listed in a more intelligible form or be subject to a detailed analysis which is printed as a documentation aid. For assembly code and FORTRAN, a favoured technique for program analysis is the flow diagram. This is not particularly appropriate for ALGOL for a number of reasons. Firstly, the presence of good implicit control statements (**if for else**) means that there is less need for a flow diagram. Secondly, the procedure call mechanism with parameter evaluation cannot be easily represented in a flow diagram. Lastly, the main premise of a flow diagram is that the blocks of code connected by lines in the two dimensional diagram can be laid out in the one dimensional program text in an arbitrary order. Although this may be logically so, the resulting code would contain so many explicit jumps as to render the code unintelligible. Existing flowcharters produce a very reasonable output from languages containing a large number of explicit jumps, but require a time usually much in excess of a compilation. Also, the complete flow chart of a large program requires so many inter-page references as to hinder the task of following the code.

The implicit flow structure of ALGOL 60 is adequate in itself to display the important features of the flow of control, provided the source text is properly laid out. Since manual maintenance of a good layout to an ALGOL program is virtually impossible, an automatic layout program is required. Several of these have been written and at least two published (Mills, 1970; Scowen, 1969a). The basic method behind these editors is to perform a partial syntax check on the program so that a layout can be produced to indicate its structure. The form produced is very similar to that of published algorithms, although many journals have a rather severe restriction on column width and length of the text. The editor called SOAP (Simplify Obscure Algol ProgramS) has been in use for over two years at NPL; experience from which has high-lighted the following points.

1. It is far more useful than one might think at first. The author would never consider programming in a language without one. This is especially true if inspection of other people's programs is required.

2. It must be convenient to use. The version at NPL works like an amendment and requires very little effort on the users' part. It is efficient, working at about twice the speed of the Whetstone compiler, and so it can be used frequently.

3. SOAP produces a very spacious layout-one assignment statement per line, for instance. Provided this does not use excessive disc storage, it should be maintained, since program readability is increased.

4. Parameters to SOAP allow one to convert to narrow pages or make effective use of long lines (on line-printer).

5. A good editor is difficult to write, so use of SOAP which is available in ALGOL 60, is strongly recommended.

Apart from providing a well laid-out listing, a source text analysis can be made. With large programs a concordance of all the identifiers with the corresponding line numbers is invaluable. A version of SOAP, designed to give information on the line-printer rather than for re-put, has been written with a concordance. The program is listed with line numbers, split into pages at appropriate points (with context information heading each page), followed by a concordance. Even this version of SOAP falls far short of a compiler, as information on the scope of identifiers is not maintained. So against each identifier is listed all the declarations and then all the uses, positional information being given with line numbers. With such a listing, making changes to a large program becomes very much easier and safer since it is possible to check that every use of a variable is consistent with the change.

An index of the use of constants can also be produced by SOAP. Inaccurate values (or even worse, inconsistent ones) given to constants can easily be found and corrected. A detailed analysis of the use of constants can be most informative. If 96 say is used, why not 95 or 97, or should the constant be replaced by a variable which is initialised at the start of the program or procedure?

A concordance program can be used in isolation for other purposes. A general concordance program is not suitable for ALGOL 60 as it stands since compound ALGOL basic symbols, strings and comments must be ignored, which as pointed out in 4.1.1 is not straightforward.

Production of more detailed analysis of the source text than that of SOAP with the concordance is most effectively achieved by the compiler itself, which is considered in the next section.

A different technique with the same general aim is described by Mills (1970). Here the source text is scanned by a program running interactively. At critical points, additional information is requested from the user which is inserted in the text, mainly as comments. The advantage of this technique is that it could be used to ensure that a certain standard of source text is maintained without placing a severe burden on the often unwilling programmer.

6.2 Compiler-oriented Documentation Aids

In the course of the compilation process, a compiler necessarily deduces much about the program which could be of value to the program writer. Unfortunately, many systems do not allow one to obtain this information in any reasonable form. In desperation, one may print a machine-code listing in order to discover whether a particular program loop has been optimised, even though the details of the code as such was of no interest. A further difficulty is that such options that are available to list compiler information may not be ordinarily known to the user, or the output may be unintelligible to anybody but a compiler-writer (or worse, not even to the compiler-writer!).

Information that a compiler-writer can make available, perhaps by a special version of the compiler, is not just of academic interest. The use of non-locals within a procedure is paramount for its inclusion in a library system, and should be checked automatically. Of equal importance are details of any use of language features commonly not available, since unnecessary use of such features can severely affect an algorithm's portability. Lastly, the use of certain constructs is almost certainly bad programming practice and so should be brought to the programmer's attention.

A list of the constructions which a compiler ought to note in such a documentation mode is:

- real to integer type conversion
- array variable as **for** loop control variable
- equality of real expressions
- no assignment made to a name parameter
- use of constant sub-expressions
- use of arrays by value
- function designator called by a procedure statement
- an array whose elements are only accessed with constant subscripts.

Almost all of these constructs are likely to be bad programming practice, particularly the last one which can arise with programs converted from lower level languages.

It would be desirable to develop a compiler which was capable of making a very detailed analysis of a program. Unfortunately this cannot be justified in economic terms unless a machine-independent version could be developed. Such a program could make a global analysis of a program in order to exhibit its overall structure. The use of Jensen's device or recursion could be detected so that the validity of a straightforward conversion to FORTRAN could be checked. Some compilers make an analysis of this form, for instance, the Kidsgrove ALGOL compiler produces a calling matrix of procedures to detect recursion and side-effects. An analysis of the use of subscripts is very important if recoding in machine-code is to be attempted, since some of the largest gains in speed are to be found in this area. In many cases, particular control variables are used for certain subscript positions. Consistency of use in this can be checked, the anomalous ones pinpointed which may allow the programmer to recode particular program parts to achieve uniformity and greater intelligibility.

Several bizarre features of ALGOL 60 can be involved in the execution of what appears to be a harmless program. This is really a design defect in ALGOL 60, but one which has appeared in many languages since then. Unfortunately almost all systems either reject the program (which at least informs the use of his unusual requirement) or accepts it without comment even though extra run-time routines are required and in consequence the program may run at a fraction of the speed necessary. This is a most unfortunate state of affairs which a compiler could do a lot to help. Educating the programmer by traditional means is not adequate because the traps are too easy to fall into. The compiler invariably has the information available and so should inform the user if he so desires.

Some of the language features in this area are as follows:

- call-by-name involving dynamic type check
- formal procedure call involving dynamic parameter checking
- use of $\uparrow$ forcing dynamic type check (or ad hoc rule, see 1.1.4.2)
- formal array dimension cannot be deduced at compile-time
- an invalid assignment to a call-by-name parameter is logically possible
- real **for** loop variable with **step until** element.

A further degree of elaboration would be worthwhile if large volumes of code were written in ALGOL 60, as is the case with operating systems (in assembly language?). In order to control the production of such code, the managers of large software projects could use a version of a compiler giving a substantial amount of statistical information. Such things as the average length of identifiers, the volume of comments, average size of a procedure and so on can be easily obtained. One cannot pretend that even a sophisticated compiler can assess the coding style of a programmer, but such aids could be a useful adjunct to more conventional methods of quality control.

6.3 Execution Aids

In this section it is assumed that it is easy to obtain different versions of a program either by recompilation or reloading. This was a requirement of some diagnostic aids and was covered in Chapter 5.

The most important information that can be obtained at execution time is an accurate account of the dynamic characteristics of the program. Very many programs spend 90% of the time execution 10% of the code. This is important in two ways. The 10% of the code can possibly be recoded to run quicker, while the 90% can be recoded to be more compact. The output from a dynamic flow analysis package very often surprises the author of the program, because one tends to concentrate on the complex parts of the program, forgetting that a simple loop is nested one deeper than the rest, or not allowing for the time taken for the input and output. An automatic method is therefore essential for the proper understanding of a program.

Several methods are available for obtaining a count of the usage of each part of a program. Knuth (197la) describes a method whereby a FORTRAN program is analysed by means of a pre-processor. To each executable statement is added a further one which increments the elements of an array. The main program is enlarged by a subroutine call which zeroises the array initially, and prints out the array with the source text of the program at the end. This is not quite so easy with ALGOL 60, since a partial syntax check of some complexity needs to be performed to add statements.

The author has added a system to the Kidsgove ALGOL run-time controller to determine the main characteristics of a program without recompilation. When the program is loaded, one can detect a procedure entry, passing a label, and storing a value in an assignment statement. Subroutine call, not ordinarily generated, can be placed in the binary program at these positions. The numerical value associated with the procedure and label can be used to increment a count inside these subroutines. In the case of an assignment, no positional information is available except the machine-code address, so this is used. As the key used for each count is now not dense, a table search must be employed. As speed is of some importance, a hash table with a linked list is used. At program termination, the key and the counts are listed, in both address and count order. The only major difficulty with this scheme is that the user must spend a few minutes linking the listing produced to the source text.

Obviously, when both the compiler and run-time system are suitably modified, a very good dynamic flow analysis is possible. This has been done by Satterthwaite (1970) and Scowen (1972). In this case, every point altering the flow of control is modified so that counts are accumulated. With a compiler modification it is possible to ensure that these counts are not more frequent than necessary. Hence the program size is only likely to increase by about 15% and the speed reduced by a similar amount. This is important since it means that large and long-running programs can be analysed—just the ones for which the information is most valuable. The extra frill of producing a side-by-side listing of the program text with the dynamic counts makes it very acceptable to the programmer.

Merely knowing the most frequently executed parts of a program is not all that is required. A time estimate is also necessary so that the break-down of the most time consuming parts of the program can be given. The ideal solution here would be to combine the machine independent estimates of time (in chapter 3) with the dynamic counts mentioned above. This would be extremely useful in the assessment of algorithms. At the moment, there is a tendency for algorithms to be coded to run optimally on a particular compiler without regard for other compilers which may perform somewhat different (or no) optimisation. It is not always easy to time small program parts on computers (see 2.2.1) but it has been possible to give a reasonable division of execution time on KDF9. A procedure was taken as the program unit since a run-time routine is called for procedure entry and exit in most cases. A modification was made to these routines to account for the time, which involved using a small stack to keep account of the current procedure being executed and assumed procedures were called in a nested fashion without **goto**s. Again, the output from this program invariably comes as a surprise to the programmer. In one of my own programs, I was amazed to see such a high proportion of the time (half) was spent in the standard input routine.

Interpretive execution is useful for a detailed execution analysis simply because it is very much easier to monitor the program. With a well-designed interpretive system one can consider changing the form of the floating point facilities in order to explore the characteristics of an algorithm with respect to round-off. One of the first ALGOL compilers, Dijkstra's one for the X1, allowed interpretation with single or double length real numbers. Of course, with many modern computers hardware is available for two lengths of real numbers, but experimentation with other forms like interval arithmetic is desirable. The difficulty in making such changes is not the arithmetic routines themselves but the consequential changes to storage allocation, parameter handling, array address calculation and the like. With the KDF9 system, as it is possible to ensure that a subroutine is called on every assignment, one can arrange to truncate floating point numbers by a specified amount. This allows investigation of an algorithm's behaviour with a smaller word length.

Interpretation can also help in verifying that certain forms of error are not present, because a substantial volume of checking code can be added. Use before assignment is probably the most important error in this class (see 5.3.4).

6.4 Machine-code Analysis

Since many ALGOL compilers produce little information about a program except the compiled machine-code, a program to analyse the machine-code may be useful. It can also be helpful to a compiler writer since static counts of the use of various language features can be found (see 2.3.4).

One advantage of an analysis of machine-code is that its structure is much simpler than that of the ALGOL source text. On the other hand, it may be difficult to relate the code to the source unless various diagnostic options are used—in which case, some code optimisation may be inhibited. Assuming these difficulties can be overcome, such an analysis will contain the essential semantic information. One obvious feature to highlight is calls of the run-time routines, since these are necessarily expensive in processor time. Similarly, compiler macros which are particularly long, or are likely to be removed by suitable recoding, can be noted. Quite a few programs do contain constructs, while being very natural to the applications programmer, are either inherently inefficient or are so in a particular implementation. Evaluation of call-by-name parameters, integer-real type conversion, and exponentiation to a real power do often occur unnecessarily and can be noted in this way. This is a valuable adjunct to a source text analysis, since none of these are easily noted by a program working like SOAP. The use of non-locals within a procedure is also easily found, since a different display register is accessed or similar action taken (see 1.8).

Again, some statistical information is valuable. The size of the procedures, number of instructions involved in the inner program loops (noting subroutine calls), and the amount of first-order working space required for the procedures, can all be useful. In practice, an analysis program would print-out all the information readily available from the code which could be explained in source text terms. The programmer should not be expected to understand the machine-code generated by the compiler—use of such knowledge could be dangerous as it could lead to an inability to enhance the code produced.

All the techniques described in this chapter cannot replace clear and precise documentation provided by the author of the software. Conversely, no system can interpret comments and judge them to be adequate. The programmer who writes $i := i + 1$; **comment** increment i; to increase his quota of comments in his source text, is not documenting his program. Unless an overall description of a piece of software can be given in a few pages, it is unlikely that the implementation team had a sufficiently coherent view of the project to construct an effective system.

Chapter 7

Trouble-spots in the Compilation Process

In chapter 1 only the straightforward aspects of code production were considered, the more complex cases are analysed in detail in this chapter. Every ALGOL compiler-writer is forced to decide how these rather awkward issues can be managed. In every case, the emphasis is on a practical solution, practical both to the compiler and run-time system, which does not have any detrimental effect on less pathological programs.

It is important to note in this section that only compilers which demand complete specification of parameters are being considered. Most of the difficulties arise from procedure parameters, and this is made virtually impossible to control if specifications are not necessary. The majority of compilers do require specifications, and the few that do not (like the ALCOR compilers (Grau, 1967)) must inevitably attempt to deduce the specification or otherwise generate extremely inefficient code.

7.1 The Dimensions of Formal Arrays

An obvious defect in the specification of formal arrays is that no explicit indication is given of the array dimension. In practice this can often be deduced from the body of the procedure, but not if the procedure merely passes the array formally to another procedure. On the other hand, in such cases the array addressing information itself is not required—the array word can merely be passed on.

The solution is not difficult, although the majority of compilers simply ignore the issue completely, trusting that if a program passes the compiler checks, then the array dimension is always correct. A complete check in the compiler could involve an arbitrary long chain of array parameter passing, and in any case is not possible if independent compilation is allowed with only the usual specification of module parameters. Hence a run-time check must be made, for those parameters where a complete compiler check does not work.

The most reasonable solution is to check in the compiler when the procedure body gives the dimension, and every actual parameter is of known dimension. If the second of these conditions fails (which will always be the case with an independently compiled procedure) then additional checking code is added to the body of the procedure. This code need only be executed once for every procedure call and should be very short. The only significant general point that arises, is that the complete addressing information for any array must include its dimension.

The Kidsgrove and Atlas ALGOL compilers both ignore the problem, whereas Whetstone and the two X8 ALGOL compilers check on every array element access! The Buroughs 5500 compiler demands that the array dimension is specified—very reasonable but not ALGOL 60. This additional specification can also include the lower (fixed) bounds for each dimension so that open code can be produced for array element fetch without using the full addressing information. The array dimension is then checked on each call, but the lower bounds are assumed to be correct (this cannot lead to a violation of the core protection, merely wrong answers!).

The author has not known anybody violate the array dimension check, but one can imagine for instance, that it is rather easy to interchange two array parameters of a procedure. About a dozen of the errors listed in 5.4.5 arise from dynamic check in Whetstone ALGOL, but it is not known to what extent a more careful compiler check would have rejected the programs. The ALGOL 60 report gives no indication of the extent to which the array dimension can be varied. For instance is

if b **then** $a[1]$ **else** $a[1, 1]$

valid ALGOL 60? Such a program is rejected by the Whetstone translator although the interpreter would accept it provided the array dimension matched dynamically.

7.2 Formal Procedure Parameters

It has been shown in 1.9 that reasonable code can be produced for procedure and function calls provided parameter checking is performed at compile time. Assuming that parameter specification is necessary, the only substantial difficulty is that no information is available on the parameters to formal procedures. This is a major defect in the design of the language since its effect has been that many ALGOL systems check parameters at run-time incurring a substantial overhead (and delayed validation of the program). It is obviously unacceptable for ordinary procedure calls to pay a penalty for the formal procedure call mechanism which is less than a hundred times as common (see Call Function and Call Formal Function in Appendix 6). Therefore the compiler must attempt to solve this problem.

Several solutions have been adopted. In one system, a language change was made, so that the specification of parameters to formal procedures must be given with the specification of the formal procedure itself. This, of course, means that every actual procedure given as a parameter must have identical specifications—and the compiler can check this with only marginally more complication than ordinary parameter checking. This solution is very practical and is used in CORAL 66 (Woodward, 1970), but it is not ALGOL 60, and the purist could rightly object. A simple alteration can give all the advantages with no language change, by making the specification information into a mandatory comment. This is the approach used by the Egdon ALGOL compiler for KDF9.

One can tell in the main syntax pass of the compiler whether it is possible for a procedure to be called formally. This can only happen if the procedure identifier appears as a parameter in a procedure call (and, of course, there is at least one procedure parameter in the program). Since probably 90% of programs do not use formal procedures, and of those that do only 10% of the procedures could be called formally, this simplification already solves the main problem. All procedures that cannot be called formally are treated in the manner given in 1.9 which gives efficient code and compile-time checking of parameters.

The way in which formal procedure calls are handled is not now critical. The simplest solution is to produce checking code for *any* call of a procedure which can be called formally. One can improve upon this by ensuring that only formal calls are handled by the checking process which passes control to a special entry of the procedure. This entry checks the parameters, evaluates value parameters and places the parameters in an identical position to that they would occupy if a simple direct call had been made. At this stage, a jump to the ordinary entry of the procedure will produce the desired effect. This means that only the checking code interface need be uniform from procedure to procedure, since it is only this method which need be interchangeable. The ordinary calling method can adopt any optimisation that may seem reasonable, since it is only an agreement between the call and the procedure (such as leaving parameters in registers).

With an independent compilation system the same method can be used. A procedure called directly can have its parameters checked by the loader in the manner determined by the operating system. This is not possible with a formal call, so dynamic checking must be performed. However, for an independently compiled procedure one has no means of knowing whether it will be called formally. The solution is to generate for an independently compiled procedure two separate code blocks. One block contains the translated procedure body and the ordinary direct linking code. The second block is only loaded if a formal call has been requested, and contains the dynamic checking code, the entry point being the "formal address" for the procedure, and exit being to the first code block. With independent compilation it will ordinarily be impossible to avoid dynamic checking—although in some systems, the independent compilation parameter interface is much less general than the ALGOL 60 mechanism, implying a simple checking process.

A system based upon link-editing, has a number of advantages concerned with formal procedures. Assuming parameter checking is ordinarily done at compile-time, it is only in the case of formal procedures that the checking code will be required. This will inevitably involve quite a few subroutines, which need not be loaded unless required.

7.3 Arrays By Value

In 1.9.5 it was noted that the handling of an array parameter called by value does not cause any complication in the calling sequence. The "array word" can merely be passed into the called procedure. This procedure, must however, produce a local copy of the array before the body of the procedure can be executed. There are two major complications in this, firstly, the array dimension may be unknown, and secondly, an integer-real type change may be involved.

The difficulty with not knowing the array dimension at compile time is that the compiler cannot allow space in first order working store for the array addressing information. The KDF9 Whetstone and Kidsgrove compilers overcome this by using the addressing information of the original array. This is possible because only the array word contains machine addresses, whereas the other addressing information gives the equivalent of the array bounds, which can be shared for arrays declared in the same segment, or value arrays. The disadvantage of this approach is that it prohibits (on KDF9) a more optimal layout of the addressing information, and so it cannot be regarded as necessarily the best technique. In practice value arrays are very rare (see Appendix 6, about one twentieth of name arrays) and so restricting their dimension altogether to less than 5 (say) would be unlikely to be noticed, and would allow the addressing information to be placed in first order working store.

The problem of type conversion is not difficult, but it does mean that the array word or the other addressing information must indicate the type (integer or real) of the array. A run-time routine can then inspect the type to determine whether type conversion is required or if a simple copy is sufficient. This is an example of a relatively large run-time routine which should not be loaded into a program unless value arrays are specified.

It is interesting to note that although array assignment is not possible in ALGOL 60, such an assignment is implied by the value array mechanism. If the rule given in the Report (4.7.3.1) was applied then an array assignment would result, but nevertheless, it is clear that arrays by value were intended by the authors of the Report (see 4.7.5.3).

7.4 Subscripted Variable as a Control Variable

In 1.7, the case of the control variable as a subscripted variable was explicitly excluded. There were several good reasons for this. Firstly it is not a common construction, although one method for initialising an array uses it, viz:

```
$i := 1;$
**for** $a[i] := 1.0, 2.3, 4.8$ **do**
        $i := i + 1;$
```

Secondly, it is not entirely clear that constructs like the above are valid since the Report speaks of "*the* controlled variable". And lastly, the code generation is substantially complicated by the fact that the controlled variable is no longer given by a simple address. The example given above assumes that the controlled variable is $a[i]$ which is re-evaluated after the execution of the controlled statement. Some compilers calculate the address of the controlled variable only once, although any simple expansion of the **for** loop in other terms would require that the address was calculated at least twice per loop. Several compilers avoid these issues completely by restricting the controlled variable to be an unsubscripted variable (although similar problems arise when the controlled variable is a name parameter).

The difficulty with the code generation arises because the controlled variable appears only once in the source text, but code to access it is required several times in the generated program. The difficulties can most easily be seen with the **step-until** element, where accessing code is required three times. In the worst case, the subscript could be very complex, forcing the compiler to use a subroutine. As this is a rare occurrence in any case (see 2.3.1.5 and 2.3.3.1.1) the use of a subroutine for every subscripted variable would be quite reasonable.

With such an approach, the example given above, would use two generated subroutines, one for the controlled variable, the other for the controlled statement, viz:

```
      JMP       L1
CV:   LDI,1     I       address of a[i] left in register 1.
      ADI,1     A       link left in register 3.
      JPI       REG3
L1:   CALL,3    CV      := 1.0
```

```
        LDA       1.0
        STA       1,(0)
        CALL,3    CS
        CALL,3    CV      := 2.3
        LDA       2.3
        STA       1,(0)
        CALL,3    CS
        CALL,3    CV      := 4.8
        LDA       4.8
        STA       1,(0)
        CALL,3    CS
        JMP       L2
CS:     LDI,1     I       controlled statement code
        ADI,1     1       link left in register 3
        STI,1     I
        JPI       REG3
L2:
```

Only four instructions are generated per for list element, and the number executed is good partly because it is assumed that short subroutines do not cause the subroutine link to be stacked.

7.5 Function and Name Calls

The generality in the design of ALGOL 60 is such that a procedure or name call can occur in most language constructs. Because of the possibility of side-effects or Jensen's device, both these forms of call must be treated very carefully. To produce good code some reordering of the generated code from its position in the source text is desirable. An important example of this is that the array address calculation of the left hand side of an assignment statement is best performed after the expression evaluation. There are three reasons why this cannot be done in general, firstly a multiple assignment involving an integer subscript (see 1.6), secondly a name call and lastly a procedure call.

If a function or name call does occur, then in the critical cases, the code must be executed in the manner determined by the Report. Thus all partially evaluated results must be preserved, the call made, and then the results restored. This militates against preserving partial results in registers, except in the case where function calls are not present.

The net effect of procedure and name calls is that the majority of ALGOL compilers put very few partial results in registers. Hence the code produced often appears to be using temporary storage in an unnecessary manner. The easiest way over this problem would appear to be a form of window optimisation of the simple code (see 10.3). Short sections of code which generate and then use a temporary storage location can have this location equated with a register. Although this approach necessitates a subsidiary pass, it is reasonably simple to specify and to ensure that it is logically sound.

7.6 Bound Pair List Evaluation

Array declarations are exceptional to the extent that code must be executed for them unlike the declaration of simple variables. Assuming that the source text is not reordered in any way, this implies that jumps over procedure declarations must be arranged so that the array declaration code is executed. Also, the compiler must treat the bound pair list somewhat differently than other expressions in that block, because access to locals is not allowed (Report 5.2.4.2).

Array declarations are rather different from ordinary statements because, although the block has been entered, the second order working storage has not been completely allocated. The block entry mechanism therefore sets up the link data and first order working store (if necessary see 1.8), whereas the array declaration code allocates the second order working store. These two functions cannot, in general, be combined because the bound pair list evaluation could be very complex. It could involve a function which executed a jump out of the block.

The final complication is that of preserving the bound pair list expression values for the array declaration subroutine. These values can either be placed in a block of storage in the first order working space (the user-declared variable space could be used) or on the run-time stack. Care must be taken with the use of the run-time stack if code words are planted with the array itself, as the array will almost certainly use the same store in run-time stack as the bound expression values.

7.7 Labels by Value

In the Report (2.8), it states that a label is an example of a **value**. The mechanism of value assignment explicitly refers to section 2.8, so presumably labels can be called by value. As with arrays, the value assignment cannot be accomplished with an ordinary assignment statement.

The actual parameter corresponding to a label parameter can be a designational expression, or if integer labels are permitted, an arithmetic expression. In practice, the actual parameter is likely to be a simple label, but it could involve an arbitrarily complex Boolean expression or integer expression (as a switch subscript or in a conditional designational expression).

With the value mechanism, the label resulting from the designational expression must be found without the **goto** being performed. This means that the **goto** mechanism itself is usually divided into two parts, label evaluation and the jump (see for instance Randell and Russell, Section 2.4). The actual information passed to the procedure corresponding to a value label is therefore the data required to perform the jump—a machine code address and environmental information (see 1.9.6 and 1.11). In contrast, a label by name will give the details of a designational expression thunk which, when evaluated, will give the label information. The jump will not be performed because this may not be required—for instance, if a formal label by name is passed as a parameter to a second procedure which passes the parameter by value.

All the above is of little concern in most practical programming so it is not surprising that some compilers (1900, for instance) exclude labels by value. Their implementation appears to place a burden on the compiler rather than the run-time system (the exact opposite of value arrays).

One can easily appreciate the difference between name and value labels with actual parameters of the form

if $sqrt(-1) = 0$ **then** $l1$ **else** $l2$
or $s[sqrt(-1)]$

The value mechanism does have the advantage of validating the designational expression at the earliest possible moment. On the other hand, as labels are often used for error conditions, evaluation of the label may be unnecessary anyway.

7.8 Switches and Wild Gotos

Having introduced label values in the manner described in 7.7 and 1.11, the implementation of even the most exotic **goto** is straightforward. A dubious use of a goto is to jump out of a function, but there does not appear to be anything in the Report to exclude this, and in any case, it is virtually impossible to check as an arbitrarily long chain of further procedures may be involved before the goto is performed. In such cases, the function does not give a value, and any temporary variables used in the expression in which the function was called, must be removed. This is essentially automatic in the method given in 1.11. If temporary results have been stacked (say, some parameters) then resetting the stack front on the jump will have the desired effect. It is interesting to note that this is not so on the B5500, where jumps out of functions are excluded, although a complete check is not made by the compiler.

A further complication with jumps is the implementation of an out of range switch index. If the evaluation of the label yields such an index, then the jump must not be performed (Report 4.3.5). The majority of compilers do not implement this (as recommended by the IFlP subset), although the Report is most explicit on this point. The implementation is relatively straightforward in that the label evaluation must produce a special “dummy label”. The action of performing the jump must test for this “dummy label” so that if it occurs, the jump can be omitted.

7.9 Own Arrays

The implementation of simple own variables is very straightforward. They can be given space as for a global variable, but the scope is, of course, local to the block in which it is declared. Any difficulty there may be resides with the compiler which must allow for allocation of storage for **own** variables in a non-standard fashion, compared with all other variables.

Own arrays are more difficult for two reasons, firstly, the space to be allocated cannot necessarily be determined at compile-time, and secondly, on re-entry to the block, the storage mechanism must be reactivated because of the bound-pair list evaluation. However, the request for storage is not necessarily the same as on the first occasion, which gives rise to numerous problems, the most obvious being to decide what such a construction means. Not surprisingly, many compiler writers have side-stepped this issue by restricting the bounds of own arrays to be constant. This allows the compiler to determine the store required, so that space can be allocated globally as for simple own variables. This is the method adopted by the Whetstone compiler (Randell and Russell, 1964, Section 3.4.1.2.1.2). On first entry to the block containing the own array, the array segment bound pair list is evaluated, and the storage mapping routine allocates space in a global area calculated by the compiler. The syllable Avoid Own Array is also executed, which overwrites itself into an unconditional jump so that on re-entry to the block the array segment code is avoided (including the bound pair list evaluation). The Kidsgrove compiler uses a slightly different technique, which in theory allows greater flexibility. Space is not allocated for the array by the compiler, but rather space is claimed from the top end of store in the opposite direction from the stack. The array word is treated as a global variable as with the Whetstone compiler, thus ensuring that re-entry to the block reuses the same array space. Repeating the array storage allocation is avoided by overwriting the calling sequence of the run-time subroutine. On re-entry, the array bounds are re-evaluated, but a different entry to the storage allocation routine merely unstacks he bound information.

The implementation of own arrays with dynamic bounds is very complex, and barely worth attempting from the practical viewpoint. One significant difficulty is that stack allocation of storage is not adequate so that a second method must be used. Two reasonable interpretations are possible if the bounds change, first that an entirely new array is required and secondly that values within range of both subscripts should be copied over into the new array. This second interpretation is adopted by the B5500 system, which is one of the few compilers to attempt implementation. The task is made substantially easier by the segmentation system, where in fact, all arrays are stored off the stack. In the ALPHA system, Yershov (1971) reports that dynamic own arrays, although implemented, were never used by a practical program in a whole year's use of the system. Drum storage was used to overcome the allocation problem.

Unlike some deficiencies in ALGOL, own arrays, however generally they are implemented, do not appear to have a detrimental effect upon the other parts of the implementation. In practice, of course, own variables are made virtually useless by the lack of any feature to initialise them. However, the introduction of such a feature would create just the same problems as that of the bound pair list of own arrays.

Much discussion has taken place between ALGOL specialists on the exact meaning of own variables (Knuth, 1961). The "static" interpretation has been followed here because of the ease of implementation. Programs which distinguish between various interpretations have almost certainly been written as such rather than as genuine application software.

7.10 One Pass Translation

Several compilers have been written for ALGOL 60 which produce machine-code or a near derivative in one pass. Clearly some addressing cannot be determined until the exact size of the program parts are known, but such details can often be handled by the link-editor. A few compilers accumulate the compiled program within their own core space, enabling changes to be made when more details of the program structure become available. Such a technique might reasonably be called $1\frac{1}{2}$ pass. The Whetstone compiler uses this technique. Type information is not available in general for variables so skeleton code is produced, with chaining back of the unknown address to a name-list entry. When full details become available (usually on an end of a block when the name list is collapsed), the skeleton operations are filled in. This completion could be performed by a loader, since no information required for the compilation process is deduced from it.

The Whetstone system requires very few restrictions on source text so that it is simply not possible with $1\frac{1}{2}$ passes to resolve all the problems necessary to generate tolerable machine-code. Hence the interpreter must check types dynamically, not only for procedure parameters but also for arithmetic expressions. This is far too high a price to pay except when high translation speeds are paramount and no restriction can be applied to the source text.

Faced with the problem of producing machine-code in one pass, one must either restrict the language or devise a language extension to provide the information that is lacking with the single scan. The Elliott family of compilers (Hoare, 1964) and the compiler produced at the Royal Radar Establishment, Malvern for R.R.E.A.C. adopt the restrictive approach. The source text is ALGOL 60 but substantial restrictions are made, the most important of which is that no use may appear before the corresponding declaration. With most practical programs of the less sophisticated type, this is fairly easy to arrange by placing simple and array variable declarations first, then switches and procedures in the order determined by the switches and procedures accessed within them, but with larger programs this restriction becomes increasingly awkward. Also ALGOL programs produced elsewhere are unlikely to satisfy the restriction so that substantial editing may be necessary (if it is possible, even the author may not know if his program can meet the restriction).

The approach which includes extensions must provide some method for allowing the user to inform the compiler of the existence of switches and procedures before the body of the declaration occurs in the program text. The B5500 provides for such a method of forward declaration. In fact, the declaration of labels is also demanded by the compiler, even for those which appear in a switch list. In contrast, some Elliott compilers demand declarations of labels by insertion in a switch list. In order to produce efficient code for procedure calls it is important that forward declarations of procedures should include the specification of the parameters. The first compiler produced for the Univac 1100 series omitted any specification of parameters, and so produced very poor code for procedure entry.

Forward declarations on the B5500 are used as a documentation aid with large programs by some users. The program text starts with an alphabetical list of the procedures as forward declarations. Such a listing could be produced automatically by a source text analysis program (see 6.1), but it is a substantial improvement upon mere comment since its validity is checked by the compiler.

The actual form of the forward declaration is usually similar to a procedure heading allowing the same compiler routine to be used. The forward declaration can be in the form of an interpreted comment so that the extension need not be regarded formally as such. An alternative approach is for the declaration to be bracketed by the two symbols **forward** and ; so that the ALGOL purist can regard **forward** as an alternative representation of **comment**. Such methods are very convenient if program portability is a critical issue since it means that conversion can be accomplished at basic symbol level, which is in any case the natural unit of text translation for ALGOL 60 programs.

The degree of optimisation that can be accomplished with one pass translation depends critically on the amount of buffering within the compiler. Since there is inevitably quite a large "buffer" for remembering the current syntactic context, a buffer of 50 or so words of instructions or its equivalent is not unreasonable, allowing "window" optimisation described in 10.3. Of course, if the compiler accumulates as much as a whole procedure before producing code then almost all important forms of optimisation are possible, such as loop optimisation. It is interesting to note that both the B5500 and RREAC compilers are one pass but do manage to assign storage at procedure level.

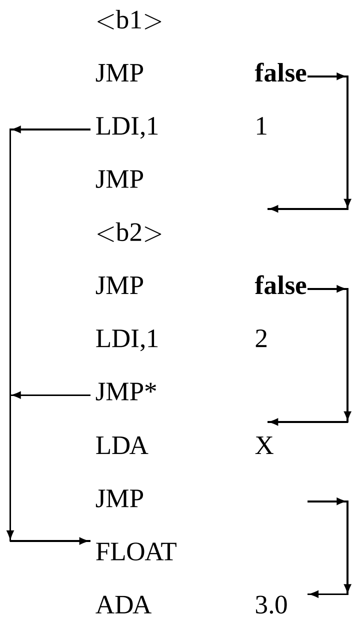
One pass compilation is usually chosen to give very fast compilation times. Although one pass is necessary it is certainly not sufficient for fast compilation. The first ALGOL compiler for the Univac 1100 series was one-pass, but its successor was faster and multi-pass. Introducing a pass into a compiler is bound to require not only peripheral capacity, but also processor time in the basic input-output handling routines. Because of this overhead on a pass, it is desirable to spread the processor load between the passes of a multi-pass system. The major difficulty with one-pass compilers is that the size of the code becomes very big. Their speed, combined with their core requirement means that the operating system must support such programs by reducing the multi-programming factor and avoiding the swapping of the compiler if it can possibly be arranged. A totally different operating system behaviour is desirable with a multi-pass compiler which is likely to be peripheral limited and is of such a size that several programs could multi-program with it. Manufacturers who must produce compilers for a large range of configurations often favour a multi-pass type because the minimum core requirement is much less.

There is also a logical objection to one pass translation in that it is very easy for *ad hoc* methods involving both the syntax and the semantics to creep into the coding. The interface between compiler passes, enforced by the overall design, means that such coding tricks are more easily shown up to be undesirable and are hence avoided. On the other hand, the interface between compiler passes raises a number of severe design problems which do not have to be faced with one-pass compilation. Also, these interfaces have to be finalised before the compiler is coded in earnest.

Assuming that a forward declaration system is used, then parameter handling can only depend upon the specification. Therefore one must assume that an assignment will be made to a name parameter. This in turn means dealing with the problem of integer-real type conversion, even if this facility is not used (see 1.10.3). Integer-real type conversion also causes trouble in simple expressions if code is produced without any buffering. Consider

$$i := (\textbf{if } b1 \textbf{ then } 1 \textbf{ else if } b2 \textbf{ then } 2 \textbf{ else } x) + 3$$

Because of the **real** x, both 1 and 2 are required in floating point, but this is not known when the code is generated. Hence one is likely to generate



```
<b1>
JMP       false
LDI,1     1
JMP
<b2>
JMP       false
LDI,1     2
JMP*
LDA       X
JMP
FLOAT
ADA       3.0
```

The FLOAT instruction could have been better placed at the point marked with a star, but this is unlikely since a complex expression could have been used instead of x so that its real type could not have been detected immediately. In fact this example is a controversial issue in ALGOL 60 because the rules of expression evaluation (Report 1.3.3.3) suggest that integer addition should be used unless x is evaluated, requiring a dynamic type check.

Chapter 8

Testing an ALGOL Compiler

As Dijkstra has observed it is never possible to prove that a program is correct from its behaviour, but only to show the presence of bugs (Boon, 1971). Nevertheless, a sufficiently large and carefully constructed set of test programs can exercise a high proportion of a compiler code to give one more confidence in its correctness. At the very least such tests serve to demonstrate that ‘ordinary’ programs are compiled and executed in an apparently correct manner, while a reasonable aim would be to show up all the known errors in some existing systems.

It will be some time before correctness proofs are available for anything as large and complex as an ALGOL 60 compiler, so that in the intervening period one must be satisfied with behavioural testing. Such tests have been written for COBOL (AFSC, 1970) mainly to demonstrate compliance with the language standard.

8.1 Reference Manual Standards

A compiler must clearly be judged against its specification. This specification is usually in the form of a reference manual. The exact form this takes varies markedly, in some cases consisting of a teaching manual as well, running to over a hundred pages. At the other end of the spectrum, a short list of the differences from the Report, error message layout and hardware representation may suffice, being no more than ten pages. The B5500 manual consists of a language definition in BNF followed by examples and the semantics in an identical manner to the Report. The KDF9 manual lists a number of changes to be made to the Report which will define the subset of ALGOL 60 acceptable to the compiler. This is the procedure adopted for the definition of the IFIP subset (IFIP 1964a). An alternative approach is a simple list of those features of ALGOL 60 which are not implemented, usually added anyway, since it is substantially clearer to say that integer labels are not allowed rather than

$$\langle\text{label}\rangle ::= \langle\text{identifier}\rangle$$

In addition to this, the manual often specifies some limits on the complexity of programs that can be compiled. If a display is kept in registers, then there must be some limit on the depth of nested blocks. There are about twenty such features in ALGOL 60 which are likely to impose some restrictions. Compilers which do not give any limit should be very carefully tested since it is unlikely that such restrictions do not exist (see chapter 7).

The manual must, of course, specify the hardware representation used. In most cases this is a simple table of the reference language and the representation. One is left to guess how the concatenation of characters into basic symbols is performed. This is a valid question since it affects the way in which the compiler handles various error conditions. It is important to note the extent to which format-effector characters are allowed internally in the basic symbols. With escape characters one may be able to write

’INTEGER ARRAY’ for ’INTEGER’ ’ARRAY’

On the other hand, one compiler on reading a quote takes the first four characters (before the next ’) to determine the symbol and ignores the rest. Hence the first representation above would be read as **integer**! From the point of view of simple recovery in the face of errors, it is advisable not to allow a newline internally in basic symbols. However, it is usual to allow both **goto** and **go to** and possibly a space between : and =. An ISO proposal for a hardware representation for ALGOL 60 uses an escape character to delimit compound words. Unfortunately it makes no mention of the positioning of format effectors. It also contained an error in the sense that E is allowed for $_{10}$ which means that $_{10}6$ cannot be distinguished from the identifier E6. Although the proposal uses the 7-bit ISO code which includes both upper and lower case, only one alphabet is used.

Tests can obviously be devised to discover how various misrepresentations are handled. It is impossible to produce these tests in the same manner as those which involve only well formed ALGOL basic symbols since they are at a lower level which is machine dependent. Above this level, it should be possible to construct machine independent tests which can be converted automatically on a strict symbolic basis.

8.2 Construction of the Tests

Such tests must be constructed with great care for the following reasons.

1. No existing compiler can be used to check all the tests.
2. The interpretation that can be placed upon the test should as far as possible be unambiguous.
3. The tests should cover all the important syntactic and semantic elements of ALGOL 60.
4. With the possible exception of the part of ALGOL 60 under test, all the features used in the test programs should be non-controversial.
5. Some input-output must be used over which there is no general agreement.

As far as the completeness of the tests are concerned, a number of techniques can be used. For any language feature, a particular implementation can be taken. Tests can then be constructed which will exercise every path in the algorithm used in this implementation. This method of construction immediately pin-points a major weakness in the tests, namely that they are built round a particular implementation technique which might be quite inappropriate to another method. However, it does guarantee that most of the salient features have been tested. To take an example, the Whetstone ALGOL system could be used. Tests must therefore be written to compile and execute each of the elementary operators of the interpretive code. Errors will probably be found with this method, since it is known that several operations were never executed with a 155 million operation executions of ordinary programs. Tests for the Whetstone compiler in particular will be heavily dependent upon the one-pass technique used. In practice it has been found that the major source of error in the compiler was incorrect deduction of the type of an identifier during the scan of the source text before the actual declaration was encountered. The number of possibilities here is very large since the type of an identifier can be deduced in several stages. A correspondingly large number of cases arises in the interpreter in the checking of formal-actual correspondence of parameters. This is best illustrated with an actual algorithm of reasonable size. Take as an example the procedures given in 1.1.4.2 for exponentiation. All paths through the two procedures are followed by 8 cases

				Answer
2.0	$\uparrow$	2.0	:	4.0
0.0	$\uparrow$	2.0	:	0.0
(-1.0)	$\uparrow$	1.0	:	undefined
1.0	$\uparrow$	0.0	:	0.0
1.0	$\uparrow$	0	:	undefined
0.0	$\uparrow$	(-2)	:	$1/(0.0 \times 0.0) =$ undefined
2.0	$\uparrow$	1	:	2.0
(-2.0)	$\uparrow$	3	:	-8.0

In this particular case the Report (section 3.3.4.3) gives a table of distinct cases which amount to 18 when all the type information is considered. Test cases for the Report implementation might be

				Answer
(-2.0)	$\uparrow$	2	:	4.0
3	$\uparrow$	3	:	27
3.0	$\uparrow$	0	:	1.0
3	$\uparrow$	0	:	1
0	$\uparrow$	0	:	undefined
0.0	$\uparrow$	0	:	undefined
2.0	$\uparrow$	(-2)	:	.25
2	$\uparrow$	(-2)	:	.25
0.0	$\uparrow$	(-1)	:	undefined
0	$\uparrow$	(-1)	:	undefined
2.0	$\uparrow$	(-3.0)	:	$exp(-3.0 \times ln(2.0)) \approx .175$
2	$\uparrow$	(-3.0)	:	$exp(-3.0 \times ln(2)) \approx .175$
0	$\uparrow$	2.0	:	0.0
0.0	$\uparrow$	2.0	:	0.0
0	$\uparrow$	(-1.0)	:	undefined
0.0	$\uparrow$	(-1.0)	:	undefined
(-1)	$\uparrow$	2.0	:	undefined
(-1.0)	$\uparrow$	2.0	:	undefined

It is important to note that some deviation from the Report is to be expected since otherwise dynamic type checking is implied. Unfortunately it is impossible in ALGOL to determine the type of an expression so that one cannot be sure that the type is correct.

Testing equality of a real result must be done with caution owing to the effects of round-off. Since some compilers evaluate constant expressions at compile time, care must be taken to allow compilation of several tests within one program.

The existing systems noted below force restrictions on these test programs as follows:-

- no spaces in identifiers (Univac 1108)
- no identifiers which could be reserved words (1108, B5500)
- no spaces in constants (1108)
- no recursion (IFIP)
- no type changing on call-by-name assignment (KDF9)
- no label parameter to a function or goto out of a function (B5500)
- no strings (B5500)
- no parameter comments (B5500)
- no upper case alphabetic characters (B5500, CDC 6000, 1108, etc)
- six significant characters in an identifier adequate (IBM, level F)
- not more than 32 characters in an identifier
- no use before declaration (Elliott, B5500)
- no integer labels (Atlas, 1900, KDF9, B5500)
- no side-effect on function calls (ALCOR).

8.3 Tests to Demonstrate Language Features

Tests can be written in a virtually identical manner to that proposed for exponentiation above, for many language features. For instance, the following would test a quite large part of ALGOL 60:

- boolean operators (exhaustive)
- standard functions *abs*, *sign* and *entier*,
- integer divide
- precedence of operators
- integer to real type conversion
- real to integer type conversion
- call-by-name type conversion
- Jensen's device
- examples of all three types of for list element

- all possible formal parameters and corresponding actual parameters
- general designational expression in **goto**, switch list and as parameter
- labels in all possible places
- dummy statements in all possible contexts.

A check should be made that every valid consecutive pair of delimiters appears somewhere in the tests. This should ensure that compilers which are driven by the delimiter structure execute the major parts of the compiler code.

8.4 Complexity Tests

One needs to check that a compiler will accept programs with a reasonable degree of complexity. The easiest measure of complexity to quantify is that of the lists appearing in the ALGOL syntax. With a slightly more powerful notation than BNF, these lists would be explicit, but in any case, they are easily identified. Each list can be tested with an example of three times (say) the largest length that is typical of most programs. Suggested limits for each list appear below

digit length	60
string length	300
identifier length	60
switch length	300
left-part list	15
for list	60
array list	60
array segment list	60
declaration list	300
labels to statement	6
array dimension	12
parameters	60

nested depth of the following

procedures	6
blocks	15
for loops	12
conditional statements	24
conditional expressions	9
designational expression	6
arithmetic expression	15
function call	9

Such tests have been constructed by the author with the aid of a macro-generator ML/I (Brown, 1967). They are designed to print one number merely to provide some very crude indication that correct machine-code was generated. In addition to this, a check should be made that complex expressions can be placed everywhere where an expression is valid. Again, such a test can very easily be constructed with a macrogenerator.

8.5 Numerical Tests

Several aspects of a compiler involve processing real quantities which can be subject to a test. Real constants within a program should be converted to floating point form with the minimum of error. This is not as straightforward as it might appear due both to the wide variety of forms of a real constant and to the pitfalls in the conversion unless double length calculation is used. Excess digits must be handled correctly since algorithms written for general use must necessarily quote sufficient figures to allow for the most accurate machines. A further source of error is a large number of zeros after the decimal point, viz.

$$.000000000001234567891234$$

The initial zeros should not detract from the accuracy of the representation of the number since these merely contribute to the exponent part.

Apart from the reading of constants within the program text, the input-output scheme will also have the same difficulties. Consistency between a value for input and those in the program text is obviously desirable. It should also be possible to ensure that input and output are exact reciprocals by specifying sufficient accuracy. Unfortunately many input-output schemes specify a limit on the number of digits output within a number, effectively prohibiting such tests in a straightforward manner.

In all of the above tests one ought to differentiate between an error due to lack of rounding and an actual coding error which could result in a complete loss of accuracy. To do this one needs to be able to determine the characteristic features of the floating point facilities (see Malcolm, 1971). An environmental enquiry is really required (see Naur, 1964).

The standard functions *sqrt*, *sin*, *cos*, *arctan*, *ln* and *exp* can all be tested for the quality of their numerical results. Presumably such routines are expected to yield a value very close to the floating point y such that $y - f(x)$ is a minimum. Since one cannot assume the existence of any double length facilities, such tests are not easy to devise, and because one merely wishes to demonstrate that the functions have the right sort of behaviour, the easiest approach is to use the numerous functional identities. As *sqrt* is defined by an identity it is an obvious candidate for this approach. On a modern computer, several hundred values distributed both uniformly and randomly over the range of applicability can be checked in a few seconds. The *sin* and *cos* routines can be checked for mutual consistency, but since this is likely to be used in their construction no errors can be expected. There are a large number of trigonometric formulae which can be used, but $sin^2 + cos^2 = 1$ and the double angle relations are obvious candidates.

The functions *ln* and *exp* can obviously be tested against each other, although in the case of an error it will not be clear which one is to blame. Identities such as $ln(x \times y) = ln(x) + ln(y)$ are better in this respect, especially since it gives a method of relating values with different arguments.

8.6 Tests to Evaluate Diagnostic Features

Incorrect programs will frequently be submitted to an ALGOL compiler. It is most important that such programs should be handled in a manner to minimise the resources of both the computer and the man. Thus errors should be located at compile-time if at all possible, and if not, then at run-time. Programs should never lose control, attempt to obey invalid code or go into an infinite loop if diagnostic options are being used. Unfortunately, it is necessarily the case on most third generation computers that some programs must be run without bound checking which inevitably means that some data-dependent bound violations will occur with running programs.

Scowen (1972) has produced a set of tests designed to exercise the diagnostic features of a system. The programs have been divided into four classes according to the expected action, i.e. (1) failure at translate, (2) at run-time, (3) legal, but worthy of comment by the compiler (4) test of the rigour of the ALGOL 60 implementation. Since the last two classes are handled more extensively by the other sections of this chapter, only the first two are considered here.

The programs contain errors which are more or less typical of those to be found in programs during development (see 5.4.5). The tests have been run on sixteen different ALGOL-like systems and so it is possible to give some indication of the relative merit of the compilers. A major difficulty with such an intercomparison is that a subjective judgement must be made. Weight should be given to the clarity of the message, the ease with which the offending code can be located and any additional information that could even lead to the location of further errors (this is one advantage of a post-mortem dump).

The results indicate a very significant difference between compilers, in a similar manner to that of the statement times. However, there does not appear to be any correlation between speed and diagnostic power. The approximate results were

ALGOL W, ALCOR, KDF9(Whetstone), 1900(XALT), ALGOL 68-R, Babel	good
IBM level F, Atlas, KDF9(Edgon)	average
System 4, 4100, GIER 4, X8(Eindhoven)[1], GEIS/Honeywell(GE 235)	barely adequate
XDS 9300[2]	unsatisfactory

As with the statement times, quite significant advances can be made in new releases of a compiler, so these tests should be re-run if the results are at all critical.

8.7 Tests of Exotic Features of ALGOL

Throughout this book many references occur to parts of ALGOL 60 which cause difficulty in the compilation process. Programs which use these features are naturally rare, and thus some defect in the compiler may not become apparent for years after its construction. Test programs which exercise these features are particularly useful since they can allow the rapid detection of such defects.

To illustrate this point it is merely necessary to list the exotic features already mentioned and the reference:

integer divide with variously signed arguments	1.1.4.1
$<$integer$>\uparrow<$integer$>$	1.1.4.2 and 8.2
step-until for list element	1.7.2 and 7.4
environment control	1.8 and 1.9

[1]Compiler performs well except for error handling on the standard functions, i.e. *sqrt*(-1), etc.

[2]The author understands that this compiler is no longer supported.

label parameters	1.9.6
real-integer conversion on name-call	10.3
dimensionality of formal arrays	7.1
formal procedure parameters	7.2
arrays by value	7.3
arrays by value with type-change	7.3
labels by value	7.7
switches and wild **goto**s	7.8
syntax of comments and strings	4.1.1
labelled comment	4.1.1
left-hand side assignment, i.e. $i := a[i] :=$	1.6
side-effects on function calls	9.8

Many examples of programs which could cause trouble to compilers have appeared in the literature, all of which could be collected into such a set of tests.

A major difficulty with such tests is the interpretation of the results. If the compiler or run-time system merely fails in an abnormal way, then it is obvious something is wrong. On the other hand, many cases rest upon an interpretation of the Report which may be far from clear[1].

[1](added in proof) Tests such as proposed here are now available from the author (National Physical Laboratory Report NAC 33).

Chapter 9

Six ALGOL Compilers

The study of performance detailed in Chapter 2 revealed wide disparities between the expected and observed times for individual statements, which can only be clarified by an examination of the machine-code. The author selected four systems with a wide range of computer architecture which were conveniently available. These were Atlas, KDF9 (Kidsgrove), 1900 (XALT), and the B5500. P. T. Cameron of Univac Ltd. kindly ran the tests on the NU 1108 compiler, and J. Wielgosz of the University of London ran the same tests on the CDC 6600 and 6400 (version 2.0). The six machines in question cover all the major types of machine architecture except the parallel processing machines such as Illiac IV and CDC's Star.

The tests consisted of four sample programs. These were converted automatically from the KDF9 source text to avoid any possibility of mispunching. Four characteristics were measured for the programs: the number of instructions compiled, the number of instructions executed, the size in bits of the compiled code, and the execution time. It is clear that the compiler writer can often exchange time and space for the size of compiled code by the use of subroutines, so by measuring these four characteristics the choices made should be apparent.

9.1 The Test Programs

The four programs were as follows:-

Quickersort

The program consists of the CACM Algorithm 271, Quickersort (Scowen, 1965). The algorithm uses some rather complex **for** loops to achieve the sort. Tests for inequalities, use of both a formal array (the one required to be sorted) and two local arrays make it a reasonable test.

Apart from the procedure *quickersort* itself, a procedure *random* is used to generate numbers to sort which is called by the procedure *inarray*. A procedure *checksort* makes a check that the sort has been successful. All four characteristics are measured for the three procedures *inarray*, *quickersort* and *checksort*. Since *inarray* merely contains one **for** loop which calls *random*, it is a test of the speed of procedure calls. The procedure *random* uses a linear congruence with integers of less than 24 bits so as not to risk overflow.

Compiler tests

This program consists of an amalgam of 13 separate compiler test programs. The programs were obtained by the author from M. Woodger. Many of them were written by Randell and Russell to test the Whetstone compiler and most use at least one exotic feature of ALGOL but do not cover all such features.

The object of including these tests is mainly to see if they compile and execute correctly. The machine code can also be examined to see how the "general case" is handled. The number of instructions compiled and the code size in bits is measured, but not the other two characteristics.

The last of the 13 tests is somewhat different, it is "Man or boy?" (Knuth, 1964b). It consists of two small procedures one of which is recursive. It uses call-by-name very extensively, and is a very severe test of the speed of environment control with thunks and procedure calls. This test is an exception in that the execution time is substantial and so it is measured.

GAMM

This is a coding of the GAMM loops which are sometimes used as a measure of processor performance (Heinhold, 1962 and 2.5). Since the test is only functionally defined, it can be coded in any language. It cannot be regarded as an effective measure of computer performance since it only uses a very restricted subset of the facilities available on modern computers. When it is coded in ALGOL 60, it is a severe test of one dimensional array access and simple for loops—but nothing else. The loops are extremely straightforward and so can be optimised very extensively.

The program repeats the test twice, once with formal arrays to a procedure, and once with the actual arrays. The four characteristics were measured for the five loops both with and without formal arrays. In fact, no significant difference was noted with having the arrays formal, so only one case is considered.

Timing

The last program consists of the 41 statements used in the timing described in 2.2. It is obviously advantageous to add these statements to the more detailed tests since a lot of information is available about their expected performance. It is possible to see if any special coding tricks have been used, which could invalidate the ALGOL mix. All four characteristics are measured for all the 41 statements together with the for loop control code in

for $i := 1$ **step** 1 **until** n **do**

To describe the actual machine-code generated it is necessary to devise a suitable notation. Since it is unreasonable to expect familiarity with all six assembly languages, the actual code is given in a stylised form of ALGOL. Each "action" is given in terms of ALGOL, using capital letters to identify parts of the processor (registers, etc.). Small letters are used to denote core-store addresses corresponding (usually) to variables declared by the programmer. If more than one action is involved in one instruction, each action is separated by a comma, the semicolon being reserved for the end of a generated instruction.

	Atlas	1108	KDF9	1907	BSSOO	CDC6600
Instructions compiled	1.0	0.41	0.78	0.65	0.63	1.20
Compiled code size in bits	1.0	0.31	0.35	0.32	0.16	0.56
Execution time	1.0	0.28	3.50	1.20	1.80	0.28
Instructions executed	1.0	0.83	1.50	0.94	0.96	1.40

Table 9.1: Compiler Summary

9.2 Summary of the Analysis

Each characteristic was assumed to be multiplicative in two factors in a similar manner to the statement times. For instance

$$\begin{aligned}&\text{(Number of instructions generated for S on machine A)}\\&\quad\approx \text{(Instruction factor for S)} \times \text{(Instruction factor for A)}\end{aligned}$$

where the factors are calculated in the manner given in 2.2.4. It is important to note that since some characteristics were not measured, it was necessary to estimate values based upon the model. In a few exceptional cases the estimated value based upon the model was inappropriate. For instance. if the number of instructions executed was not known but the time was, then a better estimate could sometimes be obtained from the time rather than that from the model. It was necessary to do this in a few anomalous cases otherwise the figures would not be self-consistent. This was important because one statistic derived from the data was the average time to execute a machine instruction in a compiled ALGOL program (for a particular machine).

As with the statement times Atlas is taken as the scaling factor, the complete set of factors appearing in Table 9.1.

The average times taken to execute an instruction, although subject to the errors mentioned above were Atlas 3.2, 1108 1.0, KDF9 7.4, 1907 4.0, B5500 5.8, CDC 6600 0.64 microseconds.

The factors can be more readily appreciated by the diagram with vertical logarithmic scales shown in Fig. 9.1.

The major reasons for the deviations are as follows. Atlas compiles more instructions because the compiler does not make much use of subroutines, which is only tolerable because of the paging. The 1108 does well because it makes good use of the auxiliary registers. In terms of code size, Atlas does badly because of the large instruction size of 48 bits, whereas B5500 does well having only 12 bits in an instruction. The time measurements are roughly in line with the previous measurements (see 2.2.3). Remarkably small variation occurs with the number of instructions executed. KDF9 does badly because it is a stack machine and, unlike the B5500, it does not contain other additional features required to exploit the stack.

The exact reasoning behind this summary will become apparent with the description of each machine and the generated code which follows. Every exceptional value, i.e. where the estimated and observed value of a characteristic differ by more than a factor of two, is detailed. The following sections give an outline of the machine architecture, the compiled code and then examples of actual code produced.

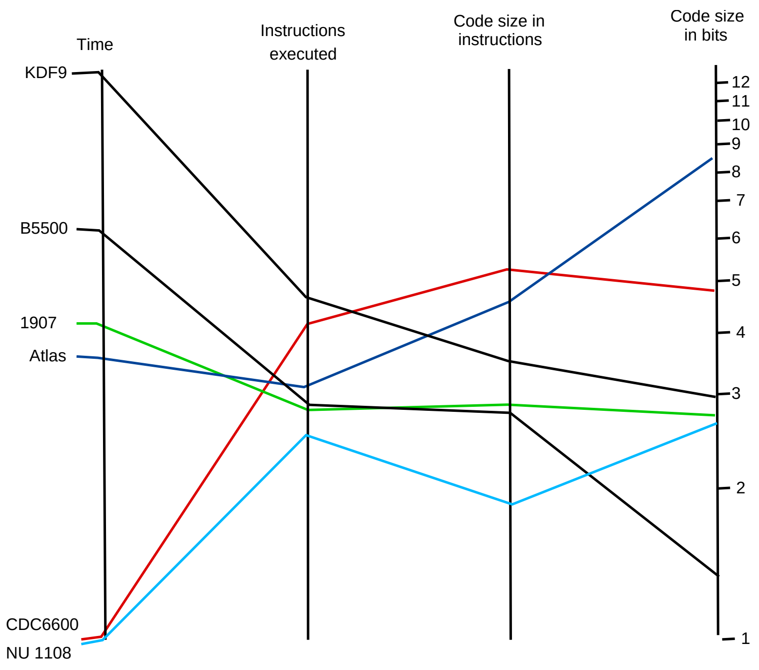


Figure 9.1: Six compilers — some characteristics compared

9.3 Atlas

This computer was the first machine to use paging and represented a very significant advance in computer hardware technology when it was first produced. Ideas from Atlas are to be found in many modern computers even though there has been a radical improvement in technology since then.

The processor has two main functional parts, the B-lines or index registers and the main accumulator. There are about 90 24-bit index registers for general use. The complete complement of registers is 127, but many are reserved for the supervisor while others have a special meaning—for instance B127 is the current instruction address register. There is a very large instruction set, especially of operations just on the B registers. In fact the computer would be quite useful even without the main accumulator. The instruction format allows access to two B registers and one main store address in one 48 bit word. The virtual address is very large at 24 bits, which is poorly exploited by the compiler (see 11.1). Some instructions are not executed directly by the hardware, but are handled by a supervisor routine called an extracode. These normally reside in the fixed store, and special hardware is provided to allow fast decoding of such extracodes. For instance one extracode consists of only six instructions and is included to complete the set of conditional jumps.

The main accumulator is 48 bits long and provides for the usual integer and floating point operations. Multiplication and division cannot be done in the B registers, so it is occasionally necessary to use the main accumulator for integer calculations, but this can usually be avoided. The core store can be conveniently accessed as 24 bit integers (for the B lines) or 48 bit floating point or integers (for the accumulator). See ICT (1965) for a more complete description of the machine.

The ALGOL compiler was produced in a very short timescale using the Compiler-Compiler of Brooker and Morris (see Brooker, 1967). The compiler is essentially two pass. The first pass does a syntax check, producing a tree-structure for the analysed program. This structure is then used to generate machine-code. The subset of ALGOL 60 it accepts is a large one, although many of the usual restrictions apply, viz. complete specification of parameters, no integer labels, no dynamic own arrays. The compiler is very robust and gives sensible messages in most circumstances. It does have a number of deficiencies—for instance, fixed sized tables in the compiler sometimes prohibit compilation of deeply nested constructs.

The compiler used was the version including a preprocessor to allow various dialects to be accepted (see Hopgood and Bell, 1967). The liberal use of the available index registers (denoted by B1,. . . B90) and the lack of subroutines makes the compiled code very easy to follow. The general strategy is to allocate B_i, for addressing the ith static block. B90 points to the first free location in the stack, and temporary results are usually placed on the stack although in a few cases the current environment is used.

The compiler tests revealed two minor faults. An **end** inside a string had made the preprocessor get out of step so that an incomplete program was presented to the compiler (which duly objected). The compiler failed to compile a procedure call which involved a type change from integer to real on a value array. The manual makes no mention of this as a restriction and indeed the code compiled for value arrays appears to allow for this contingency. All the other tests were successful and every error message produced was reasonably intelligible.

Four short statements are used for all six of the compilers to give an indication of the compiled code. It is easy to place undue emphasis on a few statements—every compiler produces reasonable code under some circumstance and very few do not have weaknesses which can make individual statements look appalling.

The first statement is one of those used for the timing experiment, namely $e2[1, 1]$:= 1. Atlas gives a classic example of the code word technique. Subscript brackets denotes address modification with the index register. The identifiers represent fixed address values corresponding to that identifier.

```
B79 := 1;
B79 := B2[e2] + B79;
B78 := 1;
B78 := B79[0] + B78;
B90[0] := B78;
ACC := 1;
ACC := ROUND (ACC);
B99 := B90[0];
B99[0] := ACC;
```

The compiled code has two major weaknesses. Firstly it does not spot that it is not necessary to preserve the address of the left hand side in the stack (i.e. B90[0]). Secondly the ROUND instruction could be avoided in many circumstances where it is generated by the compiler. However many dimensions are in an array, two index registers are always adequate. But if a subscript involves a multiplication or a further subscript, then the index registers in use will be stacked. B99, which is not necessarily preserved by extracodes, is only used for very short sections of code.

The second example is also an integer assignment statement

$$random := ri := ri \div 12601 \times 12601$$

where *random* is a function identifier and *ri* is global. This generates

```
     ACC := B2[$ri$];
     B90[0] := ACC;
     ACC := B2[$ri$];
     ACC := ACC/12601;
     if ACC $>$ 0 then goto $l1$;
     ACC := abs(ACC);
     ACC := ROUND (ACC);
     B90[1] := ACC;
     B90[1] := B90[1] and 1; (= B69)
     ACC := $-$B90[1];
     goto $l2$;
$l1$:  ACC := ROUND (ACC);
     B90[1] := ACC;
     B90[1] := B90[1] and 1; (= B69)
     ACC := B90[1];
$l2$:  ACC := ACC $\times$ 12601;
     ACC := $-$ACC + B90[0];
     ACC := ROUND (ACC);
     B4[random] := ACC;
     B2[$ri$] := ACC;
```

This illustrates the difficulty with integer divide if the instruction does not give the appropriate signs. The length of code is absurd since it could all be handled by a short subroutine. The variable ri is reloaded unnecessarily even if strict left to right expression evaluation is used. The example also shows the use of stack space in expression evaluation, B90[0] is the first location used, then B90[1], etc.

The third example is a real expression

$$y := a - i \times x \uparrow 2$$

where i is a local integer, x and y outerblock reals and a is a real function. The display is preserved and restored by open code in the calling sequence which is consequently very bulky.

```
B90 := B90 + 8;
B90[$-8$] := B2; B90[$-7$] := B3; B90[$-6$] := B4;
B99 := 2; B90[$-5$] := B99;
B99 := B127 + 2; B90[$-4$] := B99;
    $a$;
B90 := B90 + 8;
B2 := B90[0]; B2 := B90[1]; B4 := B90[2];
ACC := B79[0];
B90[0] := ACC;
ACC := B4[$i$];
B90[1] := ACC;
ACC := B3[$x$];
B90[2] := ACC;
B99 := 2; ACC := 1.0;
if B99 = 0 then goto $l1$;
```

```
        B99 := B99$-$1;
$l$2:   ACC := ACC $\times$ B90[2];
        **if** B99 $\neq 0$ **then goto** $l$2, B99 := B99 $-$ 1;
$l$1:   ACC := ACC $\times$ B90[1];
        ACC := $-$ACC + B90[0];
        B3[$y$] := ACC;
```

The power of the Atlas loop instructions is shown by the two instruction loop to calculate $\uparrow$. In this particular case the code is far from optimal because of the end condition tests. Note that B79, apart from its use in subscripting also holds the address of the result of a function evaluation. There does not seem to be any reason why the result cannot be left in the accumulator.

The last example illustrates a function call with a parameter in the middle of an expression

$$s := s + f(5)$$

where the parameter to f is integer by value. Atlas produces the code

```
ACC := B3[$s$];
B90[0] := ACC;
B90 := B90 + 1;
B90 :=B90 + 8;
ACC := 5; ACC := ROUND (ACC); B90[$-1$]:= ACC;
B90[$-8$] := B2; B90[$-7$] := B3;
B99 := 1; B90[$-6$] := B99;
B99 := B127 + 2; B90[$-5$] := B99;
    $f$;
B90 := B90 $-$ 8;
B2 := B90[0]; B3 := B90[1];
B90:= B90 $-$ 1;
ACC := B79[0];
ACC := ACC + B90[0];
ACC := ROUND (ACC);
B3[$s$] := ACC;
```

It is interesting to note that the stack is increased independently for two different reasons, first because of the partial result, and secondly because of the function call. This procedure call is in one less block depth than the previous example so that one fewer register is preserved. In fact, it is only necessary to preserve the elements of the display between the call and the procedure body, since the others are common to the calling code and the body.

The procedure used to calculate the overall measures of program size, etc., produces a residual matrix in an identical manner to that listed in Appendix 3. An inspection of the machine code should allow one to determine the exact reason for any anomalous values. The residual values greater than 2 or less than 0.5 are regarded as anomalous.

In terms of execution time, two tasks were handled well and three badly. Knuth's "Man or boy" had a ratio of 0.47 because holding the display in registers makes environment control very easy and direct. The test uses call-by-name and procedure calls very heavily, both of which involve environment control more than anything else. The other task handled well in terms of time was array declarations. This is achieved by use of open code—an enormous amount, to set up the code words and also sufficient information for bound checking. The three tasks handled relatively badly are **goto**, *sign* and *abs* (ratios 3.8, 2.8, 3.8). The times for *sign* and *abs* are long because the standard procedure mechanism is used even though the permanent routine which is eventually called is only 2 or 3 instructions. The time for **goto** is long because there is no optimisation. The subroutine which is called deals with jumping out of blocks and procedures, and label parameters. In fact, label parameters are handled incorrectly in that the first activation of a label is called and not the last with a recursive procedure. The subroutine also handles the implementation of a switch index being out of bounds yielding a dummy statement.

In terms of both instructions executed and instructions compiled $k := l \div m$ was bad (ratio 2.2), the reason for which can easily be seen from above. In all other cases, the exceptional values were in line with those for the times noted above. Code size in instructions varies much less than the other measures, and apart from integer divide only one further is exceptional. This is the dummy block **begin real** a; **end** (ratio 2.0), caused by the generation of code which places on the stack with the link data the address of variables to print on a post mortem. This code could be made more compact with little difficulty.

The main features of Atlas compiled code should now be apparent. The code is very large, even when the instruction size is taken into account. This could only be accepted with a large paged machine. Virtually no optimisation is performed, yet the product is a very useable one. The instruction address length of 24 bits is poorly used by the compiler, because every simple variable—or array word can be assessed by a small offset from a display register. The only exceptions to this are jumps (which could be but are not handled by signed offsets from the current address), references to the pool of constants, and the procedure address table. This is discussed further in 11.1.

9.4 KDF9

This computer is a contemporary of Atlas for medium sized scientific calculations. It is roughly comparable in power to the IBM 7090. The main memory is word organised with a 48-bit word length and an address limit of 32K words. Instructions to access the main memory are either 24 bits long or 16 bits long if the address is within the registers. In fact instructions are either 8, 16 or 24 bits long, so that as many as six instructions can be in one word.

The main registers are stack organised. and the arithmetic instructions are zero address which are only 8 bits long. The instruction buffers on the machine consist of two 48-bit words, which are not reloaded for each instruction of a straight section of code. The zero address add instruction is only one microsecond whereas the core cycle time is six microseconds. Thus main storage access and jumps are relatively much more expensive on this machine. To allow processor logic overlap with the main memory, the register stack instructions are executed some way behind the store and fetch instructions. This prohibits the processing of stack overflow interrupts until the state of the stack has been corrupted. Because of this, care must be taken not to overfill the stack (16 words). However, it must be well utilised on KDF9 otherwise the extra use of long instructions and main memory will seriously degrade performance. The size of generated code must also be watched with care because the limit on an instruction address is 8K words or typically 20K instructions. This limit is acceptable with hand-coding, but can easily be exceeded with a compiler which generates lengthy code (like Atlas).

Apart from the main register stack, there is an additional stack of 16-bit words for subroutine links. Use of this is not critical as far as performance is concerned, which is fortunate as in ALGOL the depth restriction would pose awkward problems. The address modification registers are also somewhat unorthodox. Each of the 15 registers is divided into three 16-bit parts. Loop code instructions use one part for counting (down to zero), while another part is used for address modification. The third part is an increment which is added to the address modifier if a fetch or store instruction states that this is required. These features allow rapid scanning of matrix elements by rows or by columns. The address registers can only be loaded from the stack and not directly from main memory. In short, KDF9 contains several facilities (stack, subroutine stack, auto-increment) which are very difficult to exploit in any high-level language.

Three ALGOL compilers are available for KDF9, the Whetstone interpretive system, the Kidsgrove compiler and the Egdon compiler. In this section only the Kidsgrove compiler is considered. This compiler was designed to work in conjunction with the Whetstone system (with fast compilation for program development), whereas the major aim with the Kidsgrove compiler was the production of efficient machine code. This design was very ambitious in that the intention was to exploit most of the significant features of the machine in a comparable manner to an assembly code programmer. This was to be achieved by an optimising phase in the compiler. The basic system produces "safe" code but with poor array accessing.

The optimiser improves this substantially by achieving most array element accesses within **for** loops in one instruction. The penalty for attempting this is that a very elaborate analysis of the program is necessary, the majority of which is performed even if the optimiser is not used. Unfortunately the optimiser not infrequently generates invalid code or even worse, generates code which gives different answers. In practice, this means that at NPL we advise programmers not to use optimiser unless the speed improvement is worthwhile (can vary from 200% to -20%) and they can check the compiled program effectively. For this reason, only unoptimised code is considered here.

The compiler did not do particularly well on the compiler tests. Of the 13 tests, ten were completely satisfactory and three failed at compilation time. These failures were not detected in the first pass of the compiler, so positional information is not given. Moreover, the actual error messages are not phrased in a manner which is likely to be intelligible to the average programmer. Fortunately, provided adequate testing is done with the Whetstone compiler, such error messages with the Kidsgrove compiler can often be avoided. The main difficulty is that the slowness of execution in Whetstone (20 times slower than Kidsgrove) is such that testing often has to be done with the Kidsgrove system. Many compilation failures result from the compiler checking the parameters to formal procedures. The actual procedures must therefore have the same specification (although this is not always checked properly), and it must be possible for the compiler to deduce this. Sometimes this can be overcome by inserting a dummy call (this is also necessary on Atlas). With one different test program, the Kidsgrove compiler actually got into a loop attempting to check formal procedure parameters because the chain of formal procedures was cyclic. This formal procedure parameter problem accounted for two failures, the remaining one being due to a complex switch. On the other hand, run-time errors were reasonably explicit except where code has been overwritten. A major defect of the compiler is that there is no facility for subscript bound checking, making errors of this type very hard to trace.

The overall structure of the compiled code reflects the requirement of the optimising phase since the majority of the compiler is independent of the optimiser. Hence only 5 out of the 15 address registers are used ordinarily leaving 10 for the optimiser. Environment control is in fact handled by only two or occasionally three index registers. This is achieved firstly by assigning storage at procedure level, and secondly by using fixed locations for global first order working store. One index register points to the current procedure environment while a second gives the top of the stack. Stacking a variable can be done in one instruction by use of the auto-increment as the increment is set to 1. Expressions which do not involve a function call (or name call) are calculated entirely with the register stack. Within these expressions common subexpressions not involving a type change are calculated only once. Since subscript optimisation is separate from this, subscript expressions are not included in this common subexpression optimisation. Common subexpressions usually result in its value being copied within the register stack and later the stack contents are manipulated so that the operators can evaluate the expression correctly.

Illustrating instruction sequences is rather awkward because of the zero address coding. So the convention is adopted that ACC is the top of the register stack, assignment to ACC causes an element to be put on the register stack and conversely with the use of ACC. STP is used to denote the register containing the address of the top of the ALGOL stack.

The assignment statement $e2[1, 1]$:= 1 generates

```
ACC:= 1;
STP[0] := ACC, STP := STP + 1;
ACC := 1;
STP[0] := ACC,STP := STP+ 1;
ACC := e2;
CALL 2 dimensional array subroutine;
ACC := 1;
if overflow then goto overflow fail;
REV;
M3 := ACC;
M3[0] := ACC;
```

Very poor code is generated because explicit stacking is performed for each subscript. In general the register stack cannot be used because it may overflow, and use of other extra workspace is excluded because this is allocated independently of whether optimisation is done or not. Overflow is not an interrupt condition on KDF9, so explicit tests must be made occasionally (see 1.1 and 6.3). Note that while the right hand side is being calculated, the register stack contains the address of the left-hand side—which is a piece of optimisation that quite a few compilers omit. The instruction REV interchanges the order of the top two elements of the register stack so that the address can be placed in M3.

The second statement illustrates the common subexpression optimisation. The statement

$$random := ri := ri - ri \div 12601 \times 12601$$

produces

```
ACC := ri;
ACC := 12601;
REV;
DUP;
CAB;
CALL integer divide;
ACC := 12601;
×;
DUP;
$ri$ := ACC;
**if** *overflow* **then goto** *overflow fail*;
CURRENTEN[*random*] := ACC;
```

The integer ri is global and is therefore accessed as a fixed variable, whereas *random*, being local to the current environment is accessed via CURRENTEN. ri is not loaded twice into the accumulator, but additional register stack manipulations are necessary to achieve this. Note that the REV could have been omitted if the loads had been reversed. The instruction CAB does a cyclic permutation of the top three words on the stack putting the third one on top. The subroutine integer divide is quite short, but is necessary because of the sign problem (see 1.1.4.1). The extra instruction after the multiply is a normalisation order but it is written as a dummy since it is of no interest (apart from noting that it is necessary). In this example, the common subexpression optimisation reduces the code size by about 4% and increases the speed by rather less (owing to the subroutine call). The optimisation appears to take place in between 5 and 10% of expressions and so contributes very little to the processing speed. The fact that it does not effect the simple statements used in 2.2 is therefore not significant.

Function and procedure calls are handled on KDF9 in a virtually identical way to that of Atlas. Each parameter is evaluated and the corresponding information stacked in the position required by the calling procedure. This is illustrated by the next example

$$y := a - i \times x \uparrow 2$$

which gives

```
STP := STP + 1;
STP := STP + 1;
ACC := 1;
$a$;
ACC := CURRENTEN [$i$];
ACC := CURRENTEN [$x$];
ACC := 2;
CALL ↑;
REV;
CALL float;
REV;
×;
**if** *overflow* **then goto** *overflow fail*;
CURRENTEN [$y$] := ACC;
```

Note that two unnecessary REV instructions are generated because i is not floated when first loaded into the accumulator. Also the incrementing of STP could have been incorporated into the procedure a itself since it is parameterless. The 1 placed in the accumulator before calling a is in fact the identification of the calling environment. This could be avoided if it were placed in the link data.

The subroutine used on procedure call is itself called by the code corresponding to the procedure. In this way, the procedure calling code is fairly compact—at least with value parameters. The procedure entry subroutine sets up the link data and increments the stack pointer to the end of first order working store for the new environment. Then a check is made to see if any address registers need to be preserved owing to their use in optimisation. This is then followed by a check that adequate stack space exists. Dynamic preservation of address registers is difficult on KDF9 since no instructions exist for accessing the nth register (n is data). Consequently, optimisation can quite often lead to worse code. Consider

for $i := 1$ **step** 1 **until** n **do** *output*(30, $a[i]$)

An address register will be allocated to access $a[i]$ saving about three instructions whereas about a 20 instruction overhead will be involved in preserving and restoring this round the procedure call. Moreover the compiler does very elaborate checks to ensure that procedures do not overwrite control variables which would invalidate this optimisation. Such register resetting is the main reason why optimised programs occasionally run slower than unoptimised ones.

An interesting feature of the Kidsgrove compiler is the classification made of procedures, although this has the annoying effect of restricting the number of procedures in a program to 96. As far as unoptimised code is concerned, a procedure is classified into one of three types, simple, non-recursive or recursive. Simple procedures are very simple indeed in that they access no non-locals, have no name or array parameters, and call no other procedure. Even with these restrictions it does (fortunately) include the majority of input/output and library procedures as well as all the standard functions. The call of simple procedures is performed by open code using the stack pointer as the current environment pointer within the procedure. The dummy procedures used in the timing comparison are simple and so give an optimistic figure for the compiler. The test for non-recursion is used to help with the access to variables in nested procedures. If the outer procedure is non-recursive then its environment pointer is placed in a fixed location allowing rapid access by the inner procedure. If the procedure is recursive, then a subroutine is used in the inner procedure to access outer procedure variables. This subroutine goes down the dynamic chain searching for link data containing the correct identification (see above). This is very slow, but in 200 programs scanned, no call of this subroutine was found (see 2.3.4.2). Hence the main disadvantage of this technique seems to be the necessity to check recursion, but this is required by the optimiser anyway.

The final example of compiled code is $s := s + f(5)$ which gives

```
ACC := CURRENTEN[s];
CURRENTEN[TEMP] := ACC;
STP := STP + 1; STP := STP + 1;
ACC := 5;
STP[0] := ACC, STP := STP + 1;
ACC := 1;
CALL f;
ACC := CURRENTEN[TEMP];
REV;
```

```
+;
if overflow then goto overflow fail;
CURRENTEN[s] := ACC;
```

Note that a location in the current environment (TEMP) is used to preserve the partial expression—it cannot be kept in the register stack because of the risk of overflow. The unnecessary REV is generated because the compiler does not distinguish commutative from non-commutative operators.

Code is generated twice in two contexts. With call-by-name, two thunks are generated, one giving the value, the other the address (or error). This removes the necessity for a dynamic check on assignment to a name parameter, but the volume of code generated is excessive. The other case is with the expression in the step element of a for loop. In both cases, the complexity of the context is restricted by the compiler, presumably owing to a buffer size.

The exceptional values for KDF9 on the tests are as follows. Firstly Knuth's "Man or boy" is executed quite fast. This is surprising in some ways since the "up level" addressing subroutine is used. The compiler does, however, generate open code to set-up name parameters, and no dynamic check is needed on assignment. The ratio is 0.40. On the other hand the statement x := 1 and x := l comes out badly with ratios of 4.0 and 3.1 owing to a slow subroutine for the conversion at run-time. Similar reasoning applies to k := 1.0 with a ratio of 2.8. The dummy block **begin real** a; **end** comes out well at 0.23 because of procedure level storage. It does generate some code, because no distinction is made between blocks and array blocks—the code is in fact that necessary on block exit to release array storage. Two dimensional array access is just exceptional at 2.1 (see above), and three dimensional access more so at 2.7 because a general subroutine is used to cover any number of dimensions. x := $abs(y)$ comes out poorly as the general procedure mechanism is used (ratio is 2.5) although this is partly offset by the fact that the procedure is "simple". Needless to mention, the dummy procedure calls come out well through the "simple" procedure classification with ratios of about 0.32.

With the number of instructions executed exactly the same pattern emerges as with the execution time with one exception. The statement x := $ln(y)$ executes 3 times as many instructions as expected. This is due to use of a large number of fast stack register instructions which is characteristic of hand-coding. No significant variations occur in terms of the code size in instructions or code size in bits—so the overall ratios given in 9.2 characterise the compiler fairly accurately.

In order to assess the effect of optimiser, just one program was compiled with it—the GAMM test. As already noted, this program can be optimised very extensively, so it provides a yardstick for measuring the limits of the process. For the statement

for $i := 1$ **step** 1 **until** 30 **do** $r[i] := a[i] + b[i]$;)

the generated code was essentially that which would be produced by an ordinary assembly code programmer—except that a very specialised coding trick the "short" loop is not used. To achieve this, an index register is allocated to each of the arrays r, a and b. Optimal control code is produced by the auto-increment facility, and since the loop is always executed once, the test is done only at the end of the loop. The rest of the machine code remains unaltered, so for instance in a similar loop containing **if** $j > c[i]$ **then** the array variable is loaded in one instruction but the code to standardise the Boolean is produced. The nett effect on the GAMM figure is to improve it by a factor of about 4 which is indeed a remarkable achievement. Optimiser has virtually no effect on increasing the use of the register stack, as this requires leaving some values in the stack in between statements. This is difficult with a language having a complex statement structure. Optimisation is considered again in 10.3.

To summarise, the compiler has been used for a number of years at NPL as the main high-level language processor. Undoubtedly better compilers now exist, but it has served its purpose reasonably well. The main advantages are:

1. Large subset of ALGOL 60
2. Compatibility with Whetstone ALGOL
3. Optimiser can give substantial gains.

The major disadvantages are:

1. Lack of subscript checking option
2. Restriction of less than 96 procedures
3. Slow compiling speed (somewhat offset by Whetstone). Minimum program takes about 30 seconds, and one of 12K machine orders takes 7 minutes
4. Lack of local optimisation (see 10.3)
5. Subsequent pass errors are not informative enough.

It must be remembered that the KDF9 contains a number of hardware features which are difficult to exploit effectively in a compiler for a high level language.

9.5 B5500

This machine is probably the best known of the computers with a non-conventional architecture. Although it is a stack machine like KDF9 they have very few similarities. The stack automatically overflows into core-store, and, more significantly, the stack is used in the main-store addressing mechanism to produce very compact code. The word length is the same as for the last two machines—48 bits, but there are four instructions to a word. Two flag bits determine the instruction type—a load, load address, set literal or operator. The other ten bits give the address, literal value or operator code as appropriate. Every word has a flag bit which is zero for user defined variables, but if this bit is set, then any access to the word causes an action depending on the remaining bits in the word. The action could be a procedure call, a level of subscripting, the loading of the flagged word into the stack, or an interrupt.

The core-store is allocated dynamically to the user program under the control of the supervisor (MCP) which is critical for the correct functioning of the machine. Procedures and arrays need not stay in the core-store for the duration of program execution, since the appropriate control word can merely have its "presence bit" flag set so that access will cause an interrupt. This allows the segmentation system of virtual memory to be implemented as opposed to the paging logic of Atlas. Each program contains two main areas for direct addressing. The Program Reference Table (PRT) which corresponds to the global variables in an ALGOL program, and the stack which contains non-global variables as well as being the overflow from the stack registers. Only two registers are used for the top of the stack, so from the point of view of data access the B5500 does no better than a conventional one address machine. However, the reduced instruction size means that instruction word fetches are much less frequent.

The arithmetic operations are the same for integers or reals (which are distinguished by the exponent). The store operations do distinguish these two types, so that rounding can be performed on integer assignment. Overflow is an interrupt condition with real or integer working and underflow can occur with real arithmetic. Thus there is little difficulty in the implementation of arithmetic expressions—there is even a special operator to deal with name parameter assignment which ensures correct rounding (see 1.10.3).

The ALGOL compiler is itself written in ALGOL 60—something which would barely be possible with any of the other systems considered in this chapter. As there is no true assembly language for the B5500, the design of the ALGOL system has been aimed at fulfilling the requirement to exploit every feature of the hardware. This implies that the language is not true ALGOL 60 but contains some significant omissions owing to the hardware limitations, and some much more significant additions. The two major restrictions are that access to outer procedure variables is not allowed (as there is only one stack address pointer), and that every bound pair is restricted to less than 1023 words. In practice, the first restriction can be overcome by declaring the outer procedure level variables as **own** (which is transformed by the compiler into global). The second limitation is more severe since if 2000 words of store are needed a two dimensional array must be used which increases the accessing overheads. This apparent deficiency is used with advantage by the ALGOL compiler which declares its namelist space as *namelist* [0:32, 0:256]. One might imagine that 8K words were required for each compilation, but this is not true as the declaration merely sets up 32 descriptors pointing to blocks of 256 words which are “not present”. Hence, as the compilation proceeds an increasing large number of the 256 word blocks are allocated to core or disk. The major additions in the language and machine are part-word operations, character operations and input-output handling “intrinsics” (i.e. built-in procedures).

The restriction on array length prohibited the execution of the sorting test. In order to give more data for the analysis, program figures were estimated based upon the compiled code. One could argue that the program should be recoded to perform the same function using two dimensional arrays.

The ALGOL compiler is one-pass, using the method of forward declarations to overcome the problem of use before declaration (see 7.10). The compiler did not do very well with the compiler tests. Of the 13 tests, two failed, three required modification, and three minor modifications. Knuth’s “Man or boy?” failed for two reasons. Firstly it used nested procedures with access to the outer procedure (which is recursive, so **own** cannot be used). This could probably be overcome but it could not execute as there is a MCP limit of 1024 words on the stack which would be exceeded.[1] The second test to fail used a label parameter to a function designator. This is not allowed in B5500 ALGOL because a jump out of a function could get the stack out of step. In fact, the compiler cannot check for this sort of action completely so there are circumstances when the stack will be corrupted. The minor modifications were caused by the necessity for some reordering of declarations (or inserting forward declarations) and changing parameter comments (to) “abc:” ( ). The tests which required additional modifications were due to the nested procedure addressing restriction, the absence of strings and the use of a B5500 reserved word as an identifier. These difficulties illus-

[1]“Man or boy”works with Compatible ALGOL; a true compiler designed for timesharing and compatible with the B6700.

trate that the conversion of programs to B5500 ALGOL is not necessarily easy, and certainly not possible merely by a change in the hardware representation.

The author has been informed that the compiler makes no check on the parameters to formal procedures. It is difficult to see how a one-pass compiler could make any reasonable check especially as no language extension is provided to specify the parameters (see 7.2). This is surprising as the dimension of formal array parameters must be specified, via a language extension. This extension can also provide the values of the lower bound of each array subscript, allowing the generation of more efficient code. The syntax checking performed by the compiler is very poor—although this is specified in the manual as an "extension". The **for-else** ambiguity of the unrevised Report is allowed (see 8.6), as are a few other ambiguities, but this is somewhat off-set by a manual which states very accurately (the author believes) the language accepted by the compiler.

Very few constructs result in subroutine calls, because the hardware provides most of the necessary facilities. The subroutines that are used are called "intrinsics" and are ALGOL procedures (written in ESPOL). One example is the "goto solver". Labels must be declared, so that local jumps can always be detected in one pass and are translated as straight jumps. Otherwise, the goto solver is used. This must cut the stack back, which necessitates a search for array descriptors in the stack so that array space can be returned (which is stored off the stack). Intrinsics are the only way code not compiled with the source can be used—hence the library procedures all use this mechanism. The MCP loads programs with the intrinsic descriptors set as "not present" but pointing to the intrinsic file to allow dynamic linking. The other technique to aid program execution is the call of supervisor functions—which must be used for array declaration and deletion since the MCP is responsible for dynamic storage allocation. This accounts for the very long times needed to declare arrays on the B5500, although dynamic space is not actually required except for dimensions greater than one for the descriptors. B5500 machine code is not easy to follow because the fetch and fetch address syllables depend upon the word accessed.

Both the advantages and disadvantages of the array accessing method are illustrated by the code produced for

$$e2[1,1] := 1$$

which produces

```
ACC := 1;
ACC := 1;
ACC := ADD(e2);
LOAD ADDRESS;
ACC := 1;
ACC := 1;
−;
CALL DESCRIPTOR;
ACC := 1;
REV;
STORE INTEGER;
```

The array $e2$, being written in the usual manner has 1 as its lower bound. Since the descriptor method only handles a lower bound of zero, open code is produced to reduce the subscript values. This is not optimised as can be seen from above. Not unnaturally most B5500 ALGOL programs have arrays with a lower bound of zero. The descriptor mechanism is handled explicitly at every level. After calculating the reduced first subscript, the descriptor for $e2$ is loaded and an indirect fetch executed to load the descriptor of the first row. The second reduced subscript is then calculated, after which the descriptor of $e2[1, 1]$ is produced. The store operation requires the address on the top of the stack so a REV is necessary, which is avoided with simple variables by loading the address after the calculation of the expression. Even with the short instruction length, this code is barely better than a conventional machine with the important exception that subscripts are necessarily checked.

The use of the stack is better illustrated with the statement

$$random := ri := ri - ri \div 12601 \times 12601$$

which gives

```
ACC := $ri$;
ACC := $ri$;
ACC := 12601;
$\div$;
ACC := 12601;
$\times$;
ACC := ADD ($ri$);
STORE NON DESTRUCTIVE INTEGER;
ACC := ADD (*random*);
STORE INTEGER;
```

The common subexpression optimisation that is performed on KDF9 is not in general possible on the B5500 because it does not have methods of addressing from the stack front downwards nor does it have the large repertoire of manipulation instructions (only two elements are in registers, so there is little point). However in this case it could have generated DUPLICATE instead of reloading ri. In this particular sequence, eight data accesses are required, just under three instruction accesses, but also, the stack overflows once and must be restored (i.e. two hidden core accesses).

The third example illustrates the non-conventional architecture more clearly.

$$y := a - i \times x \uparrow 2$$

gives

```
MARK STACK;
ACC := $a$;
ACC := $i$;
ACC := $x$;
DUPLICATE;
$\times$;
$\times$;
$-$;
ACC := ADD($y$);
STORE;
```

The MARK STACK operation is the equivalent of making space for link data on a conventional machine. This sets up part of the link data, the remaining part being formed by the procedure call itself—which looks like an ordinary load operation. Being a function, the result of a is left in the stack in just the manner required for the expression evaluation. The exponentiate is performed by repeated multiplication in the stack—in fact any fixed power up to 1023 is done this way using the "bit pattern" technique (see Knuth, 1969, p. 398). The compact and fast procedure call mechanism is certainly one of the outstanding characteristics of this machine.

The statement $s := s+f(5)$ produces less than two words of machine-code, namely

```
ACC := $s$;
MARK STACK;
ACC := 5;
ACC := $f$;
+;
ACC := ADD ($s$);
STORE;
```

Note that although no special action need be taken to preserve the value of s before the function call, a store and reload into the stack is performed implicitly by the hardware. If the body of a procedure is a compound statement, then only three instructions are needed to call a procedure—MARK STACK, the load of the procedure descriptor and finally the return instruction. So no additional code appears in the procedure itself.

The anomalous values of the characteristics measured are as follows. With the execution time, six values are anomalously high and eight values are low. The large spread is caused by the substantial underlying differences between this machine and its conventional counterparts. The high values are from the array declarations (ratios of 5.1, 3.9, 10 and 13 from **begin array** $a[1:1]$; **end**, **begin array** $a[1:500]$; **end**, **begin array** $a[1:1, 1:1]$; **end** and **begin array** $a[1:1, 1:1, 1:1]$; **end** respectively) and from the standard functions $x := y \uparrow z$ (ratio 6), $x := ln(y)$ (ratio 2.5). The long time for the array declarations is caused by the MCP call which must dynamically claim core-store. It is interesting to note that this does not degrade the performance of the machine significantly because the frequency of array declarations is not high enough. The reason for the slow times for the standard functions is not known. The good values come from the dummy block (ratio 0.31), the switch (ratio 0.41), *abs* and *entier* (ratios 0.36, 0.35) and the procedure calls (ratios 0.24). The good time for the block entry is caused by procedure level storage, although code is produced to segment the instructions—each block is a segment. Simple switches are optimised by the compiler into a jump table, whereas *abs* and *entier* generate open code. In terms of instructions executed the only other anomalous one is $l := y$ it does not take significantly shorter to execute than conventional machines but it has fewer instructions because the type conversion is implicit on storing the value.

With the measure of code size in instructions very little variation occurs. However the **goto** takes 2.2 times the expected number of instructions, because of the necessity to place dummy instructions in the code so that the label can be at the beginning of a word. This could have been avoided with a multi-pass compiler which would have noted that a short branch instruction could have been used, not requiring the destination to be at the start of a word. Needless to mention, the procedure calling code is also exceptional producing about half the number of expected instructions. The code size in bits follows the measure using instructions, so the short instruction length is largely absorbed into the extremely good overall figures for code size given in 9.2.

To summarise the results, the non-conventional architecture of this machine gains mainly by producing extremely compact code—half the size of its conventional counterparts. This is achieved mainly by a very short address length. There are very substantial deviations from true ALGOL 60 in the language offered—this could be troublesome in program conversion but is unlikely to be significant in writing programs for the B5500. Writing systems programs in ALGOL 60 is made possible by the fast procedure entry mechanism and the efficient array accessing hardware—provided arrays are declared with a lower bound of zero. The restriction of less than 1024 words in an array segment could be very awkward indeed in some applications. It is interesting to note that this restriction has been removed from the B6500—in fact hardware even exists to page large one dimensional arrays so that the ALGOL compiler technique of store economy for the namelist can be implemented more directly. Another restriction to be removed is that of addressing outer procedure variables.

9.6 1108

The Univac 1108 computer is a member of a series unlike the previous three machines. The series does not span the range of IBM 360/370. The most well-known member of the 1100 series is the 1108, and until the recent announcement of the 1110 it was also the most powerful. The computer is word oriented, with a 36-bit word length. There are a very wide variety of machine-code operations for double and single length floating point as well as fixed and part-word operations. The computer is a multi-register machine, it being possible to use them as index registers or floating point accumulators. In fact there are a total of about 80 registers, but some of these have specialised uses. The processor is also capable of handling some single instructions which loop within themselves. A MOVE and some search instructions are of this form.

The compiler being considered is one written in Trondheim (Norway) to replace the original Univac compiler. It was written in collaboration with another team implementing SIMULA (Dahl, 1966). The specification includes a number of significant additions, for instance, double length reals, complex, string and character handling. The implementation of these additions does not seem to have any detrimental effect on the pure ALGOL 60 part, although the requirements of SIMULA would have such an effect.

The compiler did reasonably well with the compiler tests. Ten of the thirteen were satisfactory whilst only minor objections could be made to the other three. In two of these, a function involved a very severe side-effect, so as expression evaluation is not strictly left to right this gave a different answer. The remaining test failed owing to a type conversion on a value array. A further minor error was found in that parameter comments are not allowed. One error was found in the compiler due to a piece of optimisation, whereby constant expressions are calculated at compile time, but with the expression $1 \times 1 \uparrow 1$, the wrong value was calculated.

The general strategy of the compiled code is to use several index registers for a display, allocating variables at block level. The specification of the compiler demands that variables are assigned zero (or its binary equivalent) on declaration. This naturally means an overhead on block entry although this should be small compared with the processing on the variables. Assigning storage at procedure level would therefore not be straightforward.

This compiler and the two subsequent do not deal with procedure parameters in a pre-call sequence, but the parameters are checked and evaluated as necessary after procedure entry. The run-time routine which evaluates parameters and the other similar routines were not analysed so it is not possible to state precisely the reason behind some performance troublespots.

The first illustrative statement gives a good example of the dope vector method of array access, $e2[1,1]$:= 1 generates

```
A3 := 1;
A3 := A3 × B2[e2'];
A3 :=A3+1;
A3 := A3 + B2[e2];
A3[0] := R15;
```

A3 is a register used to calculate the address of the array element, B2 contains the pointer for the block containing the array. Since the array is global, this pointer could be dispensed with, allowing its use for further optimisation. The register R15 always contains the integer 1 to assist in stacking, etc., and this is used in the expression code, but not for the subscripts. $e2$' is, of course, the number of elements to the first subscript, while $e2$ is the address of the element $e2$[0,0].

The assignment statement $random := ri := ri - ri \div 12601 \times 12601$ illustrates the effect of evaluation which is not strictly left to right. It generates

```
A6 := B2[ri];
A6 := A6 **shift** 36;
A6 := A6 + 12601;
A6 := A6 × 12601;
A6 := A6 − B2[ri];
A6 := −A6;
B2[ri]:= A6;
B4[random] := A6;
```

Only one instruction is saved by the reordering because of the extra order to negate the accumulator. A6 is used for expressions, which does not conflict with the use of A3 for address calculation and simple subscripts.

The next example involves a function call, the mechanism of which requires some explanation. The subroutine call instruction on the 1108 places the return address in an index register, so parameters can conveniently be placed after the CALL instruction. Every procedure has a precall sequence generated separately from the actual code. This sequence does another CALL using a different index register to the procedure call subroutine and as a result two index registers can conveniently be used to access details of the actual and formal parameters. Corresponding to each formal parameter there is a word whose fields give: the type (**integer**, **real**, **Boolean**, **label** or **switch**, etc.), the form of specification (**procedure**, **array**, **simple**, **label**, **switch**), and if the parameter is by value, together with the position of the parameter in the procedure working store. Fixed information concerned with the procedure is also contained in the prelude (type, extent of working store, block depth). The particular representation of the words giving details of the formal and actual parameters have presumably been designed so that simple masking operations can deal with the straightforward cases in a few instructions. The times for the simple statements show that value parameters are handled rapidly although the set-up time for any procedure is rather long. The example

$$y := a - i \times x \uparrow 2$$

gives

```
A0 := ’procedure entry address’;
CALL ’a prelude’;
’2’
CALL ’proc enter’;
’2’
A4 := 2.0;
A2 := B3[$x$];
A6 := A0;
CALL ’exp sub’;
A7 := A0;
; (A0,A1) := $i$;
A1 := A1 $\times$ A7;
A6 := A1+A6;
B3[$y$] := A6;
```

The actual address of the code of procedure a is found from the prelude sequence. Two subroutines appear to be called in the procedure entry process, the first one whose address is assigned to A0 presumably sets up the link data and the parameter evaluation as necessary, the actual entry being accomplished by the second CALL. The result of the function is in A0. The use of a subroutine for exponentiate is not very good here as two register transfer instructions are necessary because of the fixed specification of the routine (A0 := A2 $\uparrow$ A4). Note that a real value 2.0 is used for the exponent which indicates that only one routine is used for $\uparrow$ so that error conditions are probably not handled correctly. Floating of i seems to be reasonably straightforward only requiring two instructions.

The final example is $s := s + f(5)$ which gives

```
A0 := ’procedure entry address’;
CALL ‘$f$prelude’;
’I’
A7 := 5;
A3[2] := A7;
CALL ’proc enter’;
’I’
A0 := A0 + B3[$s$];
B3[$s$] := A0;
```

Note that the addition of s is performed after the function call.

Apart from the precall sequence being generated separately, thunks are not placed in line, avoiding additional jump instructions which would otherwise be necessary.

The anomalous values for the four characteristics measured for the tests are as follows. Firstly, the execution time. The statement k := 1.0 is fast (ratio 0.38) owing to compile-time conversion and optimisation using the fact that register R15 always contains 1; that is, the one instruction B2[y] := R15 is generated. The statement x := $y \uparrow z$ is fast (ratio 0.21) due to exceptional coding for $z = 1.0$— this can be seen from the coding given above for $\uparrow$ 2 which is of the same form. Entry to a dummy block is slow (ratio 4.1) owing to block level allocation of storage and zeroising first order working store. The declaration of an array of 500 elements is 8 times as long as expected, also because of the initialisation. The **goto** is fast (ratio 0.23) as only a single instruction is generated. The statement x := $abs(y)$ is optimised very well indeed producing only two instructions— A6 := *abs*(B2[y]); B2[x] := A6;. The switch is slow, but this is accounted for mainly by the block—the switch time itself is not really exceptional. All the standard functions are relatively fast and $x := exp(y)$ is only just anomalous (ratio 0.43). This is apparently through the good use that hand coding can make of the many fast registers on the machine. The weakest point is undoubtedly the procedure entry times which have a ratio varying from 5.5 to 4.4. As the overhead per parameter is only of the order of six instructions it is clear that the major problem is the procedure set-up. There does not seem to be any good reason for this—certainly with block entry allocation of storage rather more registers must be reloaded (4 or 5) but this does not account for the discrepancy. The only differences between time and instructions executed is with $l := y$ with a ratio of 2.4. A run-time subroutine does the type conversion again making good use of the many registers so that the time is not excessive.

As usual, there is much less spread in the code size—measured either in bits or instructions. Only a parameterless procedure call is exceptional with a ratio of 2.4 in the number of instructions. The reason for this can be seen from above with the double call sequence—which is unnecessary when there are no parameters.

The NU 1108 compiler therefore achieves reasonable compact code by an intelligent use of quite a few of the registers, but has the significant disadvantage of being slow on block and procedure entry.

9.7 CDC 6600

The CDC 6000 series has two major members the 6400 and the 6600. The 6600 is the one considered here, although most comments apply to both machines since they have the same order-code. The major difference is in speed, the 6600 achieving this largely by having several accumulators which are semi-autonomous. For many years the 6600 was the fastest commonly available scientific computer, and in its design some simplicity in programming has been sacrificed to achieve this speed. The processor contains 24 arithmetic registers and 10 autonomous functional units. In order to exploit the speed of the machine, neighbouring instructions must use different sets of the 24 registers so that processing can be done in parallel by the functional units. This requirement is made more difficult by the fact that the variable length instructions cannot overlap word boundaries. The word length is 60 bits, and register instructions are 15 bits whereas those for memory reference are 30 bits long. This forces the compiler to generate dummy instructions—as much as 20% in some cases. The optimising FORTRAN compiler avoids this, but the PASCAL compiler of Wirth has the same difficulty (Wirth, 1971b). Also to achieve greater parallelism the registers are divided into store registers (mainly X6, X7) and fetch registers (mainly XO to X5). The load and store operations are the same, but reference to a fetch register causes a load and conversely for a store. Thus three instructions are necessary for $x := y$, y must be loaded into a fetch register, the fetch register value placed in a store register, then the store to x. The register division is thus

X1 to X5	operand fetch registers
X6,X7	operand result registers
A1 to A5	operand address registers (address of values XI to X5 on fetch).
A6,A7	result address registers
B1 to B7	address modification registers (B0 = 0).

In many ways the instruction set is more like that of a microprogrammed machine since store and fetch are separate from any computation. This has the advantage that good hand coding can produce extremely fast instruction execution—up to about one instruction every 100 ns. The disadvantage is that it is very difficult for a compiler to exploit the parallelism properly—although reordering of instructions is often possible to achieve greater overlap and better packing in the instruction words.

Another substantial difficulty is that there is no fixed point multiply and divide instructions (a multiply instruction has recently been added). Presumably for this reason the ALGOL compiler represents integers as floating point numbers using the rounding instructions. Although like Atlas, unnormalised arithmetic is available this is not convenient for subscript calculation. The difficulty with this choice is that the performance depends critically on the relative frequencies of subscripting and integer multiply and divide. Unfortunately the Whetstone figures given in 2.3.1.2 do not give the type of the operands and hence the required information is lacking.

The general strategy of the compiled code is rather disappointing. Firstly storage is allocated at block level. With only eight index registers this leaves too few registers for other purposes—in particular for the important aspect of subscript evaluation. Secondly, the generated code is divided into segments for no apparent reason. The segments do not correspond to any natural division in the program. Inter-segment jumps are via a control routine, so it could happen that an inner loop straddles two segments causing the program to run at half speed. The segmentation appears to be due to the compiler construction itself, being based upon the GEIR compiler which had a small internal store. It also seems that ALGOL statement code always starts on a word boundary, although there appears to be no reason for this. It is only labelled statements or statements to which an implicit jump is made that need start on a word boundary.

The compiler did very well on the compiler tests. All 13 tests ran successfully without modification, the first compiler apart from Whetstone to achieve this.

The first example $e2[1, 1]$:= 1 illustrates some of the difficulties with the machine. With multi-dimensional array access, dope vector multiplication is certainly not possible. Code-words could be used, but as the store-speed is long compared with the logic time this is not very good either. So the compiler attempts optimisation which certainly succeeds to an extent, with the constant subscripts, as the code for $e2[1, 1]$:= 1 illustrates:-

```
X1 := B4[e2];
X2 := B4[e2'];
X7 := X1 + X2;
X3 := 1;
X6 := X3;
X7[0] := X6;
;;;
```

The final three dummy statements represent dummy instructions to align the next statement at a word boundary. Note that in the order generated the instructions cannot overlap very much because the result of one operation is often used in the next. Some simple re-ordering would achieve greater parallelism, for instance

```
X1 := B4[e2];
X3 := 1;
X2 := B4[e2'];
```

```
X6 := X3;
X7 := X1 + X2;
X7[0] := X6;
```

This could reasonably be achieved by a post-processor on the code generation part of the compiler (as in 10.3).

The second example is the statement

$$random := ri := ri - ri \div 12601 \times 12601$$

which generates

```
X1 := 12601;
X2 := B2[$ri$];
X7 := X2/X1;
X3 := CONSTANT;
X6 := X7 + X3;
X0 := NORM (X6);
X4 := 12601;
X7 := X0 × X4;
X5 := B2[$ri$];
X6 := X5 − X7;
X7 := NORM (X6);
B2[$ri$] := X7;
B4[random] := X7;
```

The CONSTANT is a bit pattern to achieve the necessary rounding on the divide operation. This code does use the available registers reasonably well, so in order to make a significant improvement it would be necessary to detect that X4 and X5 need not be loaded because the values are already in X1 and X2. It would then be possible to reorder the instructions to achieve greater parallelism.

The procedure call mechanism is interesting because it differs radically from all the other compilers considered. Parameters are checked at execution time—a severe penalty. However much of the overheads are shortened if the actual parameter is a simple variable (or identifier). In this case, the precall sequence stacks the value address and type details. The procedure entry subroutine checks the type details against the specification using simple masking operations. Then the procedure is entered. The first part of the code of the procedure does a "name call" for each value parameter, and then stores the value within the procedure. The "name call" sequence checks dynamically whether the actual parameter was simple; if it was, then the value is already stored and no thunk is evaluated. This means that the overhead for value parameters when the actual parameter is simple is quite small, as can be seen from the times obtained for procedure calls in 2.2. If a thunk is required, then this is recorded in the type details, so that the code at the beginning of the procedure will call a subroutine to evaluate the thunk. One difficulty of this technique is that name call evaluation code is quite bulky and it appears once for every value parameter as well as in the essential cases. As the formal definition of the value mechanism is described in terms of evaluation and assignment within the procedure, this implementation could be described as the official method. The call of parameter less procedures are handled by the same routine, EXECEXP, as is used for name calls.

The statement $y := a - i \times x \uparrow 2$ generates

```
        X7 := LINECOUNT;
        B4[CALLED FROM LINE] := X7;
        X1 := B3[$a$];
        (X1,B7) := UNPACK(X1);
        X5 := X1[0];
        **if** B7 $>$ 0 **then goto** $l$;
        CALL EXECEXP;
$l$:      X1 := B3[$x$];
        X2 := B3[$x$];
        X7 := X1 $\times$ X2;
        X3 := B4[$i$];
        X6 := X3 $\times$ X7;
        X0 := X5 $-$ X6;
        X7 := NORM (X0);
        B3[$y$] := X7;
```

The line number of source text from which a is called is preserved in the environment for diagnostic purposes. This code is also generated on every tenth line anyway, so the total overhead is quite significant. As noted in 5.4 this should not be necessary as tables relating source text to machine address should be available on failure. The second interesting point is the name call sequence which allows for the possibility of not calling a. The optimisation of $\uparrow$ 2, although good is somewhat surprising. One would have expected only one load of x with the product calculated in the registers (X7 : = X1 $\times$ X1), since this is quicker and will work for the square of a general expression. Note also that since integers and reals are stored internally in a similar manner, no type conversion code is required.

The last example $s := s + f(5)$ illustrates the simple case of the parameter mechanism. As with a simple variable, no thunk is generated. However, the volume of code is quite large.

```
X1 := B4[$s$];
X7 := X1;
B4[TEMP] := X7;
;;                  dummy instructions appear unnecessary here
X1 := LINE COUNT;
B7 := 3;
CALL SATISFY;
;;                  dummy instruction to return jump
X2 := RETURN ADD;
X6 := B4;
X7 := X6 + X2;
X1[2] := X7;
X3 := 5;
X7 := X3;
X1[0] := X7;        the parameter
X4 := CONSTANT;
X6 := X1 + X4;
X1[1]:=X6;
X1 := B4[$f$];
CALL PROC ENTRY;
```

```
                ;;                  dummy instructions to return jump
RETURN ADD:     X1 := B4[TEMP];
                X7 := X1 + X5;
                X6 := NORM (X7);
                B4[s] := X6;
```

The line count is loaded into X1, after which the routine SATISFY is called. This must preserve X1 as with the name call mechanism, but also reloads X1 with the address of the stack front. The word which is stacked first contains the return address and also the number of parameters. The long word length means that several items can be passed in one word. Next the actual constant 5 is stacked, followed by a word presumably denoting that the actual parameter is a constant. Finally a word containing the address of f is loaded into X1 and the procedure entry subroutine called. In both cases of subroutine calls it was necessary to insert two dummy instructions in order to make the return address start at a word boundary. The comparatively simple tasks of stacking parameters give little chance to overlap instructions effectively. To do this it would be necessary to stack the items in parallel rather than series. Note that it is not necessary to increment dynamically the stack front, since the stacking code cannot directly involve another procedure call.

This last example shows the compiled code in a very bad light. Undoubtedly, procedure calls are one of the weak points not only from the point of view of speed, but also size of code. To make a radical improvement to this it would be necessary to use procedure level addressing to release more B-registers and then check at compile-time parameters in all the straightforward cases. Even then, procedure calls are likely to be relatively slow because of the predominance of core accessing over logic times. Obviously the 6600 is much better when the logic times are large as with floating point work.

The anomalous figures obtained from the analysis of the four characteristics are as follows. With execution speed, four statements are fast and eight are slow. The fast ones are $x := y \uparrow 2$ and $x := y \uparrow 3$ (see above, ratio 0.4), $e2[1,1] := 1$ and $e3[1,1,1] := 1$ owing to the constant subscript optimisation (ratio 0.41 and 0.42). The slow ones are $x := y \uparrow z$ (ratio 2.7), **begin real** a; **end** (ratio 3.6) caused by block level storage, **begin goto** $abcd$; $abcd$: **end** (ratio 3.1) because general segment changing mechanism is used even for jumps internal to a segment, $x := entier(y)$ (ratio 2.2) due also to a subroutine call, and the procedure calls. The procedure call ratios vary from 4.3 for no parameters to 3.1 for three parameters indicating that with simple value parameters the main overhead is in the constant set-up time. In terms of instructions executed, the procedure *inarray* in quickersort is exceptional (ratio 2.1), but none of the other known values are anomalous. With code size in instructions the only exceptional value is $e3[1,1,1] := 1$ (ratio 0.45) owing to the constant subscript calculation. This value is also the only exceptional one in terms of the code size in bits.

The main characteristics of the CDC 6000 series compiler (version 2.0) should now be apparent. The level of language accepted is high but this is achieved partly by delaying some checking—especially parameter checking, to run-time. The difficulties presented by the machine architecture are not well handled by the compiler. Little use is made of the autonomous arithmetic units, and the percentage of dummy instructions generated is larger than necessary. Even with dynamic parameter checking, the relative speed of value parameter handling with simple actual parameters is quite good. The speed improvement in release 1.0 and 2.0 of the compiler is over 2, achieved mainly in the block entry and procedure calls. A further improvement of at least 50% would seem to be possible, but to exploit the machine fully would need a radical redesign of the compiler. Version 3.0 of the compiler has simple statement times similar to 2.0, but a 30% improvement has been made to the GAMM figure. An excellent critique of the 6600 instruction set has been given by Wirth (1972).

9.8 1900

The 1900 ALGOL compiler under consideration is XALT Mark 1A, the current (2/72) version of which is Mark 5. This certainly has some differences, but the basic structure is thought to be the same.

The 1900 series encompasses a wide range of actual machines from small (but not mini) to large (but not very large). In a similar way the software is intended to stretch over a wide range of processing speed, peripheral capacity and core-store sizes. The compiler in question will work on a machine with only 32K 24-bit words of store—substantially less than the other compilers considered. The instruction code is only upward compatible, that is, some of the larger, newer processors have instructions and modes of operations which are not available on the smaller machines. In general, this compiler does not make use of these additional instructions.

The computer is one-address, so with only 24 bits not all the core-store is directly addressable. In fact only 4K of “lower data” is directly addressable, other accesses being made via the three index registers. These three registers are part of the seven integer accumulators, the floating point accumulator (of 48 bits) being separate. The use of index registers for jump instructions is not necessary as jumps can be relative to the current instruction, in any case 15 bits are allowed for the addresses in jump instructions. In fact all the tests were compiled in direct branch mode since even the compiler tests were well within the limit. The short address length is of no consequence in ALGOL, but the existence of only three index registers is very awkward. The integer registers are also words 1 to 7 of the core-store (conceptually, not actually on the faster machines). In many respects, this machine is more similar to the illustrative machine than any of the other five considered here.

The compiler was produced around 1970 as a replacement. The compiler generates “semicompiled” or relocatable binary as on the 1108 and CDC 6600 computers. The control routines are split up into quite small units of code so that it is unlikely that much additional code will be loaded with a program except that necessary for its execution.

The compiler passed all the compiler tests except that the two involving a side-effect on a function call produced the wrong answer, as did a similar one, the reason for which is not known. As mentioned before, this cannot be regarded as a serious criticism of the compiler (indeed, it is more a criticism of the program).

The general strategy of the compiled code is that storage is allocated at block level, except that the parameters and locals to a procedure are regarded as one block. The current version of the compiler allocates storage at procedure level. The main program variables are allocated fixed locations in lower data (that is, are directly addressable). One index register (X1) is used for the current environment, and outer static levels are reached by chaining back from the current level. In view of the scarcity of index registers this seems a most reasonable method. The second index register is used for non-current environments, while the third is mainly used for subscripts. An important difference with this system is that a different amount of storage is used for integers (one word) and reals (two words).

The first illustrative statement shows the integer accumulators to good advantage.

$$e2[1,1] := 1$$

generates

```
X6 := 1;
X5 := 1;
(X5,X6) := X5 $\times$ CURRENT ENVIRONMENT [$e2$'] + X6;
X3 := X6;
;;     two shift instructions
X3 := X3 + CURRENT ENVIRONMENT[$e2$];
X6 := 1;
X3[0] := X6;
```

Note the dope-vector multiply instruction which is particularly convenient for high dimensional arrays. The two shift instructions are apparently necessary to ensure that overflow conditions are handled correctly, since the multiply instruction gives a double length answer. If the right hand side is anything more complex than a simple variable or constant, the address of the left-hand side is preserved.

The second example is $random := ri := ri - ri \div 12601 \times 12601$ which gives

```
      X6 := $ri$;
      X6 := X6 + 12601, X5 := remainder;
      **if** X6 $\geq$ 0 **then goto** $l$;
      **if** X5=0 **then goto** $l$;
      X6 := X6 + 1;
$l$:  X6 := X6 $\times$ 12601;
      ;;     two normalisation instructions
      X6 := X6 $-ri$;
      X6 := $-$X6;
      $ri$ := X6;
      CURRENT ENVIRONMENT[*random*] := X6;
```

Note that ri is addressed directly because it is global, whereas *random* is modified by X1, the current environment register. The fact that the expression is not evaluated strictly left to right helps a little, but could have been improved further if there was a negate and add instruction. In this case, a store negative instruction which does exist, could have been used instead. As usual, integer divide cannot be done directly, but it is unusual for two normalisation orders to be required on an integer multiplication.

The third example is more interesting as it shows the environment control and real expression handling. The statement

$$y := a - i \times x \uparrow 2$$

gives

```
X2 := CURRENT ENVIRONMENT [BACK STATIC];
CALL $a$;
CURRENT ENVIRONMENT := CORE COPY;
CURRENT ENVIRONMENT[TEMP] := ACC;
```

```
X2 := CURRENT ENVIRONMENT [BACK STATIC];
ACC := X2[$x$];
X3 := 2;
CALL INTEGER EXP;
CURRENT ENVIRONMENT[TEMP2] := ACC;
X6 := CURRENT ENVIRONMENT[$i$];
X7 := 0;
ACC := FLOAT (X6,X7);
ACC := ACC $\times$ CURRENTENVIRONMENT[TEMP1];
ACC := CURRENT ENVIRONMENT[TEMPI] $-$ ACC;
X2 := CURRENT ENVIRONMENT[BACK STATIC];
X2[$y$] := ACC;
```

As usual ACC represents the floating point accumulator. Since a, x and y are not local or global, but one static level back from local, X2 must be loaded with the correct environment address by going back one level through the static chain. Evidently, the setting of X2 is not remembered, as unless $\uparrow$ (integer) subroutine uses X2 it will still be set for the store to y. In other, shorter code sequences, the setting of X2 is definitely remembered. The floating of i, although done in a single instruction, is in fact an extracode, which is a special form of supervisor call to augment the ordinary instruction set. Note also, that there is a tendency for all the organisational instructions to appear at the beginning and the actual calculation at the end of the statement. This is unfortunate, since several 1900 processors allow overlap of floating point and fixed point operations. More typical examples of the same phenomena appear with subscript calculations. In this case, the final setting of X2 could have been overlapped completely if it had appeared in between the two floating point operations. Note that the floating point operations include a negate and add, which is used by the software.

The last example shows the parameter mechanism, which requires some explanation. After the subroutine call instruction, there appears one instruction for each parameter. This instruction either handles the parameter in one instruction or calls a subroutine (essentially a thunk). With value parameters, the value is calculated in a precall sequence, but it is stored in a working variable, which is accessed by the thunk. This technique partly avoids the overheads of a general thunk mechanism. The statement in this case is

$$s = s + f(5)$$

which gives

```
X2 := CURRENT ENVIRONMENT[BACK STATIC];
CALL $f$;
X3 := 5;            executed in $f$
CURRENT ENVIRONMENT := CORE COPY;
X6 := X6 + CURRENT ENVIRONMENT[$s$];
CURRENT ENVlRONMENT[$s$] := X6;
```

The code is very compact because no thunk is necessary since the parameter can be handled in one instruction. Even this could be improved; for instance, if f is called frequently from an inner level, then it would be worthwhile putting the first instruction in the code of f (and jump to $f + 1$ if this is not needed). It is not clear why the setting of X1 on return from the call of f is done by open code from a core copy. It ought to be possible to place this in the return mechanism.

The four characteristics measured as usual had a number of anomalous values. With the execution speed (on a 1.1μ 1907, similar ratios would be obtained with other processors), four values are relatively fast and two are slow. Knuth's Man or boy? is slow (ratio 6.5) owing to the thunk mechanism which must be used for the name parameters. The environment control with this test is very extensive, so that the use of only two index registers is apparent. Nearly 10% of the generated instructions consist of loading and storing the core copy of the current environment pointer. Clearly the organisation behind this is not ideal, because KDF9 does manage to execute the test reasonably quickly even though it only uses two registers for environmental control. The other slow statement is k := $l \times m$ (ratio 2.1) which is due to the standardisation instructions mentioned above. The fast statements are k := 1.0 (ratio 0.48) because of the compile-time type conversion, **begin array** $a[1 : 500]$; **end** (ratio 0.43), the switch (ratio 0.48) as simple switches are optimised, and x := $ln(y)$ (ration 0.29) due to special coding with $y = 1.0$. There are no significant deviations from the above figures for the instructions executed. The only exceptional value for the code size in instructions is that for the call of a parameterless procedure (ratio 0.31), which can be seen to be very compact from the second example above. The measure of code size in bits follows the same pattern.

The significant differences in later versions of the compiler is that storage is allocated at procedure level and array subscript optimisation can be performed (see 2.2.6).

The general characteristics of the compiler should now be clear. With remarkably few exceptions the 1900 compiler code was "locally good", that is, short sections of code did not have obvious defects. The major weaknesses are that call-by-name is handled inefficiently, and that value parameters are not handled in the most straightforward and efficient manner. It is interesting to note that the ALGOL 68 compiler for the 1900 does handle value parameters in the precall sequence. Because of the lack of index registers, the address of the top of the stack is not kept in a register, but parameters are still stacked by open code. ALGOL 68 has a significant advantage here in that the thunk mechanism is never necessary. There is an interesting problem in the compiled code with the use of X2. Since it is used for up-level addressing, almost any sequence of code has a potential requirement for X2. However up-level addressing is not common (once globals and blocks are ignored), so to dedicate X2 for this is a waste of a precious resource. In fact the compiler does use it occasionally for subscript calculation. The effect of this is that X2 is sometimes reloaded unnecessarily for up-level addressing. A conflict such as this on the use of a register is obviously quite hard for a compiler to manage.

9.9 Some Comparisons

Apart from the purely numerical characteristics given in 9.2, several other comparisons can be made. Unfortunately, the very important issues of compiler diagnostics and debugging aids were not considered (at least not deliberately!). The tests did contain several statements and constructs which were obviously capable of optimisation almost independently of the object code. A few of these appear in Table 9.2, a tick indicating optimisation.

	Atlas	KDF9	B5500	1108	6600	1900
$x := abs(y)$	X	X	$\surd$	$\surd$	$\surd$	X
step 1	X	X	$\surd$	$\surd$	X	$\surd$
if $x < y$ **then**	X	X	NA	$\surd$	$\surd$	$\surd$
$a[1]$	X	X	X	X	$\surd$	X
$x := 1$	X	X	NA	$\surd$	$\surd$	$\surd$
$root := 0.5 \times (root + j/root)$	X	$\surd$	X	$\surd$	$\surd$	$\surd$
goto $local$	X	$\surd$	$\surd$	$\surd$	X	$\surd$
$c := c + 1 \times 1 \uparrow 1$	X	X	X	$\surd$	$\surd$	X

Table 9.2: Optimisation Table

In two cases with the B5500, the obvious coding is optimal and these have been marked not applicable (NA) in the table. With the statement $root := 0.5 \times (root + j/root)$, 1108, 6600 and 1900 achieve some optimisation by calculating $j/root$ first, whereas the KDF9 compiler optimises by not reloading $root$ into registers (the common subexpression analysis, see 9.4). Only the 6000 compiler optimises $a[1]$ to a significant extent. The address is calculated out of a loop, so that a relatively simple indirect access is all that is required. This is not possible on the B5500, but the code could be improved significantly by avoiding the lower bound subtraction (see 9.5). Only 6600 and 1108 compilers do constant expression optimisation as illustrated by the last example.

The compilers can also be ranked in terms of the generality of the ALGOL 60 subset which is accepted. Some subjective judgement is necessary, but a reasonable order would be

CDC 6600,
ICL Atlas,
ICL 1900,
ICL KDF9,
Univac 1108,
Burroughs B5500

This order seems to bear no relationship with the performance measures.

Chapter 10

Improving an ALGOL Compiler

A frequent question asked by intending programmers is how efficient is a particular ALGOL compiler? Such a question is virtually meaningless—for instance, the Whetstone ALGOL compiler on KDF9 is twice as fast as the assembler (in terms of words generated per second), and it produces incomparably better diagnostics in program failure. When pressed to define what they mean by "efficient" one is usually told that a comparison of execution speed with good hand coding is required. The answer to this will depend critically on the application, and any features of the computer hardware that a hand-coder could exploit but a compiler does not. On most conventional third generation machines two features of ALGOL tend to be implemented very poorly compared with the equivalent construct which a machine-code programmer would use. These are array accessing and procedure calls. The reason for this is that the ALGOL 60 constructs are much more general than the basic requirements of most applications, and that this generality forces most compilers to generate code which does not take any advantage of the specialised, simple use of the facilities.

10.1 Procedure Calls

On almost all machines, the advantages of assigning storage at procedure level outweighs the slight complication in the compiler and run-time system (see 1.8). Block level storage works well on Atlas because of the large number of index registers and the fact that the rich instruction set allows environments to be preserved and restored quickly. In almost all other cases, block level addressing can be seen to be a distinct disadvantage—if only because fewer index registers are available for array accessing and local optimisation. Unfortunately it is unlikely that a compiler using block level addressing could be changed to procedure level without considerable effort.

Intimately related to environment control is the method used to implement call-by-name. A decision must be taken not only on the general thunk mechanism, but the degree to which this is by-passed if the actual parameter is "simple". As stated in 1.10, the best method is probably to execute an instruction which is placed in the stack. If this is not possible, then the evaluation of a thunk can include a test to see if the parameter is simple so that environment control can be avoided. However, both from the point of view of procedure calls and call-by-name, it is convenient if an environment can be quickly set-up. The simplest solution is back-chaining the static chain, since then no arbitrarily long display is involved. The 6600 PASCAL compiler did have a display in registers, but this was changed to back-chaining, which resulted in shorter and faster code (see Wirth, 1972). Two of the index registers that were freed by this change were used for input-output buffer points, allowing much faster peripheral management. One doubts if a change of this magnitude could have been made if the compiler itself was not written in a high-level language (PASCAL).

The really critical aspect of procedure calls is the parameter mechanism. The preponderance of value parameters (or other "simple" parameters) means that for a large proportion of cases a very efficient method can be used. This involves merely stacking the relevant information in the precall sequence in such a way that the called procedure need take no additional action. This technique demands that formal-actual parameter correspondence is checked at compile time (see 1.9 and 7.2). With such an approach, parameter passing should involve only two or three instructions per parameter. Because of the problem with formal procedures, two crucial interfaces must be decided early on in the design of a compiler: those with and without formal-actual parameter checking. These two interfaces together with the environment control method, determine a large proportion of the code generated by a compiler.

The necessity to optimise procedure calls can be illustrated by an example. The procedure *max(x,y)* takes 47 units to call according to the estimation method given in Chapter 3, whereas the open code only requires 7 or 8 units. With the illustrative machine, a procedure call and return is 9 instructions if no level pointers need be set-up and 16 otherwise; so even this is twice as long as the body of the procedure. To make any effective gains one must not set up the level in its entirety. Clearly the link data is unnecessary for such simple procedures. The method used by the KDF9 compiler is quite effective, although the body of the procedure must be severely restricted, for instance, it must not use up-level addressing or the stack (hence no further procedure calls). Then the stack register can be set to the position of the calling procedure and used to access the parameters. Alternatively, the parameters could be placed in fixed locations. Assuming also that such procedures are not called formally, the calling sequence can be changed so that the incrementing of the stack register is avoided, and no space is made for the link data.

The call of *max(x,y)* now becomes

```
LDA        X
STA        5,(0)
LDA        Y
STA        5,(1)
CALL,3     MAX
```

which is three instructions shorter than the general case. Obviously, the final store can be placed in the body of the procedure. The procedure body now becomes (from **if** $x \geq y$ **then** x **else** y).

```
MAX:       LDA    5,(0)    X
           SBA    5,(1)    −Y
           JAN    L1
           LDA    5,(0)
           JMP    L2
L1:        LDA    5,(1)
L2:        JPI    REG3
```

The KDF9 code is not as good as this, moreover the body of the procedure is unnecessarily restricted, for instance, access to globals is not allowed. Care must be taken with compiling *max(z,max(x,y))* since, in this case, the stack must be incremented to avoid x overwriting z.

Although the advantages of this optimisation are worthwhile, only short procedures need be considered. With the machine-code used for the illustrations, the following list can be made of constructions allowed and not allowed in those simple procedures:

Allowed	Not allowed
value parameters	arrays by value
arrays by name	name parameters
strings	label/switch parameters
simple variable declarations	procedure calls
array accesses	array declarations
access to globals	access to non-locals, non-globals
local jumps	jumps out of procedure

Unfortunately this form of optimisation can lead to complications with a post-mortem facility. Simple procedures do not leave any trace on the stack, so failure during such a procedure will not be handled correctly without a significant extra overhead in the system. In fact, just this difficulty arose with the post-mortem system for KDF9. In practice, the information printed has been adequate, but it has been necessary to inform users of the two different classes of procedures.

10.2 Array Access

In contrast to procedure calls, array access optimisation is frequently performed by compilers, and there is a considerable number of papers on the subject. Much of the work refers to FORTRAN but the ALCOR compilers (Grau, 1967), the Kidsgrove KDF9 compiler (Huxtable, 1963), the CDC 6000, IBM level F and 1900 XALT ALGOL compilers all do some optimisation. The problems are not easily discussed without reference to a particular machine because the optimal approach depends critically upon the index register structure.

One first requires an order of magnitude estimate of the gains to be made with such optimisation. Users of the ALCOR system quote a factor of 3 between optimisation and non-optimisation with bound checking (on 7094). However users of Atlas quote a factor of 2 between bound checking and no bound checking (but no optimisation, see 9.3). Estimates can also be made from the Whetstone ALGOL statistics (although these were from non-production runs). The table given in 2.3.1.10 is most easily used for such an estimate, although in particular cases use of the tables in Appendix 6 would give better figures. Ideally the loading of subscripts would be virtually eliminated, and the **for** loop control code would be significantly simplified. The net effect of this would be to reduce the equivalent of the operation Take Integer Result by about 30% and remove the necessity for INDex Result and INDex Address. The equivalent of the operation LINK would also be drastically reduced, so that about 30% of the total Whetstone elementary operations would be removed. Assuming the times for these operations are typical of elementary operations as a whole, then a 30% saving in time could be expected. This could easily be increased to 50% with longer running programs. Of course, particular programs which make very heavy use of arrays could well be two or three times faster, but on average one should expect a 50% improvement between non optimised code without bound checking and well optimised code.

Optimisation can roughly be divided into three types, safe, checked and unsafe. As an example consider one of the GAMM loops

for i := 1 **step** 1 **until** 30 **do** $a[i] := b[i] + c[i]$

Only loading i once for the three subscripts could be regarded as safe since (ignoring name call and functional side effects) this is certainly valid. However using an index register to count the thirty repetitions of the loop is technically unsafe owing to a possible assignment to the control variable (which can easily be checked for). Optimisation which is not checked for validity is unsafe. This is not necessarily an error. For instance, in ICL's optimising FORTRAN compiler a user can let the compiler perform "unsafe" optimisation where it will be assumed that no assignments will be made to COMMON by subroutine calls. In a similar manner, their ALGOL compiler will produce better **for** loop control code by assuming that the code is non-recursive. Some forms of optimisation reduce the volume of code whereas in other cases code is moved outside loops. For detailed optimisation one needs to make an analysis of the program's flow-structure, which is significantly easier with ALGOL than FORTRAN because of the smaller number of explicit jumps (see Allen 1971).

There are three main areas of difficulty with subscript optimisation. Firstly the dynamic allocation of array storage means that little calculation can be done at compile-time. Secondly is the possibility of assignment to the integers involved in subscripts apart from the for loop mechanism. Thirdly is the problem of possible side effects on name calls, procedure and function calls.

The problem of dynamic allocation of space for arrays, is of course, a fundamental part of the design of ALGOL 60. Many FORTRAN programs wish they had this facility, but it certainly makes array access more difficult. This can be partly overcome by storing the address of $a[0, \ldots 0]$ with the array storage information (see 1.2). With formal array parameters it is obvious that little else can be done at compile time since the addresses of the actual array elements will vary dynamically. However, the statistics given in 2.3.3.1.2 showed that in about half of array declarations the array bounds are constant (i.e. FORTRAN is half-right). In such cases, the array space could be calculated by the compiler and allocated in first order working store. No code is therefore necessary for an array declaration except in setting up the array information. Care must be taken with this to ensure that as a result excessive space is not allocated in first working store. For instance, the size of the fixed storage must not exceed the address length of the machine. The advantage is that $a[10]$ can now be compiled identically to the reference to a simple variable. This is very worthwhile as the statistics in 2.3.3.2 show. Optimisation of particular loops involving such an array also becomes easier although addresses are still off-set by the appropriate environment pointer. However, as many arrays are declared globally (or at least in the "global" procedure), further simplifications can be made.

The main problem is to keep the subscript information within the registers. Therefore any assignment to an integer used in a subscript must be noted at compile-time, otherwise invalid optimisation could take place. Other forms of optimisation are likely to involve noting the uses of variables so one can expect that such information is available. As fetches exceed stores in a ratio of about 3 to 1, and as not performing a store would mean that any post-mortem might be misleading, it seems wise, even with the most optimising compilers, always to write the value to core and not depend on the register value alone (it is interesting to note that the 370 cache stores use this approach).

The remaining difficulty is that of side-effects on name calls, procedure and function calls. Attempting optimisation round these is difficult since it would mean a global analysis of the program. Moreover, it is easy to see that the proportional gains in such optimisation is very small. A name call, etc., is likely to involve twenty or more instructions whereas saving one integer subscript in a register is only going to save about three instructions. Certainly dumping register values and restoring them on such calls is likely to waste more time than it saves (see 9.4 for an example with KDF9). Hence such optimisation even in the most favourable case is only going to save a few per cent, in contrast to most other forms of optimisation where loops saving a factor of three can be illustrated.

How good can one reasonably expect ALGOL compiled code to be? As a simple example, consider one of the GAMM loops for

```
for i := 1 step 1 until 30 do
    $a[i] := b[i] + c[i]$
```

The straightforward, locally optimised code on the illustrative machine is

```
      LDI,1     I
L1:   STI,1     I
      SBI,1     30
      JIP,1     L2
      LDI,2     A        controlled statement
      ADI,2     I
      LDI,1     B
      ADI,1     I
      LOA       1,(0)
      LDI,1     C
      ADI,1     I
      ADA       1,(0)
      STA       2,(0)
      LDI,1     I
      ADI,1     1
      JMP       L1
L2:
```

The following points have been noted. Firstly, **step** 1 optimisation has been done well, with only one load and store of the control variable in the control code itself. Also the address of the left hand side has been left in register 2. This is already quite good executing only 451 instructions compared with an average of over 800 for the six compilers considered in chapter 9. Code of this standard is not always possible in a straightforward manner. For instance on the 1900 real numbers occupy two words so it is necessary to multiply the subscript by two before adding the base address. Similar remarks apply to the 360.

The strategy used to make significant improvements will depend on the machine resources and how worthwhile the advantages are. A critical parameter is the number of index registers. In this area it should be noted that the use of register 2 is unnecessary since the code for evaluating the left hand side could be delayed until $b[i] + c[i]$ is in the accumulator. If sufficient registers are available to allocate one per array, then inside the loop it is merely necessary to increment the register appropriately. However it is much more usual, as in this example, to have fewer subscripts than arrays. Also it is the subscript which is incremented and not the array, so if this is kept in a register there is less to increment. The difficulty here is that in order to avoid the addition instruction on each fetch, some of the calculation must be performed out of the loop. For instance, if the relative addresses of $a[0]$ and $b[0]$ and $c[0]$ could be calculated, merely adding these values to the current register setting would give the address in one instruction.

In some cases it is necessary (or possible without additional overhead) to load i into a separate register, as on the CDC 6600. Obviously merely omitting the reloading of the register is fairly simple and should be expected. Of course, if it is necessary to double i, then this value can be preserved. It is interesting to note the action taken by the KDF9 optimiser. Since the loop is always executed once, the test and jump is performed after the loop itself in one instruction. Each array fetch is only one instruction using the auto-increment facility so that no additional incrementing code is required. A separate register is used for each array, which is initialised by a short subroutine outside the loop. The code produced in the loop is really quite remarkable as it is comparable with that which would be produced by hand coding. The ALCOR compiler could well produce code which is not as good as the straightforward form given above. The reason for this is that explicit code is produced in the loop to increment every array access which is linear in the control variable. This technique could give very substantial gains with two or more dimensions but is unlikely to gain with one dimension unless there is more than one array access per array. Almost all optimising compilers perform some optimisation which can be counter-productive. This is surprising since in most circumstances it is quite easy for the compiler to test for such cases.

The conclusion to be drawn from the above is that with most conventional machines the advantages in extensive array optimisation with one-dimensional arrays are slight. Local optimisation to reduce the use of registers, increase the parallelism of execution (see 9.7) and avoid the reloading of subscript values is probably sufficient.

The situation with two or more dimensional arrays is somewhat different. The access code is much longer without optimisation and so the potential gain is greater. It is, however, more difficult unless only the simplest cases are considered, mainly because the increment in the elements to be fetched is no longer fixed. With one subscript the increment will be a single storage cell, but with the other ones the value depends upon the array size. With formal array parameters this cannot even be determined by the compiler even if every array has constant bounds. Hence the best one can do in general is to calculate the increments outside the loop so that only addition is necessary inside the loop. However, if the array element addresses overflow the number of index registers, one may be forced to perform the incrementing in core. Unless convenient instructions are available in the processor, the best code one can expect for access to any array element in a loop with a constant step is

```
load address $a[s_1, \ldots s_n]$
fetch/store $a[s_1, \ldots s_n]$
add increment
store address $a[s_1, \ldots s_{i+k}, ...s_n]$
```

This code is not very much better than that for a two dimensional array using the code-word technique (see 1.2.3). Since arrays of three or more dimensions are relatively rare, the advantages of this form of optimisation could be quite small. If array element addresses can be kept in registers, then the gains become much more significant. But this is likely to reduce significantly the size of the loop which can be handled. Of course, multiple uses of the same variable within a loop swings the balance in favour of optimisation. In this case, merely noting the address on the first use could be adequate and a lot simpler. It must be remembered that referring to $a[i, j]$ and $b[i, j]$ cannot easily be optimised because a and b are not necessarily the same size.

Hence the benefits to be obtained from optimisation depend critically upon the ordinary non-optimised code and the extent to which the optimisation can exploit additional facilities in the machine. To specify ALCOR-style optimisation for a compiler without due consideration of these points could be very expensive in effort for a modest gain. To take a few examples, optimisation on Atlas could be very effective because all addresses could be kept in registers since enough could be allocated for this purpose. Although array access is straightforward and reasonably efficient, a significant gain could be made. The case for optimisation on KDF9 is more dubious as there are fewer index registers, however ordinary code for array access suffers from the disadvantage of having to load the index registers via the stack. In fact the Egdon KDF9 compiler does significantly better than the Kidsgrove compiler (without optimiser) by arranging that the "array word" contains enough information for open code generation of two-dimensional array access (i.e. address of $a[0, 0]$ and column size). Therefore on KDF9 care must be taken to ensure that optimisation does lead to a genuine gain. On the 1900 with only three index registers, it is unlikely that very significant gains can be made with optimisation, because registers cannot be dedicated in any way. Also the incrementing of addresses in a for loop cannot be performed without a register (although it need not be an index register). To save one multiplication in a two dimensional array access could require as many instructions to increment the address. Gains will be made if the incrementing can be performed much less often than the number of accesses. On the other hand, on the CDC 6600, having decided to represent integers in floating point form, some optimisation is almost essential.

Optimisation on the 6600 is achieved by the ALCOR technique suitably modified for this machine. Every **for** loop control variable is maintained in floating point and fixed point form. In this way access to an array element involving only **for** loop control variables can be made quite rapidly. This global optimisation is somewhat spoilt by unnecessarily reloading index registers (for instance, with the GAMM loop above). This may have been rectified with release 3 of the compiler as the GAMM loops are executed 30% faster.

10.3 An Example of Peephole Optimisation

The author was able to make a 20% increase in the execution rate of ALGOL code on KDF9 by means of peephole optimisation. This technique, described by McKeeman (1965) makes use of a short buffer of generated machine code or its equivalent. The buffer is analysed to see if any code sequence can be replaced by a logically equivalent sequence which is faster or shorter (usually both). The technique has a number of advantages, firstly it is not necessary to understand the compiler in detail, and secondly one can show that such optimisation is safe.

With the KDF9 Kidsgrove compiler the code generated is capable of being locally optimised for two reasons. Firstly the code for some operations (ie. array access) which is radically different from the code using "optimiser" is very bad. Secondly the code is generated in a sequence of macros which take little account of their predecessors. The code which would be improved by the optimiser is array accessing and the for loop control code. Both of these could have been modified by direct amendment of the compiler. The second cause of locally poor code cannot really be handled without the

	Context	Instructions executed		Estimated	Estimated space gain
		Old code	New code	speed gain (%)	(2500 word program)
(1)	one-dimensional array access	5	2	4.5	109
(2)	overfbw on assignment	1	0	2.2	54
(3)	two-dimensional array access	5(+14)	10	3.5	−10
(4)	removal of redundant REV instructions	1	0	small	5
(5)	relational operators	4,5 or 6	2	0.15	8
(6)	for loop control code	7	3	0.9	28
(7)	call of simple procedures	2	1	small	50
(8)	$\uparrow 2$	2(+26)	2	2.0	small
(9)	$\uparrow 3$	2(+32)	4	(with above)	small
(10)	integers in real expressions	2(+20)	2 (shorter)	5.7	8
(11)	many (loading literals zero and 1)	1	1 (shorter)	small	20
(12)	exit from machine code proc.	2	1	small	5

Table 10.1: Kidsgrove improvements

equivalent of peephole optimisation.

The Kidsgrove compiler produces assembly code text. This was very convenient because it meant that there was an interface within the compiler generating machine code which was well defined. Two facts about the assembly code were also known which made the technique viable. Firstly, it is not possible to jump to an unlabeled location. Some assemblers allow for instance a jump forward of six locations (JMP* + 6) to a destination which may not be labelled. If the compiler generated instructions of this form (in an indeterminate manner) it would be necessary to ensure that every replacement was the same size in instructions. Secondly with one exception it was known that the generated code was never dynamically altered. The one exception was concerned with avoiding the reallocation of space for an own array.

The basic technique used was as follows. The code generation brick of the compiler has a subroutine which accepts characters of the assembly code text. This subroutine was modified to incorporate a six instruction buffer. When a new instruction is added to the buffer, a check is made to see if it belongs to a set of about 15 "triggers". If it does, then a code sequence is executed in the subroutine to see if an edit of the buffer is required. Otherwise the instruction is pushed down in the buffer and the excess instruction over the six is sent to the assembler. The subroutine is rather slow as it is given characters rather than complete instructions, and also because every generated instruction must be compared with the triggers.

The effect of these changes is illustrated by Table 10.1. Included also is one change made by directly modifying the compiler. The checking of overflow on assignment was made an option, and the figures below related to use without this check (the default case). An additional overflow check was added to the control routines for procedure exit (see 1.1).

The estimates have been made on the basis of the statistics gathered on KDF9 and given in chapter 2. The total processor speed gain is 18.9%. As a significant proportion of time is spent executing machine-code routines (input-output, standard functions and other routines) the actual net saving is probably about 10% of the total ALGOL execution time. The space saving is also about 10% with a 2500 word program.

Some of the cases optimised are of interest, as noted below.

(3) Not all two dimensional array accesses are optimised. The second subscript must be an integer or constant. The performance calculations assumed that this was so 90% of the time (in 2.3.3.2 a figure of 11/12 is given).

(4) Redundant REV instructions occur when operations are not executed in the ordinary reverse Polish manner. This happens when common subexpressions are optimised and when a function designator is the second argument of a binary operator.

(5) The extra code is due to standardisation of Booleans which is unnecessary with **if** $x > y$ **then**, etc.

(6) This is due to no optimisation for **step** 1. This optimisation partially overcomes this and removes a vestigial jump to cater for multiple **for**-list elements. Complete optimisation for **step** 1 would have to be done directly to the compiler since the code is both before and after the controlled statement.

(7) A redundant parameter is put into the nesting store which is immediately erased on entry to simple procedures. The parameter is required for the link data for up-level addressing which can never be involved with simple procedures. An additional label is added to the procedure after erasing the parameter so it need never be set. This optimisation will only occur to the procedure calls appearing after the body of the procedure.

(10) The call of a float subroutine is removed and replaced by loading a different literal and a shift. The large estimated gain is surprising. This is based upon the static count for the float subroutine (see 2.3.4.2), assuming that the subroutine appears equally often inside loops. As most programmers appear to be unaware of the possible subroutine call, there seems no reason why it should be less common inside loops.

(11) There is a short form for loading zero into the nesting store which is not generated in one case. One register must contain 1 so a short form for loading 1 is possible which gives a significant gain in space. The addition and subtraction of 1 is performed by logical operators which is both quicker and more compact.

A major reason why this technique was successful is that statistics were available to estimate the frequency of various sources of inefficiency. Numerous other examples of code could be produced which would be improved by local optimisation, but in no case was it sufficiently common to make it worthwhile. One cannot expect a compiler never to produce locally bad code, but similar sequences of bad code should not be common.

With a language like ALGOL 60, it is convenient if code generation can be determined fairly closely by the syntax. An over-rigid adherence to this is very likely to produce code which is locally bad. Actual machine-code, especially assembly code text, is a poor input language for peephole optimisation, so a language level slightly above this would be better.

10.4 The Use or Registers

The nested, recursive structure of the syntax of ALGOL 60 is quite unlike the structure of the target machine. It is therefore not surprising that compilers have difficulty in matching the two. To take a simple example, the obvious way to translate an expression is by always leaving the result in a fixed accumulator. This will lead to unnecessary load and store operations unless the compiler takes remedial action. A fixed allocation of registers to various tasks has very significant advantages because it means that the interface of subroutines and compiler-generated macros are simple to state. ALGOL 60 has significantly more problems in this area (than FORTRAN, for example) because many functions can only conveniently be handled by subroutine. On almost all machines, it is not possible to vary the registers used to pass arguments to a subroutine.

The problems can be illustrated with the machine used for the examples. The register conventions adopted have been

REG1	integer expressions
REG2	complex integer expressions, subscripts to integer arrays
REG3	subroutine links, up level addressing
REG4	current level base address
REG5	address of top of stack

To this must be added the use of the single floating point accumulator for real calculations. Note that if the registers are general purpose, possibly grouped in pairs for some operations, then the allocation can be made even more freely.

If any change is made to the use of a register then numerous subroutines and macros could be affected. The possibilities for error are endless because trouble might only arise if (for instance) an integer divide is performed with a subscripted variable. The major subroutines that are likely to be critical in this respect are:-

array access with bound checking
array access of "large" dimension
operators performed by subroutine ($\div$, $\uparrow$, etc)
name-call
function call
procedure call

Some registers are likely to be used quite rarely with a fixed allocation scheme. With the illustrative code, registers 3 and 5 could remain unused over substantial sections of code. Register 3 could quite easily be used for other purposes, but the compiler must check because any variable access could involve up-level addressing. Register 5 would need to be restored, since the procedure and name call code depends on its value being maintained. Register 4 is the most important one, and it is hard to see how any ALGOL compiler could survive without dedicating a register to this function. This is the most easily noted consequence of dynamic storage allocation in ALGOL 60.

Any system for register allocation is bound to depend upon how critical the resource is. The six compilers considered in chapter 9 adopt radically different methods. On Atlas registers are not scarce so block level addressing is not extravagant. The calculation of integer expressions (where possible) in the registers rather than in the floating point accumulator would lead to a better use of this resource. However, a significant gain could be made by using registers for ALCOR-style optimisation of array access, and with a use of registers for the **for** loop control variable. On KDF9, the design of the compiler attempted to leave as many registers as possible free for optimisation. As a result, if optimisation is not requested, ten of the fifteen registers are not used. As mentioned above, the decision to optimise round procedure calls by dumping registers using a subroutine can too easily lead to a use of the registers in a counter-productive manner. As subroutines are relatively expensive on KDF9, it would have been best to concentrate on “local” optimisation which could be shown to be demonstrably better in terms of execution time. On the B5500, the addressing registers cannot be optimised as their use is determined by the microcode for the procedure call and array accessing. One could, however, use a one-dimensional sub-array of a two dimensional array in a one-dimensional manner. This would really require a language extension, since the code would be very sensitive to the method of scanning arrays (i.e. by columns or rows). The 1108 compiler could free a significant number of registers by adopting procedure-level addressing. There would then be enough registers to consider ALCOR or local optimisation. Similar remarks apply to the 6600 where the gains would be very significant. In this case freeing the index registers must be accompanied by a use of fixed point variables to represent integers in order to exploit the hardware fully. On the 1900, the resource problem with only three index registers is severe. The compiler handles this well, although there is some evidence that the second register could be utilised more fully. To satisfy oneself that registers are being used adequately, a simple program can be written to analyse the object code produced by the compiler. The binary code should not show a regular instruction pattern caused by a restricted use of the available machine facilities. It has been reported from Bell Laboratories that the accidental statistical analysis of a systems tape revealed a distinct “clear and add” effect!

10.5 Some Optimlsation Techniques

Two methods of optimisation have already been mentioned, the peephole and ALCOR techniques. These two are largely complimentary, and a suitable combination of both of them should lead to very tolerable object-code. Several other methods are appropriate to ALGOL 60 although they are not in wide use.

Global optimisation in ALGOL 60 is inhibited to a certain extent by the necessity to check for side effects on name, function and procedure calls. An alternative to checking the validity of the optimisation is to allow the user to specify whether this loop or procedure, etc., can be optimised. The difficulty here is the assumption that the user is sufficiently aware of the logic of his program and the consequences of the form of optimisation to make a reasonable choice. In exceptional cases this may be so, but by far the largest majority of programmers do not have the necessary knowledge to make such a choice. Additional information given to a compiler could include an estimate of the number of times a loop is executed (or even produced automatically from methods given in 5.3.6). This could be used to determine whether code should be space optimised or speed optimised. The first FORTRAN compiler did include such a user option, but it had to be incorporated with a large amount of flow analysis in order to allow effective statement to statement optimisation. In ALGOL 60 simple masking operations can be used to determine valid register settings round for loops and conditionals.

A reasonable approach to ALGOL 60 code generation would be to use the technique of “basic blocks” used with some FORTRAN compilers. For ALGOL this would consist of one entry point, any number of exit points (being entry points to other basic blocks) and no internal name or procedure/function calls. A block therefore represents a unit which can be optimised without regard to the rest of the program. Independently of this, global information about the program structure may allow optimisation over the basic block boundaries.

An example of a basic block could be a simple for loop (excluding the initialisation of the control variable), which does not contain a procedure/ function or name call. In such simple cases, placing the control variable and critical subscript values in registers should present little difficulty. Such blocks could be assumed to be less than a certain size (otherwise, split them up unnecessarily), so that the analysis can always be performed with the tables in the core-store. The analysis could be machine-independent, although the output cannot be as it will depend upon the machine architecture.

Chapter 11

The Architectural Requirements of ALGOL 60

The production of efficient object-code by a compiler depends upon the facilities that the machine offers. If some facilities are missing it may be possible to program round this by suitable subroutines, but when the overheads become high it may be best to omit some language features. The absence of adequate storage allocation methods have, for instance, prohibited most implementations from considering dynamic own arrays. On the B5500, the supervisor does give the necessary allocation support, so dynamic own arrays are provided. In this chapter, the more important features of ALGOL 60 are considered in relation to the architectural requirements they impose.

11.1 Address Lengths

In ALGOL 60 all accesses to user-declared variables can be handled by small offsets from the display registers (see 9.3). It is difficult to find an offset in excess of seven bits, since this would require that the user had declared more than about 120 simple variables or arrays in one block. It would appear, therefore, that the address length within an instruction could be significantly less than is common on most third generation machines (especially Atlas with 24 bits). This belief was put to the test by the following experiment (for full details see Wichmann, 1972b).

In order to make any quantitative assessment of different addressing mode facilities it is necessary to have a model of the behaviour of program variable access. With a program containing V declared variables and U uses of those variables (i.e. source text references), the model should allow one to determine how many variables account for any given fraction of the total number of uses. With a short address length, a large program may contain variables that cannot be accessed directly. A model should allow one to determine the effects on performance that such a system would have.

A modified version of SOAP (see 6.1 and Scowen, 1971) was used to obtain details of the uses and declaration of variables.

Statistics were gathered from eleven programs giving a histogram of the total number of variables used once, twice, three times, etc. Although analysis of more programs would have been worthwhile, a significant pattern of behaviour did emerge.

If X represents the number of uses, and Y is the number of variables, then $Y = F(X)$, the function depending upon the total number of variables and uses within the

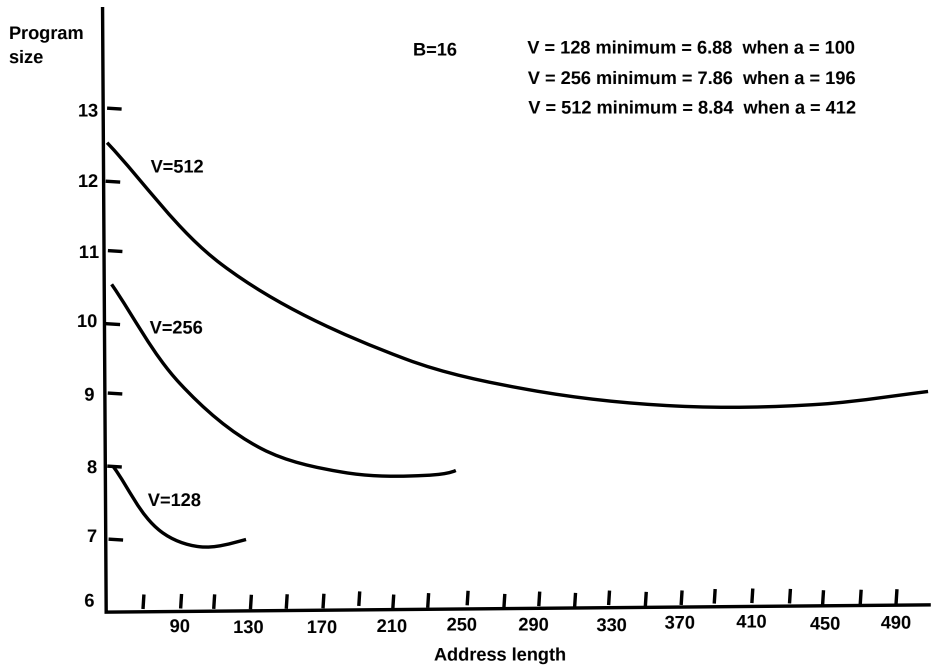


Figure 11.1: Variation in number of declared variables

program. In fact

Total number of variables	$V = \Sigma\, Y$
Total number of uses	$U = \Sigma\, XY.$

A good approximation was found to be

$$Y = \frac{V^2}{(U-V)}\left(\frac{U-V}{U}\right)^X \qquad (X = 1, 2, \ldots).$$

One can use this model to calculate the optimal address length within instructions, which will minimise program length. This becomes large if the address length is large (as on Atlas). If the address length is too small, then too many long forms of the instruction or extra instructions need to be generated. If A locations can be accessed by an offset from an index register, but for accessing beyond this a penalty of B bits is involved, then the program size is given by

$$(U - P)log(A) + BP + constant$$

where P is the number of uses within the program which cannot be accessed directly. Graphs of this function for typical values of the arguments are shown in Figs. 11.1 and 11.2, giving the minimum for A at about 70% of V, the total number of variables in the program. The minimum is quite a shallow one, but clearly the address length should be at least half the size of the number of variables expected in large ALGOL 60 programs (such as the B5500 compiler which uses about 300 global variables). Hence an address length of 7 or 8 bits appears adequate for access to data items in ALGOL 60.

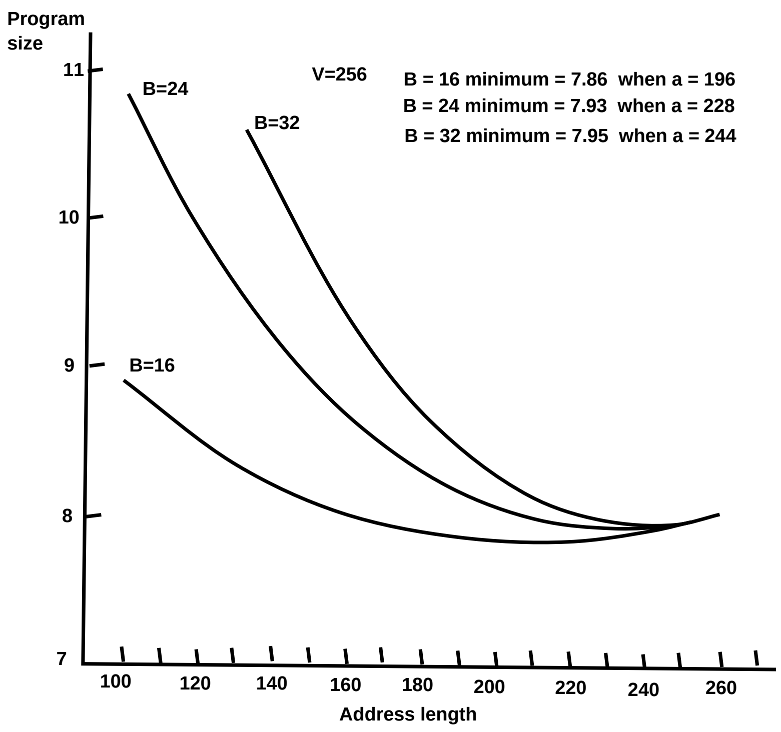


Figure 11.2: Variation in address length extension

Eight bits is also adequate for implicit jumps and local gotos provided the address is treated as a signed integer to be added to the current program address. The other major use is as literals for which eight bits is more than adequate. This leaves the problem of procedure and name calls which can use indirect addressing.

11.2 Registers and Addressing Modes

On a conventional machine, about three index registers appear to be adequate for ALGOL 60, although good use can be made of up to about eight index registers, but beyond that effective use must depend upon moderately sophisticated optimisation techniques.

The requirements with a non-conventional machine depend upon the extent to which the architecture is ALGOL-like. On the B5500, no registers are needed since the addressing logic uses the hidden registers to give the desired effect. However, to rectify the restrictions of no up-level addressing on the B5500 requires a hardware change.

There are three basic uses for which integer registers are required. Firstly index registers for the display, secondly index registers for array access and run-time routines, and lastly for integer expressions. On the other hand, multiple floating point accumulators are often poorly utilised by the compiler. Atlas manages successfully with just one, and on the 1108, where a large number of general registers are available, they are almost always used for integer values.

The procedure call mechanism can be simplified if offsets in an address field can be signed. This allows the parameters to be placed before the link data. The order of the words in the procedure base can then correspond to the order in which they are evaluated—parameters, link data, first order working store. The CDC 6600 does allow this, and procedure parameters do have negative addresses, but the run-time type checking precludes the fast call.

With a conventional machine two aspects of ALGOL are poorly handled, namely procedure calls and array accessing with bound checking. Ideally these could be handled by special instructions: so, for instance, the difficulties in dealing with array elements of different lengths, which are very apparent on 1900 and 360 computers could be accommodated in microcode. The lack of such general instructions on existing computers is very surprising since in many cases the machines were designed after the rise of FORTRAN and ALGOL 60. Hardware assistance in the name-call of ALGOL 60 would be more difficult to defend as a general facility since the requirement is due to a peculiarity, if not a mistake of ALGOL 60. Instructions to simulate a stack are not important for ALGOL 60. The possible exception is with the evaluation in the pre-call sequence of parameters. On KDF9 a simple variable parameter can be evaluated and stacked in two instructions—the only overhead attributable to the parameter itself.

Indirect addressing is not necessary for ALGOL 60. It could be used for a dynamic optimisation of name-calls, but the parameter mechanism is best handled by using the value technique (it is more efficient and safer, see Hoare 1966).

11.3 Arithmetic Facilities

The ALGOL report is careful not to specify any of the properties of integer or real arithmetic. This has been one of the most successful unspecified features of high level languages. In detail the nature of floating point differs significantly, but the general characteristics are sufficiently similar for the differences to be ignored in the design of high-level languages. However, the numerical analysts would like to have greater control over the form of the computation. Although the order of evaluation of expressions is defined adequately for their purposes, the lack of any double precision option is a substantial drawback. No solution to this can be regarded as ideal since any system is necessarily a compromise between ease of use and degree of control. The FORTRAN option does not always have the desired result—DOUBLE precision on a 360 will be required more frequently than on a CDC 6600 which has roughly twice the word length. The 360 level F ALGOL compiler allows one to use either 32 or 64 bit arithmetic throughout a program. Since there is little difference in basic speed on the faster processors, the main change is that 64 bit arithmetic will require more internal storage.

The vast majority of integer variables are used as simple counts and subscripts where a short length is no hardship, and is an advantage since storage economy could be achieved. However, a random-number generator can prove awkward with a short word length.

As far as ALGOL compilation is concerned, it is advantageous if integers and reals are barely distinguishable, as on the B5500. This means that conditional expressions involving an integer and real can be handled easily (see 1.5), as can the dynamic type conversion involved in $\uparrow$ (see 1.1.4.2), and type conversion involved with call-by-name (see 1.10.3). However language restrictions in this area, combined with good compile-time type handling, can provide a reasonable implementation with completely different forms for each type.

Several computers have a length for the address modification register which is much less than the word length (as used for integer arithmetic). On KDF9, for instance, only 16 bits of the 48 are actually used in subscripting. No check can easily be performed on the strict validity of a subscript because the instructions to load an index register ignore the top 32 bits of the word. Similar action is taken on Atlas and 1900, but not on 360 or the B5500.

11.4 Program Control Instructions

This subject has already been touched upon in the discussion of the address length for operand access (11.1). The problem is most easily analysed by considering the Whetstone ALGOL system. The major program control instructions are (in order of use):

Name	Context	Frequency
LINK	for loops	84200
Unconditional Jump	see 2.3.1.4	24600
Call Function	procedure/function call	19400
For Return	for loops	19300
FORS2	**step-until**	17800
If False Jump	conditions	15400
End Implicit Subroutine	end of thunk	13200
RETURN	end of procedure/function	2730
For Block Entry	for loops	2700
For Statement End	for loops	2510
FORS1	for loops	2330
GoTo Accumulator	**goto**	2010

It is clear that implicit jumps and procedure calls are very much more frequent than explicit **goto**s. The length of the implicit jumps are likely to be very short so that they can be handled by short off-sets from the current program address—also allowing easy relocation of code. With conditions, they can be implemented as in Whetstone and the B5SOO by always producing a Boolean result and then using an If False Jump instruction. Providing a standardised Boolean can be easily obtained, there seems no good reason to use the Whetstone method, as a compiler can very easily achieve the same effect by use of traditional conditional jump instructions.

Provision of special hardware for loop control can also be avoided by relatively simple optimisation. Since other languages have slightly different requirements in this area, any hardware would have to be devoted to ALGOL 60 alone. The B5SOO makes no special provision in this area, and the Whetstone system of instructions is too complex to be regarded as a satisfactory general solution.

11.5 Core-store Administration

On a conventional machine, a contiguous area is allocated by the operating system to the program. This area may be subdivided into code and data, as on the PDP10 and Univac 1100 series. The stack mechanism allows a flexible and efficient means of allocating storage for both arrays and simple variables, the main overhead being a few instructions on procedure entry and exit.

The simple stack mechanism is used on the Atlas system (see 9.3), but the operating system pages the store in addition. This has obvious advantages in that larger programs can be run, the less frequently used parts being kept mainly on drum. Also the available core space can be shared more effectively between processes. In allocation of space, the compiler ignores the page boundaries, and in consequence semantically related items may well lie on different pages. The construction of effective algorithms for such an environment is not easy (see Brawn, 1970).

A radically different approach is taken by the B5500, X8 (Eindhoven) and B6500/6700 systems which use the structure of the ALGOL language to determine the storage mapping. First order working storage is allocated on a stack in the conventional manner, but arrays are stored separately. The difficulty with this approach is that care must be taken to ensure that array space is controlled correctly. The advantage is that the operating system can place arrays that are not in use on secondary storage. The unit of transfer to secondary storage is a single user-defined structure which is likely to have a more consistent access behaviour than a page which contains a mixture of data structures. Array declarations in such systems are relatively expensive, but allocation is sufficiently rare for this not to be a handicap.

The stack mechanism of ALGOL is very successful as it can be easily implemented on conventional machines, and the allocation of arrays off the stack will exploit segmentation if the operating system allows this.

11.6 Program Support

High level language software could be made more robust, developed more quickly and reliably if more hardware support were available for program validation and monitoring.

The list of run-time errors in 5.4.5 indicates that subscript-bound checking and a test for the use before assignment are cases where special hardware could be of greatest benefit. On the B5500, for instance, accesses to a particular variable can be monitored by placing the descriptor of the monitor procedure in the word ordinarily taken by the variable itself. The production of program profiles showing the number of times every program part is executed is essential for the construction of effective software. Accurate program timing and instruction counters are also helpful. Unfortunately even if the facilities exist the supervisory “firmware” may not allocate time or instructions to each process on a sufficiently fine scale.

In some cases, a debugging aid may require interpretation of the machine code instructions. On 360 this is quite efficient (see Satterthwaite, 1971), but on a machine like the KDF9 it is not, because there is no fast method of unpacking instructions within a word.

Chapter 12

Language Design

The publication of the ALGOL 60 report was one of the.greatest land marks in modern computing. It set new standards in language definition which have, unfortunately, not been maintained by several major languages defined since then. After ten years it would be surprising if revisions were not thought worthwhile. The success of ALGOL 60 is reflected in the abundance of programming languages whose basic structure is derived from it.

12.1 The Mistakes of ALGOL 60

Much has been written criticising the ALGOL 60 report, mainly from a legalistic rather than practical viewpoint (Knuth, 1961; 1967). Within this book several points have arisen which cause some trouble to the compiler-writer (see chapter 7). Here a summary is made of the major points.

12.1.1 COMMENTS

As noted in 4.1.1, it is inconvenient that comments cannot be removed by a lexical scan, as they can in FORTRAN, ALGOL 68 and PL/I. It is also inconvenient to the user that he cannot put a comment between any two basic symbols. For example, it is very desirable to place a comment next to every variable declared. Nothing is more likely to ensure absence of comments than making them inconvenient to write.

12.1.2 PARAMETER SPECIFICATION

The fact that the parameters to formal procedures cannot be specified has had a very detrimental effect on ALGOL implementations. Although the majority of ALGOL compilers do require complete specification, the existence of formal procedures presents a loop hole to compile time checking of parameters (see 7.2). Consequently only a few compilers check parameters completely at compile time, the others incurring a significant penalty. This is a classic error in language design since the addition of formal procedures has had an adverse effect on ordinary procedure calls, even though they are 100 times less frequent.

12.1.3 CALL-BY-NAME

The call-by-name concept of ALGOL 60 is much more general than most applications require. The call-by-address of FORTRAN is more straightforward, although the ALGOL call-by-value is better than both. On many machines the value mechanism is more efficient, as indirect addressing necessarily involves an extra core-cycle which can be an embarrassment on a fast machine with slave stores and instruction look-ahead (see Hoare, 1966). This effect can be off-set by copying the value into the procedure and then writing it back on exit. With FORTRAN, if the actual parameter is a constant, then a special working variable must be given this value, and its address passed to the subroutine. The ALGOL value mechanism does not require this, and is more efficient if the actual parameter is a constant. The value mechanism corresponds much more closely to the machine-code practice of placing parameter values in registers before calling subroutines.

The call-by-name mechanism is fundamental to ALGOL 60, so it is impossible to consider any change within the framework of the language. The most notable attempt at an improvement is the result mechanism of ALGOL W (Wirth, 1966a). This is roughly the converse of the value mechanism. The definition of the value method involves an assignment of the actual parameter to the formal parameter within the procedure. With the result method, an assignment is performed from the formal to the actual parameter just prior to procedure exit. This mimics the value-address mechanism of FORTRAN, but type transfers could be involved on the assignments (which depend on the actual parameter). Hence the mechanism is much less straightforward than call-by-address, especially if a large number of data types are assignment compatible. With the 360 ALGOL W compiler, both the value and result mechanisms are handled in the procedure rather than in a pre/post call sequence. The assignments are done by in-line code if the types are identical, but if a type change is required, subroutines are used. Unfortunately the pre-call technique given in 1.9.3 (which becomes post-call for result) requires two assignments and hence must be rejected on the grounds of code compactness.

The generality of the name mechanism allows it to be used for a number of purposes —results, Jensen's device and even the monitoring of parameter access. Several methods must be used instead so that programmers must have the necessary expertise to use the most appropriate method for any particular parameter. The ALGOL W compiler now issues a warning on declaring a name parameter rather than value or result.

The ideal situation is where the parameter mechanism itself is independent of the implementation method. Hoare (Engeler, 1971) has given conditions for this, namely that all parameters must be distinct and different from any non-locals accessed by the procedure. Also further restrictions on the language structure were necessary which effectively precludes existing systems. It would be interesting to see this approach applied to a general purpose programming language.

12.1.4 LACK OF INPUT-OUTPUT

The lack of any standard input-output prior to the implementation of the major compiling systems has been a serious drawback in the use of ALGOL 60. Several have adopted a set of procedures implying no language extension (KDF9, Atlas, 1900). Others have used "procedures" having a variable number of arguments (1108, CDC 6000). The most radical deviation from ALGOL 60 introduces FORTRAN style format and write statements whose parameter structure is quite unlike that of ordinary procedures (B5500, Elliott family of compilers).

In practical terms the portability of ALGOL 60 programs is severely curtailed unless the input-output is localised. This is fairly easy to manage with programs written to be portable, but the vast majority of existing programs are not of this nature, and so require conversion by program or a sophisticated macrogenerator like ML/I (Brown, 1967). A very effective system for handling several ALGOL dialects is described by Hopgood and Bell (1967). In contrast, the difficulties caused by different hardware representations are minimal and can be handled by character conversion and a good editor. One particularly annoying aspect of writing portable ALGOL programs is that on the B5500 there is no equivalent of strings or string parameters. Also, some systems demand that a whole line is output in one statement which may not be convenient because of the way the program was structured. The fact that strings cannot be analysed within an ALGOL program means that it is virtually impossible to write an implementation of a set of general procedures in ALGOL (using the existing input-output facilities).

12.2 Features that could have been in ALGOL 60

In this section, extensions to ALGOL 60 are considered which do not alter the existing framework, but give an additional facility which is quite separate from the rest of the language.

One extension was noted by Higman (1967). Nothing is more error prone and cumbersome than having to write

```
**if** $j = 0$ **then**
        $write\ text(30, `monday')$
**else if** $j = 1$ **then**
        $write\ text(30, `tuesday')$
**else**
        $write\ text(30, `other')$
```

when the following is much more natural

```
$write\ text(30,$ **if** $j = 0$ **then** $`monday'$
                 **else if** $j = 1$ **then** $`tuesday'$
                 **else** $`other')$
```

Clearly this facility could be extended to array, switch and procedure parameters. Note that this can already be done with labels, integers, reals and Booleans, where the value mechanism implies that the choice is made before the procedure body is executed.

A modest increase in the control structures of ALGOL 60 would be worthwhile. A not infrequent occurrence in a program is the construction

```
**for** $i := i$ **while** <Boolean expression> **do** ...
```

when an ordinary while statement is all that is required. Of course, labels can be used to overcome this, but they can obscure the program structure if over used, apart from being complex to implement with a dynamic storage allocation language. Control structures in programming languages is a subject of its own. BCPL (Richards, 1969) has a very good set, and BLISS (Wulf, 1971) has such a rich set that no **goto** is provided. This is indeed a radical departure for what is a systems programming language.

The other major control form omitted from ALGOL 60 is the case statement (or case expression). This is a much more natural form for what is often written as a switch. Not only are switches harder to compile, but they are more difficult to use. Inevitably a user declares simple labels rather than designational expressions in a switch list, and the identifiers used are $l1$, $l2$, $l3$, etc. This could be improved by having no switch list as such but merely an indication of the extent of the subscript, i.e. **switch** $ss[10]$;. Then ten labels must be declared in this block denoted by $ss[1] \dots ss[10]$. This would be both easier to compile and use, and less radical than a case statement. PASCAL (Wirth, 1971a) combines the case statement with numeric labels to achieve a very reasonable system. The difficulty with case statements is that the code executed is chosen by the position in what could be a substantial list. One may, therefore have to count statements in the source text to determine the flow of control. Numeric labels overcome this in a similar manner to the use of $ss[2]$ as a label.

The important aspect of control structures is that they improve program legibility, are easy to compile, and are often more efficient than explicit jumps. A major extension can be made to ALGOL by allowing part of the procedure mechanism to be invoked at block entry. This mechanism consists of two parts which are not very well separated. Firstly there is the calling and exit mechanism with its associated environment control. Secondly there is the formal-actual correspondence—the passing of thunks, the value assignment and associated logic. Making this parameter mechanism available at block entry would enhance the language considerably. To quote an instance, the value assignment is the only mechanism in ALGOL 60 for initialising a variable on declaration, whereas one ought to be able to write

integer i **value** <expression>;

Here the expression must only involve outer block variables as with array bounds. The action is to make the value assignment as if i were a "parameter" of this block. This introduces implementation complications but since the evaluation of non-local expressions is required for array bounds the necessary mechanism must exist already.

A similar construction could be used for the name mechanism, say

integer j **is** <expression>;

or more usefully

array a **is** b;

The concept of choice being extended to array parameters would then automatically allow

array a **is if** $bool$ **then** c **else** d;

Note that the block structure automatically ensures that within the code in which a can be accessed, it is defined. Although the machine code produced for this construct could be regarded as a form of assignment statement, to introduce it in the language as such would be very dangerous indeed since it would be difficult to check the validity of any array access. The ALGOL block can perform many useful validity checks on a program which would be hard to implement without such a structure.

Providing string operators and the like is too radical to be considered here since it would almost certainly imply a different storage allocation mechanism, but the parameter mechanism already involves the implicit assignment of strings to string variables (actually only the addresses are so assigned, of course). So extending the block entry mechanism, one could have

```
string $day$ is if $c = 1$ then '$monday$' else if $c = 2$ then '$tuesday$'
                 else '$other$')
```

Again, the block entry mechanism ensures that day always contains the address of a valid string.

An important advantage of this extended block entry mechanism is that the overheads are less than that of the parameter passing. Parameters must be stacked, and the set-up entails a space penalty on each procedure call. In many cases, procedures are written without using any non-locals, especially general library routines. Only the variables which change from call to call need be placed in the parameter list, the rest can be handled via the block mechanism even if there is a clash of identifiers. This would require a small semantic change to ALGOL 60 whereby array bounds and these new constructs would be regarded as being in the scope of the outer block. This would remove their existing anomalous position whereby they are regarded as being in the current block but cannot use local variables (which is quite different from parameters where the handling of identifier clashes is explicitly catered for in the Report).

12.3 Language Uniformity

In the author's view language design should go hand in hand with its implementation. Any syntactic ambiguities should be discovered in the conversion of the syntax of the formal definition into a form suitable for an automatic parser. Semantic ambiguities are less likely to be found by an automatic process, but the necessities of coding a compiler may bring them to light.

Probably the most important aspect is that of the uniformity of the language structure, any inconsistencies may cause implementation difficulties. Even with the best language and compilers, exceptional cases will arise causing more effort than might be expected. If the particular coding of one language construct outweighs that of the major language features, then it is probable that redefinition is called for. In very many cases in ALGOL 60, exceptional coding by the compiler is necessary to achieve acceptable code generation. A good example is **step** 1. Clearly a language change to allow this to be handled directly (by, say omitting **step** 1) would help both the compiler writer and user (as in FORTRAN and ALGOL 68).

One technique used for both implementation and formal definition is to use an abstract machine as the target language for the compiler. Having defined such a machine and written the compiler a detailed analysis is possible. Can one write useful sequences of abstract machine instructions which cannot be generated by the compiler? Of course, in many cases no high-level language construct could be devised to correspond to the required low-level form. As an example, the while statement form is easily written in terms of the abstract machine of Randell and Russell. For **while** <be> **do** <s> becomes

```
L1:
        <be>
        If False Jump       L2
        <s>
        Unconditional Jump  L1
L2:
```

So this, combined with the ease of implementation and the usefulness of the extension are powerful reasons for thinking such an extension is worthwhile. Using the low-level machine one can attempt to assess the high-level language. Particular tasks can be coded at both levels, and converted to the low level form for direct comparison.

One must be careful that any decisions are not unduly influenced by the particular abstract machine in use. For instance, the Whetstone machine contains a number of features which are quite unlike any realistic non-interpretive scheme. These are:

1. Dynamic type checking.
2. All parameters checked at run-time (never at compile time).
3. General and complex **for** loop mechanism.

Nevertheless, several mechanisms exist in the system which are available for one purpose but which have a wider connotation. As an example, in order to preserve the address of an array element while the right hand side of an assignment statement is evaluated, a mechanism exists for calculating addresses. A low level programmer would obviously exploit this to avoid the recalculation of array addresses in a wider context. This is true of all ALGOL implementations including the B5500, so the high level language designer ought to consider how the address mechanism can be made more widely available. In a similar vein, the low level language programmer can write the equivalent of

if <be> **then** x **else** $a[1] := b$

which is

```
      <be>
      If False Jump            L1
      Take Real Address        x
      Unconditional Jump       L2
L1:   Take Real Address        a
      Take Integer Constant    1
      INDex Address
L2:   Take Real Result         b
      STore
```

It is not so easy to compile this into good code for many machines, since it is most natural to calculate the address after the right hand side. In fact **if** b **then** x **else** $y := z$ is likely to be less efficient due to use of indirect addressing than **if** b **then** $x := z$ **else** $y := z$. This form can be extended in a similar manner using the case mechanism, as in ALGOL 68 and BCPL.

A more radical departure can be envisaged for the low level equivalent of

```
sum := 0;
for i := 1 step 1 until n do
        sum := sum + a[i]
```

In the Whetstone machine, by hand coding, the variable sum would be kept in the stack. The high level language version of this would presumably be

$sum := 0$ **for** $i := 1$ **step** 1 **until** n **do** $(+a[i])$

This extension would require a major revision to the syntax of ALGOL, but the advantages are clear. More efficient code generation is possible without resorting to complex optimisation. Moreover, the fact that three references to sum are replaced by one means that a programmer following the source text can more easily appreciate the significance of sum. One must, however, draw a careful distinction between the power of expression in a language and the machine capability. One can argue that an apparently simple language structure (an operator, for instance) should not invoke any complex mechanism. An ordinary high-level language programmer using something like APL could easily be misled into assuming his program is efficient because its inner loop is one line of source text. This is one of the defects of call-by-name and perhaps of a parameterless function call in ALGOL.

In order to justify a language design more is needed than simply relying on the ideas, prejudices and hopes of the designer. The low level-high level technique above can be combined with an information-theory approach to measure, in absolute terms, the power of the language. Of fundamental importance too is the degree to which the language itself is foolproof. Minor punching errors should, if possible, result in an invalid program rather than an apparently correct one. For example, seven years after its production, an error was found in the exponentiate routine of the Whetstone translator due to a typing error. This would not have been possible had the assembler used mnemonics for store locations with the appropriate symbol-table checks.

12.4 Language Definition

The publication of the ALGOL 60 Report set new standards for language definition. Never again was there any technical difficulty in producing an unambiguous and complete syntax, all the major problems with ALGOL were semantic in nature. A semantic definition can be too tight—the fact that the exact properties of real arithmetic were left open in a substantial virtue because it allows efficient implementation on different hardware without presenting enormous problems with program transferability.

It is now possible to provide not only a syntax definition but a processor for a syntax check of the language. To this can be added a simple code generator and interpreter giving a complete compiling system. This is so with Euler (Wirth, 1966b) where the compiler can be regarded as a very rigorous definition of the language. If the interpreter is written in a subset of ALGOL 60, then the non-precision of real arithmetic is passed on to the new language. This was the method used for the implementation and definition of Babel (Scowen, 1969). Of course, a compiler will contain many aspects which cannot be regarded as part of the language definition, for instance, the method of handling errors, syntax recognition algorithm, etc. In the author's view, all languages of note should be defined this way, since the process of implementing the language definition (i.e. the compiler) will act as a check upon the consistency and compilability of the language constructs. An interesting approach of this type is given by Richards (1971) in the BCPL compiler. There is a compiler written in BCPL which generates a simple machine code called OCODE. To produce a compiler all that is necessary is an OCODE interpreter. The system can then be made efficient by making an OCODE to machine-code translator. Passing the compiler through this translator generates a more efficient version of the compiler. Further improvement can be made by altering the code generation parts of the compiler to take advantage of the machine structure that may be lost in the OCODE generation. Self-compiling compilers, which are now becoming very common (see 12.8), do not necessarily solve any problems of language definition, since even with high-level languages, the program can be machine dependent. The two level approach seems the best solution, and is the technique used by the Vienna Laboratory in the formidable task of defining PL/I (Lucas, 1971).

To attempt a formal language definition after an informal one is in wide use is bound to be difficult. This can readily be seen from the ANSI FORTRAN standard (American Standards Association 1964) and the PL/I Vienna definition. Elaboration of the FORTRAN standard has lead to a discussion of several hundred words on the meaning of a blank card. The FORTRAN standard contains many features which should be noted by programmers and designers alike. In very many cases, the values of particular variables are left undefined. However, without resorting to interpretive execution it would be impossible to show that programs do not reference such undefined variables. The reason for this vagueness is sound—the compiler writer is being given an option over the mechanism used in certain constructs (mainly parameters and DO loops). For the programmer to exploit such features could be called "sinful" rather than invalid since language processors could not expect to trap such actions. A good language design reduces the area of sinful programming without unduly constraining the programmer. The block structure of ALGOL constrains **goto**s to prevent sinful programming without restricting the programmer more than necessary. Since the block structure is largely implemented in the compiler rather than the running program this is a very powerful technique. The more or less free access to addresses that the FORTRAN parameter mechanism provides, gives enormous scope to sinful programming. In fact BCPL has the same difficulty although the instances where the image is defined are more obvious. Any language without type-checking is likely to have sinful features because the implicit conversion performed by the computer hardware will be machine dependent (e.g. addresses and integers, integers and characters).

Some of these issues are illustrated by performance tests designed at Oxford University. Programs were chosen at random for their workload as a benchmark. About 40% failed at compilation for both ALGOL and FORTRAN. However, although a high percentage of the ALGOL programs failed on execution none of those in FORTRAN entered the error routine. The difference is not that the FORTRAN programmers were more experienced, but that there is less validation in the FORTRAN run-time system (for instance, there was no bound checking). More could be accomplished by tightening language definitions to ensure that fewer errors can be made by the programmer. One technique is the **assert** statement of ALGOL W, which is of the form **assert** <Boolean expression>, and is equivalent to **if** ¬ <Boolean expression> **then** *fail*. Such statements can be compiled or not according to an option, although they are always compiled in the ALGOL W system.

In less than 20 pages, the Revised Report gave a definition of ALGOL 60 which has withstood the test of time. It has many defects, but in the author's opinion few reports have equalled it in precision, clarity and conciseness.

12.5 Data Structures

Undoubtedly one of the most severe drawbacks of ALGOL 60 in relation to its successors is the lack of data structures—apart from arrays. There are numerous design problems in providing adequate structuring facilities if they are to compete with the flexibility of machine code techniques.

The basic requirement is that several different typed "fields" are grouped together to form a "record" or a "structure" (Hoare, 1968). Unlike arrays, the fields are references by a field name rather than a subscript. A very simple example would be a structure consisting of two reals called “complex”. The two fields might be called r and i, so that the two parts of complex z would be $z.r$ and $z.i$. A compiler would normally assign two consecutive locations to the parts of z. Procedures can then be defined to multiply two complex numbers and deliver a complex result. Obviously, the introduction of complex as a separate data type would almost certainly be more efficient and convenient since infix operators could be used. ALGOL 68 allows user-defined infix operators, so this convenience could be provided, although the implementation overheads are likely to be similar to a procedure call.

Languages differ significantly in the methods that must be used to allocate storage for structures. In ALGOL W, a record is assigned storage dynamically in a manner which cannot necessarily be accommodated on a stack. Hence garbage collection is necessary if stack or record storage overflow the available space. In fact, each record type is allocated 1K word blocks of store until storage is exhausted (causing the garbage collector to be invoked). This allows very small overheads on allocation of records whose size is much less than 1K words. In SIMULA-67, structures are also allocated dynamically, but mechanisms for dynamic allocation and garbage collection are already necessary because of the simulation requirements. In Babel (Scowen, 1969), records are allocated space like arrays, in second order working store. On entry to a block, the record declaration gives the number required, so that storage can be assigned immediately. No competition between different records for core-store is allowed—something which in ALGOL W could invoke the garbage collector quite frequently.

In ALGOL 68, a structure is just another data type, in addition to the basic ones. Therefore a single structure involving only simple types, would be allocated space in first order working store. This is an advantage since many simple uses of structures need not invoke any complex storage allocation method (i.e. just a stack). Unfortunately many complex constructs in ALGOL 68 involve the “heap”—i.e. garbage collection. Moreover, although this is necessary with a flexible array (an array which has at least one bound determined dynamically), other constructs involve the heap in a complex manner which could be implementation dependent. In ALGOL 68-R Currie has taken considerable care to ensure that programs which do not involve the heap pay no penalty. For instance, with only a stack, it is not necessary to check for storage overflow on every procedure call (the machine’s limit register does the check). If garbage collection is necessary, then any language must ensure that addresses are tightly controlled so that the appropriate adjustments can be made. ALGOL 68 cannot prohibit the assignment of addresses out of the scope of the referenced entity. This is a design error, since scope is given for sinful programming, but experience with ALGOL 68-R has not shown this to be a practical problem.

Arrays in ALGOL 68 can be “sliced”, i.e. subsets formed of its elements to form another array. A consequence is that with array parameters $a[i]$ and $a[i + 1]$ are not necessarily consecutive in the store. Hence n rather than $n - 1$ multiplications are necessary to access an n dimensional array element. This could be a severe penalty with (say) a matrix multiply routine. The same facility (and difficulties) arise in ALGOL W, although the slicing mechanism is less general.

It is important to note that structures can be very efficient in many applications. Satterthwaite reports that a routine to calculate primes was faster when the previously calculated primes were stored in a list of records rather than an array. The reason for this is that ALGOL W does not optimise array access, so the overheads on moving down an array are more than a single indirection to obtain the next record of a list. Once the address of a record is known, all its fields should be capable of being accessed in one instruction. The implementor has more freedom to layout the fields within a record than can be permitted with an array so that advantages can be taken of special instructions to access part-words.

Several problems arise in the design of good record or structure handling facilities. Firstly, the layout of the fields within a record causes packing problems because of the dissimilar space requirements. This is more acute if a wide variety of data types are available, and if the hardware imposes addressing limitations. The necessity to align fields on word or double-word boundaries can mean either a significant loss of space or a large time and code-space penalty on access. Thus, to write effective programs, the user may be faced with many rules concerning record layout, which seem meaningless to him. On the other hand, the low-level programmer is very aware of how much he can gain by making a flag in a record the sign bit. Obviously some records are space and other time critical, but it may be rather hard for the compiler to determine any such criteria for field layout. In any case, variation of the layout may well be precluded by independent compilation and the preservation of records on backing store.

The major requirement for record handling is with the maintenance of file systems giving rise to numerous problems. Firstly, many fields cannot logically reside on backing store in a similar manner to that of main storage. A reference to another record in core can simply be an indirect address, whereas on backing store a file-address must be used, and checks must be performed to ensure the validity of this. Since inter-record cross references are often the *raison d'être* of a file system an adequate solution to this is of paramount importance. Secondly the field structure itself can be very complex, precluding static type-checking and making the representation on backing store far from obvious. One UK Government Department has a “file” stretching over 100 magnetic tapes, the record format of which is too complex for anyone program module to decipher completely. It is difficult to believe that any language can elegantly and concisely handle such file systems. One method of permitting use of high-level languages in such an environment is to have machine-code routines for file access which convert to a standard COBOL or PL/I internal form for processing. To produce an effective procedure entry mechanism one must check parameter specifications at compile time. This is not really possible once record handling is available. Input-output routines, and other general purpose utilities must be usable with a wide class of different record structures. A rigid adherence to complete specification of parameters would mean that such routines could not be written in a high-level language.

12.6 Input-Output

As noted in 12.1.4, the lack of any input-output system for ALGOL 60, has severely hampered the portability of programs, but over 90% of algorithms contain no input-output, and even those which do, have so little that no detailed comments are required. Nevertheless, a “standard” input-output scheme for ALGOL 60, widely implemented and accepted, would have obvious advantages. Two schemes have been proposed, the IFIP (1964b) set of procedures, and those produced by a committee under Knuth (1964a).

12.6.1 THE IFIP INPUT-OUTPUT SCHEME

These proposals consist of seven basic procedures whose action is defined in words, together with three further which are defined in terms of the basic procedures. The ten cannot be regarded as a complete input-output scheme since no format control is provided. They do provide a primitive scheme, from which more elaborate functions could be developed. A detailed examination of the proposals is worthwhile because it brings to light defects in the use of ALGOL for input-output and also in this particular scheme.

As noted by Wichmann (1972), the proposal contains two obvious typographical errors—both in the formal ALGOL 60 text. The method of handling input is by means of a primitive procedure in *insymbol* which has three parameters, a channel number, a string and a destination. The destination is a name parameter to which an assignment is made of the "code" of the next basic symbol on the input channel. The "code" used is determined by the string—the nth basic symbol of the string is given the code n. Non-basic symbols are given a negative code. This technique has several disadvantages. Firstly it virtually demands that strings are stored internally as ALGOL basic symbols. This is often not the case, and storing as characters has the substantial advantage that the running program need not perform character conversions (see 1.9.4). Secondly, the important format-effector characters are not ALGOL basic symbols so they are assigned negative values which are implementation dependent. This means that it would be very difficult to write algorithms entirely in terms of these procedures alone.

The basic input routine for real numbers assigns to a name parameter the next number on the input tape. The action in terms of basic symbols is undefined—a virtue in some respects as it can be trivially implemented in terms of available routines. However, transferability of data is not necessarily possible, although it could have been stated that all numbers conforming to the syntax of a number as in the Report should be acceptable. The difficulty with this approach is that many systems use fixed format for input, where both invisible field separators or multiple spaces may be delimiters. It is stated that the real number input and output procedures should be exact converses—this is a tight requirement since it demands that more decimal digits should be output than there are binary places available internally so that round- off can be avoided. This also makes the field length implementation dependent.

Name parameter assignment used by the procedure *inreal* poses another problem. As noted in 1.10.3 many implementations would demand that the actual parameter was a real variable. In consequence, to read an integer one must write

$$inreal(channel, x);$$
$$i := x;$$

Admittedly this would act as a reminder that 3.4 on the data tape would be read as 3, but the construction is not very clear. The obvious solution would be to have written the procedures as functions, but the author assumes that side effects were not regarded as "proper" by WG2.1 of IFIP.

Defined as primitive procedures are *inarray* and *outarray* for real arrays each element being handled in the identical manner to *inreal* and *outreal*. However, one cannot in ALGOL 60 write the array procedures in terms of the more basic procedures because there is no mechanism for handling arrays of unknown dimensionality, nor can one find the lower and upper bounds of any subscript. In fact, in some implementations, not all this information is available. For instance, if the dope vector technique is used, with only a final check on the core range, roughly half the array information will be lost ($n + 1$ numbers for an n-dimensional array). On KDF9 the situation is slightly worse in that although $n + 1$ numbers are preserved it is impossible to find the dimensionality of the array. In fact the basic array information consists of three numbers, the address of the first and last element, and the address of $a[0, \ldots 0]$. However, the order is defined—essentially by rows. This means that implementation on KDF9 is impossible with multi-dimensional arrays. As arrays are stored by columns, and insufficient information is available to remap the array, one cannot determine where successive elements should be placed.

The three additional procedures are *outboolean*, *outstring* and *ininteger*. The first two are trivial, but the third is certainly not and contains a number of deficiencies. If a non-ALGOL basic symbol is read by *ininteger*, then its internal code value is used to construct the integer! Also various invalid numbers are not detected as errors, for instance, multiple signs, delimiter not a comma, etc. Presumably these procedures should be taken as illustrative rather than definitive.

12.6.2 THE ACM PROPOSALS

In May 1964, a committee of the ACM under the chairmanship of D. E. Knuth proposed an extensive input-output scheme for ALGOL 60. It is quite unlike the IFIP proposals in that it represents a complete scheme giving all the facilities one could reasonably expect of an input-output system. The proposals contain many interesting points, mostly relating to fundamental difficulties in implementing input-output in ALGOL 60.

All control is maintained of both input and output via format strings, vaguely resembling those of FORTRAN. It is suggested that these strings are checked by the compiler for validity and transcribed into a convenient form for the run-time system. By this means, much of the work of the scheme can be performed by the compiler without using non-standard features of ALGOL 60. This is important, because on KDF9 as much as half the execution time of small scientific programs can be handling input-output as this is left entirely to the run-time system. It is stated that the effect of input-output using strings, consisting of ALGOL basic symbols not representable as one character, is undefined. This avoids the difficulties in the IPIP scheme, although some machine independence is lost. It may not be possible with some systems to avoid multi-character symbols because $_{10}$ is required in numbers and is not always an available character. The ACM proposals would appear to allow the use of E for $_{10}$ but since this cannot be permitted in program text, it does not seem a good idea.

One mistake in the scheme is that the characters used to control the format are in upper case whereas the procedure names (as well as the IFIP names, standard functions, etc.) are in lower case. Also a variation is allowed in one format control character (alignment mark is $\uparrow$ or $\times$) which presumably means that both must be catered for to give machine independence.

The fact that ALGOL 60 does not allow procedures to have a variable number of parameters has been circumvented by two techniques. Firstly, ten procedures are specified in some cases instead of one, for instance *format0, format1, . . . format9*. Secondly, a subtle use of formal procedures allows the user to specify a set of values rather than just one. Neither of these methods can be regarded as very satisfactory, but there is no viable alternative within the confines of ALGOL 60. It is interesting to note that the CDC, in implementing a version of these proposals for the 6000 series, has only one procedure *format* which has an arbitrary number of parameters. Although this is convenient, it is not ALGOL 60 hence conversion to another system is not necessarily possible.

The main objection to the system is its complexity. The format strings have the same defects as FORTRAN in that a large number of *ad hoc* rules must be mastered to use the scheme properly. On KDF9, format strings of a much less complex form are used, but many programmers—especially those who do a small amount of programming—would prefer a more basic scheme. On Atlas, formats are expressed by two parameters, the number of places before and after the point, which is adequate for most purposes and easy to use. The other objection to the complexity is the implementation effort required and the size of programs using this scheme. Apart from the format string analysis and interpretation elaborate rules are given as to the actions to be taken on various error conditions. These include not splitting an overlength line in the middle of a number (requiring a line buffer), and transfer of control to user-defined procedures in other circumstances. The difficulty here is to check that the user-defined procedure is in scope.

To summarise, the ACM proposals represent a complete, practical scheme for input-output, but a complex one entailing a significant implementation effort.

12.6.3 OTHER SCHEMES

Two different approaches have been taken with input-output. Some are strict ALGOL 60, using the procedure mechanism as the implementation vehicle. Others have used language extensions, often based upon the FORTRAN system. The language extension technique has the advantage that variable parameters and parameters of different types can be handled with ease. The Elliott ALGOL system has a particularly convenient form of extension—the **print** statement which prints strings as typed, converts integers to the current integer format, and converts reals to the current real format. The formats are set by procedures which can apply to just one print statement or to all subsequent printing.

The management of channel numbers is usually by one of two methods. Either by specifying the channel on each call (as in FORTRAN, IFIP, ACM, KDF9) or by having a current input and output setting. The author prefers the latter on two grounds. Firstly it is more efficient because fewer parameters are passed and the basic routines know which device is current, and secondly because users tend to give explicit numbers to the channels making any reassignment awkward.

On KDF9, format strings are used, but repeated conversion is avoided by a two stage process. The format string is validated and converted into a bit pattern (actually an ALGOL integer) by a procedure. The programmer can then use the integer to control subsequent write statements. The formats are very general, allowing precision to be set independently of the number of digits output. Input is free format via a real function. If an attempt is made to output a number with an inconsistent format, a full accuracy number is printed on a separate line. One defect is that one cannot specify excess digits over the machine accuracy of 12 digits—hence one cannot output a number and read it back as the identical bit pattern.

12.7 Possible Extensions to ALGOL 60

Many ALGOL systems have extensions, not all of which can be made without minor alterations to the basic language. For instance, the introduction of data type character is likely to require that strings are regarded as characters rather than as ALGOL basic symbols. The introduction of type complex has some rather interesting consequences. Difficulties arise with standard functions such as (say) *sqrt*. If type complex exists should $sqrt(-1)$ yield i rather than fail? Or should it yield i if the argument is complex? In this case, since the algorithm depends upon the type of the argument, the function cannot be implemented in ALGOL 60. This is in fact so with some systems which implement *abs* differently for real and integer arguments. To specify function actions which cannot be written in the language is to admit an inadequacy in the language. The existence of extra data types considerably complicates the language structure. In particular, the implicit type conversions of ALGOL 60 become much less defensible. This problem is illustrated in extreme form with PL/I where a user may find a conversion taking place implicitly to a data type he did not know existed. In Babel, Scowen allows no implicit conversions except for constants. Although this is a good discipline on the user—after all, these conversions are often expensive in time and space—it is difficult to see how this could be accepted for a major language. Some of the conversions in ALGOL 68 seem very sensible, for instance, conversion from integer to real to complex has the ordinary meaning and is implicit whereas conversion in the opposite direction (a loss of information) must be explicit. However, the conversions involving references and procedures are much less obvious and could easily lead to errors. In fact, it is not always easy to see when a parameterless procedure is being called. Most of these difficulties stem from what the author regards as too free a use of addresses. Similar troubles arise in BCPL and BLISS, although here the underlying structure is much simpler and so less error prone.

The author finds it surprising that so few high-level languages have features present in most assemblers, but not concerned with their low-level nature. For instance, the majority of modern assemblers allow the declaration of named constants (EQU). These are called manifest constants in BCPL. The implementation is straightforward, and, of course, it is very efficient since it allows the literal forms of instructions to be used (which are often substantially faster and more compact). Moreover it can have an amazing effect on program intelligibility and portability. Similarly, “dynamic” arrays could be added to FORTRAN—although recompilation would be needed. Because of this lack in FORTRAN, many programs appear to have arrays larger than necessary merely to accommodate a possible requirement. In a language designed by the author based upon PL360 (Wirth, 1968), named constants proved to be much more heavily used than was anticipated (see 12.8). In addition, many assemblers will calculate complete expressions at compile-time. This is not so easy to extend to a high-level language since it is less obvious that the calculation can be performed at compile-time. The ALGOL 68 **value** mechanism is not the same as named constants since the compiler needs to optimise the declaration in order not to generate an ordinary variable which is initialised on block entry, but is nevertheless read-only. Surely it is preferable for the user to specify a compile-time operation and not depend upon an optimisation that mayor may not be performed.

Another major facility present in many assemblers is that of conditional compilation. A frequent requirement of system programs is to be able to generate different versions according to certain compilation options. During this process, one wants to be able to remove any code that cannot be executed, giving a minimal sized program. With high-level language programs such options are usually restricted to diagnostic and testing facilities, and do not affect the program as a whole. There is a major advantage in this facility in the maintenance of computer software. Nothing is more error-prone than having six slightly different versions of a program to modify. Admittedly, this can also be overcome by a macrogenerator, but this involves an extra pass, and cannot relate to the language structure.

Several assemblers have a special macro-expansion system designed to work within the assembler, but logically independent from it. Obviously, if the macro-generation is completely separate, then a separate pass can be used and no problems arise. However, it is convenient, both from the programmer and compiler-writer's viewpoint to regard macro-definitions as "declarations". They then become scoped, and their position in relation to the rest of the language is clear. But a macro-expansion could involve more **end**s than **begin**s and so generate text which is out of scope. This is more of a theoretical than a practical point but shows that free text and a syntax structured language cannot easily be reconciled. Indeed, the whole virtue of a macro-generator is that textual forms, not necessarily expressible in syntactic notation, can be handled—otherwise a language extension might be more appropriate.

The compiling difficulties of handling macro-expansions are mainly in the macro-call. In the B5500 system, for instance, any identifier could be a macro-name for which the appropriate text must be substituted. Such an identifier may appear in a context in which an identifier is not expected by the ordinary syntax rules—as will be the case if the substituted text does not start with an identifier. Hence expansion must be by the basic input routine, which must work in terms of identifiers rather than basic symbols. Obvious trouble-spots are how comments and strings are handled, since they are not necessarily processed in terms of identifiers. The B5500 compiler makes a prescan of each line, locates basic symbols (which are reserved words), and makes a partial analysis to allow convenient macro-expansion. Conceptually, it seems elegant to have macros with parameters having the same form as procedure calls. However as completely different implementation mechanisms are involved, and as the context is not restricted in the same way as a procedure call, there is no necessity for this.

One inadequacy of ALGOL 60 is that there is no convenient mechanism for array initialisation. Of course, it is possible to provide such a mechanism, for instance **array** $a[1 : 3](-1, 0, 3.4)$ might be regarded as the array declaration followed by $a[1] := -1; a[2] := 0; a[3] := 3.4$; (included as statements in the appropriate position). Most languages that provide such a facility allow any outer block variable to appear in the array values. This implies that the implementation is barely more efficient than the hand-coding in strict ALGOL 60, because explicit calculations and assignment must be made for each variable. In practice, such values are usually calculable at compile-time, so that the array declaration need only consist of a copy operation—or if the array is "read only" merely a setting-up of the dope vector to point to the fixed values. This then gives an implementation which is of comparable efficiency to assembly code. One is therefore faced with provision of a general facility which ought to be optimised or restricting array values to expressions which can be calculated at compile-time.

One of the defects of most existing programming languages is that array handling is not direct but is achieved by implicit actions on each element. In the ALPHA system, a dummy subscript notation is used to imply a repeated execution of the statement over the scope of the subscript concerned. To be effective, this also requires optimisation because naive expansion produces a for loop for each statement. Another difficulty is that the semantics are not clear when more than one subscript is dummy—some identification ought to be given. There are two distinct approaches to the problem. One can improve the conventional looping and access mechanisms, or one can introduce an array as a single entity with its own operators, etc. The second radical approach is adopted by APL, which is vital to make effective use of array processing machines like CDC's STAR.

The looping mechanism can be enhanced in many ways. It has already been noted that the multiple **for** list element, and the sign of the **step** expression do not give a good loop control in ALGOL 60. These points were overcome in ALGOL W without a radical departure. In PASCAL, Wirth introduces a bound pair as a concept (called a range) which allows more effective array access. A $range[1 : 10]$ say, is a set of ten numbers (it could be a set of colours or countries). Primitive operations are provided to step through these sets, which allow the bound pair and the looping to be unified. Converting these concepts to an ALGOL 60 context, one might have

```
$b =$ **bound pair** $1 : 10$;
**array** $a[b]$;
...
    **for** $b$ **do**
        $a[b]$
```

It is conceivable that the compiler could check that access to $a[b]$ cannot involve a subscript violation, or print a warning if an array element is accessed with a bound pair element which is not identical to the one used for the array declaration. One could even leave it undefined as to whether the array was scanned forwards or backwards (if adequate restrictions were made). In any case, a suitable internal representation for the subscript could be used (say multiplied by 4 on 360 to give word addressing). This would allow better array access, but shifts would be required if b were to be used as an ordinary integer.

A point made in an early paper of Woodger (1961) is that ALGOL 60 is rather weak in semantics. Taking integers as an example, the vast majority are used as index values and markers whose values are naturally restricted (to a machine dependent address length). A compiler-writer cannot safely pack such integers because they may be used in a random number generator (say) where a much larger value would be required. There is a considerable advantage therefore, in having declarations which restrict the use of integers or other types so that a compiler-writer can take a more intelligent approach on particular machines. Similar reasoning applies to length (or rather accuracy) specification for real variables. Double length on 360 is quite different from that on the CDC 6000 series, which implies that machine-independent programs cannot be transferred in a straightforward manner. ALGOL 68 allows **long real** to be the same as **real** if the implementor so wishes (on the 6600?), but perhaps it would be more appropriate for such decisions to be made by compilation directives.

The effective handling of characters on most computers can only be achieved by packing several to a word. The ACM input-output proposals concede this by permitting a machine dependent feature to exploit packing. In PASCAL, Wirth allows a machine-dependent number of characters to be packed into a word, this being achieved by some standard procedures (which could be implemented by open code). The provision of more extensive string operations as, say in XPL (McKeeman, 1970), depends upon having a storage allocation scheme (other than a stack) to support this. In fact the majority of such requirements can be met by packed words and linked lists.

One feature can be managed with strings where only stack storage is available, although the author knows of no such implementation. This is the ability to read a string as data on block entry, so that one should be able to write

```
**string** $s :=$ **read**
```

meaning that the variable length string is to be read from the input media and assigned an amount of second order working store depending on the size of the string. ALGOL 68 is rather deficient in this respect as strings are regarded as flexible arrays implying garbage collection. This derivation from the very static form of strings in ALGOL 60 is obviously a programming pitfall.

Apart from array element access, the most important feature of ALGOL 60 whose implementation is poor relative to machine coding is procedure calls. About 20 to 30 instructions plus two per parameter are involved, whereas in most assembly code programming only 3 or 4 are required. Of course, this “inefficiency” reflects that the generality of the ALGOL 60 mechanism exceeds the requirements of many applications. One can therefore envisage two levels of procedures, one of which is less general but more efficient. For instance, in Babel, Scowen allows named statements to be called like parameterless procedures. To allow the implementation to just jump and stack the return address implies that no environment control must be involved. Hence either the simple procedure body must not contain a block or a restriction prohibiting recursion of such procedures must be made. The first seems more natural to ALGOL. Although such an informal procedure mechanism is usually used to implement for loops with multiple for lists, unfortunately the natural generalisation is not available to the programmer. It would be much better to have the mechanism in the form of simple procedures but without such complex **for** loops. Then the user could use these procedures to code the complex for loops if these are required. The reluctance to introduce two procedure mechanisms is presumably due to less concern for efficiency than for language elegance.

12.8 Systems Programming Languages

An important area of considerable current interest is the design and implementation of languages tailored to the requirements of systems programming. Many of the advantages of ALGOL 60 are notational. If a language can be designed to look superficially like ALGOL 60, and yet allow the same degree of control over a machine as a conventional assembler, then the result will be an effective systems programming language. The current interest in these languages probably stems from PL360, a language designed by Wirth (1968). However many earlier languages share the same aims notably MAD, ESPOL (for the B5500), CORAL and BCPL.

The features of ALGOL 60 to be found in such languages are invariably:-

- compound statements
- conditional expressions and statements
- a looping mechanism similar to the **for** loop
- declarations and type-checking
- **goto** is rarely necessary.

Other features of ALGOL are omitted, namely

- call by name
- multiple **for** list elements
- elaborate parameter handling
- most aspects of dynamic storage allocation.

Such languages show considerable variation. For instance, PL360 is fairly basic, it has no parameter mechanism, registers are not used without explicit user directions, and in no case are instructions generated by the compiler which are at all non-obvious. On the other hand BLISS (Wulf, 1971), has dynamic storage allocation with many of the complexities of ALGOL 60. A rich parameter mechanism and access methods are provided by allowing programmer access to addresses.

Such languages can be designed for mini-computers. For instance, the author was responsible for one on a DDP516, a 16 bit computer. The code and fixed tables only required just over 4K words. The compiler was written in its own language (almost always the case for such systems), and it can compile itself in 15 seconds of processor time. Experience with this system has led the author to conclude that this method is by far the most cost-effective method for implementing system software. The disadvantages are very few. Code size is usually about 5% bigger than the equivalent hand-coding. This is because unnecessary jumps are generated, caused by the positioning of code in textual order. The advantages far outweigh this. In the author's experience, traditional assembly code programs are often inefficient simply because they are too difficult to restructure. A proper subroutine structure, together with implicit flow control means that many changes can be made at a glance, whereas with an assembly code program one may well have to rely on comments (unchecked by the compiler!). The advantage of such a language was illustrated by finding a programmer transcribing an assembly code program into PL5l6 in order to understand the algorithm.

The importance of these developments is not only that high-level languages can replace traditional assembly languages in an effective manner, but also that renewed interest is being shown in the design and implementation of such languages. PL360 was implemented in three man-months, and PL5l6 in about nine man-months (including writing the manual), so the manufacture of such a system can reasonably be part of any major software project. In Europe, most of these languages are based upon ALGOL 60, although an increasing number are being influenced by ALGOL 68. In North America, the majority are based upon PL/I, including one which is supposed to be used by IBM. This development means that no programmer should have to write software in ordinary assembly languages. This is very important, since the development of high-level languages has been considerably hampered by the fact that a significant fraction of the industry did not use them. All too often machine-code programmers were writing parts of a high-level language compiler for a system they would never use. Use of a language or system by its designers is by far the most effective method of ensuring its rapid development. For this reason most systems programming languages are self-compiling, often by bootstrapping from ALGOL 60 or PL/I.

Bibliography

ADRAHAMS, P. W. (1966). A final solution to the dangling **else** of ALGOL 60 and related languages. *Comm. A.C.M.*, **9**, 9, pp. 679-682.

A.C.M. (1971). Symposium on Languages for Systems Implementation. SIGPLAN Notices, **6**, 9.

A.F.S.C. (1970). User's Manual, COBOL compiler validation system. Directorate of systems design and development, HQ Electronic Systems Division (A.F.S.C.). L. G. Hanscom Field, Bedford, Massachusetts.

AKKA, D, S. (1967). A quantitative comparison of efficiencies of compilers. MSc Thesis, Victoria University of Manchester.

ALLEN, F. E. (1971). A basis for program optimization, Proc. I.F.I.P. Congress 1971, pp. 385-390. North Holland, Amsterdam.

American Standards Association X3 Committee (1964). FORTRAN vs Basic FORTRAN, *Comm. A.C.M.*, **7**, 10, pp. 591-625.

BAUER, H., BECKER S. AND GRAHAM, S. (1968). ALGOL W Implementation. Stanford University Computer Science Department Report CS98.

BAYER, R., GRIES, D., PAUL, M. AND WIEHLE, H. R. (1967). The ALCOR Illinois 7090/7094 post mortem dump. *Comm. A.C.M.*, **10**, 12, pp. 804-808.

BELL, D. A. AND WICHMANN, B. A. (1971). An ALGOL-like assembly language for a small computer. *Software—Practice and Experience*, **1**, 1, pp. 61-72.

BOON, C. (Ed.) (1971). High level languages. Infotech State of the Art Report No.7. Infotech, Maidenhead, England.

BRAWN, B. S., GUSTAVSON, F. G. and Mankin, E. S. (1970). Sorting in a paging environment. *Comm. A.C.M.*, **13**, 5, pp. 483-494.

BRON, C. (1971). Association of virtual store in the THE multiprogramming system: The cost of flexibility. *In* "Operating System Techniques" (C. Hoare and R. Perrot, Eds.). Academic Press, London.

BROOKER, R. A., ROHL, J. S. and CLARK, S. R. (1966). The main features of Atlas Autocode. Camp. J., 8, pp. 303-310.

BROOKER, R. A., MORRIS, D. and ROHL, J. S. (1967). Experience with the Compiler Compiler. *Camp. J.*, **9**, 4, pp. 345-349.

BROOKER, R. A. (1970). Influence of high-level languages on computer design. *Proc. I.E.E.*, **117**, pp. 1219-1224.

BROWN, P. J. (1967). The ML/I macro processor. *Comm. A.C.M.*, **10**, 10, pp. 618-623.

BRYANT, P. (1968). FORTRAN—A comparative study. Atlas Computer Laboratory, Science Research Council Report.

BURNS, D., HAWKINS, E. N., JUDD, D. R. AND VENN, J. L. (1966). The Egdon system for the KDF9. *Camp. J.*, **8**, 4, pp. 297-302.

Central Computer Agency (1971). A comparison of computer speeds using mixes of instructions. Technical Support Unit Note 3806.

CERF, V. G. AND ESTRIN, G. (1971). Measurement of recursive programs. *Proc. I.F.I.P. Congress*, 1971, pp. 314-319. North-Holland, Amsterdam.

COHEN, D. J. AND GOTLIEB, C. C. (1970). A list structure form of grammars for syntactic analysis. *Computing Suraeys*, **2**, 1, pp. 65-82.

CURRIE, I. F. (1970). Working description of ALGOL 68-R. Royal Radar Establishment, Malvem, Wores. Memo 2660.

CURRIE, I. F., BoND, S. G. AND MORRISON, J. D. (1971). ALGOL 68-R, its implementation and use. *Proc. I.F.I.P. Congress*, 1971, pp. 360-363. North-Holland, Amsterdam.

DAHL, O. J. AND NYGAARD, K. (1966). SIMULA, an ALGOL-based simulation language. *Comm. A.C.M.*, **9**, 9, pp. 671-678.

DIJKSTRA, E. W. (1962). “A primer of ALGOL 60 Programming.” APIC Studies in Data Processing, No.2. Academic Press, London.

DIJKSTRA, E. W. (1963). An ALGOL 60 translator for the X1. “Annual Review in Automatic Programming,” pp. 329-356. Pergamon Press, Oxford.

DIJKSTRA, E. W. (1968). The structure of the THE Multiprogramming system. *Comm. A.C.M.*, **11**, 5, pp. 341-346.

ENGELER, E. (Ed.) (1971). “Symposium on Semantics of Algorithmic Languages.” Lecture Notes in Mathematics, Vol. 188. Springer-Verlag, Berlin.

EVANS, E. (1964). An ALGOL 60 Compiler. “Annual Review in Automatic Programming,” pp. 87-124. Pergamon Press, Oxford.

FELDMAN, J. AND GRIES, D. (1968). Translator writing systems. *Comm. A.C.M.*, **11**, 2, pp. 77-113.

FLOWERS, B. H. (1966). A report of a joint working group on computers for research. Her Majesty’s Stationery Office.

FLOYD, R. W. (1971). Towards interactive design of correct programs. *Proc. I.F.I.P. Congress* 1971, pp. 7-11. North-Holland, Amsterdam.

FOSTER, J. M. (1968). A syntax improving program. *Camp. J.*, **11**, 1, pp. 31-34.

FOSTER, J. M. (1970). “Automatic Syntactic Analysis.” Computer Monograph Series, Elsevier, New York.

GENUYS, F. (Ed.) (1968). “Programming Languages.” NATO Advanced Study Institute. Academic Press, London.

GRAU, A. A., HILL, U. AND LANGMAACK, H. (1967). Translation of ALGOL 60. “Handbook for Automatic Computation”, Vol 1b, Part B. Springer-Verlag, Berlin.

GRIES, D., PAUL M. AND WIEHLE, H. R. (1965). Some techniques used in the ALCOR-Illinois 7090 Compiler. *Comm. A.C.M.*, **8**, pp. 496-500.

GRIES,D. (1971). “Computer Construction for Digital Computers.” Wiley, New York.

GRIFFITHS, T. V. AND PlITRICK, S. R. (1965). On the relative efficiencies of Context-free grammar Recognizers. *Comm. A.C.M.*, **8**, 5, pp. 289-299.

HADDON, E. W. and PROLL, L. G. (1971). An ALGOL line-syntax checker. *Camp. J.*, **14**, 2, pp. 128-132.

HAUCK, E. A. AND DENT, B. A. (1968). Burroughs B65OO/B7500 stack mechanism. Spring Joint Comp. Conference. A.F.I.P.S., Vol. 32, pp. 245–252.

HEINHOLD, J. AND BAUER, F. L. (Eds.) (1972). “Fachbegriffe der Programmierungstechnik.” Ansgearbeitet vom Fachausschutz Programmieren der Gesellschaft für Angewandte Mathematik und Mechanik (GAMM), Oldenbourg, Munchen.

HENHAPL, W. AND JONES, C. B. (1971). A run-time mechanism for referencing variables. *Information Processing Letters*, **1**, 1, pp. 14-16.

HERRIOT, J. G. (1969). An ambiguity in the description of ALGOL 60. *Comm. A.C.M.*, **12**, 10, p. 581.

HIGMAN, B. (1967). “A comparative study of programming languages.” Computer Monograph Series. Elsevier, New York.

HOARE, C. A. R. (1964). The Elliott ALGOL programming system. *In* “Introduction to System Programming,” (P. Wegner, Ed.) pp. 156-165, APIC Studies in Data Processing, No.4. Academic Press, London.

HOARE, C. A. R. (1966). A note on indirect addressing. *ALGOL Bulletin*, AB 21.3.8.

HOARE, C. A. R. (1968). Record handling. In “Programming Languages,” (F. Genuys, Ed.), pp. 291-347. Academic Press, London.

HOARE, C. A. R. (1971). Procedures and parameters: An axiomatic approach. In “Symposium on Semantics of Algorithmic Languages”, (E. Engeler, Ed.) pp. 102–116. Springer-Verlag, Berlin.

HOPGOOD, F. R. A. AND BELL, A. G. (1967). The Atlas ALGOL preprocessor for non-standard dialects. *Camp. J.*, **9**, pp. 360-364.

HOPGOOD, F. R. A. (1968). A solution to the table overflow problem for hash tables. *Computer Bulletin*, **11**, 4, pp. 297-300.

HOPGOOD, F. R. A. (1969). “Compiling Techniques.” Computer Monograph Series. Elsevier, New York.

HUXTABLE, D. H. R. and HAWKINS, E. N. (1963). A multi-pass translation scheme for ALGOL 60. “Annual Review in Automatic Programming,” Vol. 3, pp. 163-205. Pergamon Press, Oxford.

HUXTABLE, D. H. R. (1964). On writing an optimising translator for ALGOL 60. “Introduction to System Programming,” APIC Studies in Data Processing, No.4, pp. 137-155, Academic Press, London.

I.C.T. (1965). I.C.T. Atlas 1 Computer programming manual for Atlas Basic Language (ABL).

I.F.I.P. (1964a). Report on Subset ALGOL 60 (I.F.I.P.). *Comm. A.C.M.*, **7**, 10, pp. 62 28.

I.F.I.P. (1964b). Report on Input-output procedures for ALGOL 60. *Comm. A.C.M.*, **7**, 10, pp. 628-630.

INGALLS, D. (1971). FETE, A FORTRAN Execution Time Estimator. Stanford Computer Science Department Report No 204.

INGERMAN, P. Z. (1961). Thunks—A way of compiling procedure statements with some comments on procedure declarations. *Comm. A.C.M.*, **4**, 1, pp. 55-58.

JENSEN, J. AND NAUR, P. (1961). An implementation of ALGOL 60 procedures. *BIT*, **1**, pp. 3847.

JENSEN, J. (1965). Generation of machine-code in an ALGOL compiler. *BIT*, **5**, pp. 235-245.

KILBURN, T., MORRIS, D., ROHL, J. S. AND SUMNER, F. H. (1968). A system design proposal. *Proc. IFIP Congress 1968*, pp. 306–811. North-Holland, Amsterdam.

KNUTH, D. E. AND MERNER, J. N. (1961). ALGOL 60 Confidential. *Comm. A.C.M.*, **4**, 6, pp. 268-272.

KNUTH, D. E. (1964a). A proposal for input-output conventions in ALGOL 60. *Comm. A.C.M.*, **7**, 5, pp. 273-283.

KNUTH, D. E. (1964b), Man or boy? *ALGOL Bulletin*, No 17, p. 7, Mathematische Centrum, Amsterdam.

KNUTH, D. E. (1967). The remaining trouble spots in ALGOL 60. *Comm. A.C.M.*, **10**, 10, pp. 611-618.

KNUTH, D. E. (1968). “The Art of Computer Programming,” Vol. 1, Fundamental Algorithms. Addison-Wesley, New York.

KNUTH, D. E. (1969). “The Art of Computer Programming,” Vol. 2, Semi-numerical Algorithms. Addison-Wesley.

KNUTH, D. E. (1971a). An empirical study of FORTRAN programs. *Software—Practice and Experience*, **1**, 2, pp. 105-133.

KNUTH, D. E. (I971b). Mathematical analysis of algorithms. *Proc. IFIP Congress 1971*, pp. 19-27. North Holland, Amsterdam.

KNUTH, D. E. (1971c). Top-down syntax analysis. *Acta Informatica*, **1**, 2, pp. 79-110.

KORENJAK, A. J. (1969). A practical method for constructing LR(k) processors. *Comm. A.C.M.*, **12**, 11, pp. 613-623.

LANG, C. A. (1969). SAL—Systems assembly language. Spring Joint Comp Conference. AFIPS, Vol. 34, pp. 543-555.

LAWNDE, W. R., LEE, E. S. AND HORNING, J. J. (1971). An LALR(k) parser generator. *Proc. I.F.I.P. Congress 1971*, pp. 513-518. North-Holland, Amsterdam.

LAUER, P. (1968). Formal definition of ALGOL 60. TR 25.088. IBM Laboratory, Vienna.

LONDON, R. L. (1970). Proving programs correct: Some techniques and examples. *BIT*, **10**, 2, pp. 168-182.

LONDON, R. L. (1971). Correctness of two compilers for a LISP subset. Stanford Computer Science Department Report CS 240.

LOWRY, E. S. and MEDLOCK, C. W. (1969). Object Code optimization. *Comm. A.C.M.*, **12**, 1, pp. 13-22.

LUCAS, P. (1971). Formal definition of programming languages and systems. *Proc. I.F.I.P. Congress 1971*, pp. 291-297. North-Holland, Amsterdam.

MALCOLM, M. A. (1971). Algorithms to reveal properties of floating-point arithmetic. Stanford Computer Science Department Report CS 211.

MCKEEMAN, W. M. (1965). Peephole optimisation., *Comm. A.C.M.*, **5**, 7, pp.443-444.

MCKEEMAN, W. M. (1967). Language directed computer design. Fall Joint Comp Conference. A.F.I.P.S., Vol. 31, pp. 413-418.

MCKEEMAN, W. M., HORNING, J. J. AND WORTMAN, D. B. (1970). “A Compiler Generator.” Prentice Hall, New York;

MILLS, H. D. (1970). Syntax oriented documentation for PL 360. *Comm. A.C.M.*, **13**, 4, pp. 216-222.

MILNER, R. (1968). String handling in ALGOL. *Comp J.*, **10**, pp. 321-324.

MOULTON, P. G. AND MuLLER, M. E. (1967). DITRAN—A compiler emphasizing diagnostics. *Comm. A.C.M.*, **10**, 1, pp. 45-52.

NAUR, P. (Ed.) (1960). Report on the algorithmic language ALGOL 60. *Comm. A.C.M.*, **3**, 5, pp. 299-314.

NAUR, P. (Ed.) (1963). Revised report on the algorithmic language ALGOL 60. *Comm. A.C.M.*, **6**, 1, pp. 1-17.

NAUR, P. (1964). Environmental enquiries: machine dependent programming in common languages. *ALGOL Bulletin*, 18, 3.9.1.

NAUR, P. (1965). Checking of operand typos in ALGOL compilers. *BIT*, **5**, pp. 151-163.

PALME, J. (1968). A comparison between SIMULA and FORTRAN. *BIT*, **5**, pp. 203-209.

PALME, J. (1969). Whal is a good programming language? Report FOA P C8231-61 (11). Research Institute of National Defence, Stockholm.

PECK, J. E. L. (Ed.) (1971). ALGOL 68 Implementation. Proc. IFIP Working Conference on ALGOL 68 Implementation. Munich 1970. North-Holland, Amsterdam.

POLLACK, B. W. (1971). An annotated bibliography on the construction of compilers. Stanford Computer Science Department Report CS 249.

RANDELL, B. AND RUSSELL, L. J. (1964). “ALGOL 60 Implementation.” APIC Studies in Data Processing, No 5. Academic Press, London.

RICHARDS, M. (1969). The BCPL Reference Manual. Technical Memorandum 69/1, Computer Laboratory, Cambridge.

RICHARDS, M. (1971). The portability of the BCPL compiler. *Software—Practice and Experience*, **1**, 2, pp. 135-146.

SAMET, P. A. (1966). The efficient administration of blocks in ALGOL 60. *Camp. J.*, **8**, 1, pp. 21-23.

SATIERTHWAITE, E. (1971). Debugging tools for high level languages. TR 29, Computing Laboratory, Newcastle-upon-Tyne.

SCHMID, E. (1970). Rechenzeitenvergleich bei Digita1rechnem. Computing, **5**, pp.163-177.

SCOWEN, R. S. (1965). Quickersort, Algorithm 271. *Comm. A.C.M.*, **8**, 11, p. 669.

SCOWEN, R. S., HILLMAN, A. L. AND SIDMELL, M. (1969a). SOAP—Simplify Obscure ALGOL Programs. National Physical Laboratory Report CCU 6.

SCOWEN, R. S. (1969b). BABEL, a new programming language. National Physical Laboratory Report CCU 7.

SCOWEN, R. S., AUlN, D., HILLMAN, A. L. AND SHIMELL, M. (1971). SOAP—A program which documents and edits ALGOL 60 programs. *Camp. J.*, **14**, 2, pp. 133-135.

SCOWEN, R. S. (1972). Debugging computer programs, A survey with special emphasis on ALGOL. National Physical Laboratory Report NAC 21.

SITES, R. L. (1971). ALGOL W Reference Manual. Stanford Computer Science Department Report CS 230.

SNOWDON, R. A. (1971). PEARL, An interactive system for the preparation and validation of structured programs. TR 28, Computing Laboratory, Newcastle-upon-Tyne.

SUNDBLAD, Y. (1971). The Ackermann function, a theoretical, computational and formula manipulative study. *BIT*, **11**, pp. 107-119.

WATT, J. M. (1963). The realization of ALGOL procedures and designational expressions. *Camp. J.*, **5**, 4, pp. 332-337.

WICHMANN, B. A. (1968). Timing ALGOL Statements. *ALGOL Bulletin*, AB 27.3.2.

WICHMANN, B. A. (1969). A comparison of ALGOL 60 execution speeds. National Physical Laboratory Report CCU 3.

WICHMANN, B. A. (1970a). PL 516, an ALGOL-like assembly language for the DDP-516. National Physical Laboratory Report CCU 9.

WICHMANN, B. A. (1970b). Some statistics from ALGOL programs. National Physical Laboratory Report CCU 11.

WICHMANN, B. A. (1971). The performance of some ALGOL systems. *Proc. I.F.I.P. Congress 1971*, pp. 327-334. North-Holland, Amsterdam.

WICHMANN, B. A. (19720). Five ALGOL compilers. *Camp. J.*, **15**, 1, pp. 8-12.

WICHMANN, B. A. (1972b). A note on the use of variables in ALGOL 60. National Physical Laboratory Report NAC 14.

WICHMANN, B. A. (1972c). Basic statement times for ALGOL 60. National Physical Laboratory Report NAC 15.

WICHMANN, B. A. (1972d). A note on the IF1P input-output proposals. *Software—Practice and Experience* **2**, 4, pp. 403-404.

WIJNGAARDEN, A. VAN (Ed.)(1969). Report on the Algorithmic Language ALGOL 68. *Num. Math.*, **14**, 2, pp. 79-218.

WINGEN, J. W. VAN (1972). Three papers an input-output in ALGOL 60. Nederlands Institute for Preventive Medicine.

WIRTH, N. AND HOARE, C. A. R. (1966a). A contribution to the development of ALGOL. *Camm. A.C.M.*, **9**, 6, pp. 413-431.

WIRTH, N. AND WEBER, H. (1966b). EULER: A generalization of ALGOL, and its formal definition. Part 1: *Comm. A.C.M.*, **9**, 1, pp. 13-23. Part 2: *Comm. A.C.M.*, **9**, 2, pp. 89-99.

WIRTH, N. (1968). PL/360, A programming language for the 360 computers. *J.A.C.M.*, **15**, 1, pp. 37-41.

WIRTH, N. (1971a). The programming language PASCAL. *Acta Informatica*, **1**, 1 pp. 35-63.

WIRTH, N. (1971b). The design of the PASCAL compiler. *Software—Practice and Experience*, **1**, 4, pp. 309-334.

WIRTH, N. (1972). On PASCAL, code generation, and,the CDC 6600 computer. Stanford Computer Science Department Report CS 257.

WOODGER, M. (1963). The description of computing processes: Some observations on Automatic Programming and ALGOL 60. “Annual Review in Automatic Programming,” Vol. 3. Pergamon Press, Oxford.

WOODWARD, P. M., WETHERALL, P. R. AND GORMAN, B. (1970). Official definition of CORAL 66. Her Majesty’s Stationery Office.

WOODWARD, P. M. (1971). AND BOND, S. G. Users’ Guide to ALGOL 68-R. Royal Radar Establishment (1971) and Her Majesty’s Stationery Office (1972).

WOODWARD, P. M. (1972). Practical experience with ALGOL 68. *Software—Practice and Experience*, **2**, 1, pp. 7-19.

WULF, W. S.,RUSSELL, D. B. AND HABERMANN, A. N. (1971). BLISS: A language for systems programming. *Comm. A.C.M.*, **14**, 2, pp. 780-790.

YERSHOV, A. P., KOZHUKIUM, G. I. AND VOWSHIN, U. M. (1963). “Input Language for Automatic Programming Systems.” APIC Studies in Data Processing, No 3. Academic Press, London.

YERSHOV, A. P. (Ed.) (1971). “The ALPHA Automatic Programming System.” APIC Studies in Data Processing No 7. Academic Press, London.

Appendix 1

The ALGOL Computer

The computer used to illustrate the generation of machine code is supposed to be typical of third generation hardware. There is remarkably small variation in the architecture of such machines—the B5500 being probably the only successful radical departure. In order to illustrate different techniques which depend upon points of detail, various aspects of the computer are deliberately undefined. The undefined aspects of the machine are as follows:-

1. word length
2. number of “index” or integer registers
3. address length
4. numeric values of the operation codes, and instruction lay-out.

The machine has a number of positive features. It is a one-address machine with a single floating point accumulator. The index registers are numbered from 1 upwards. Every index register may be used for address modification, in which case the actual address of the operand is the sum of the relevant register and the address field of the instruction. The tacit assumption is made in a few places that integers and floating point numbers both take up one word. This is quite often not the case, but it avoids confusing the generated code with extra details.

Programs for the machine can only be written in assembly language. This language resembles very many genuine computer languages in the hope that readers will not need to refer to this appendix too often. Each line represents one instruction which is in up to four parts, namely a label, the operation, the address and then a comment. Mnemonics must be used for the operations, and are usually used for addresses. The syntax of the language is as follows

```
<program>          ::=  <instruction> | <instruction> <program>
<instruction>      ::=  <label part> <operation part> <address part>
<label part>       ::=  <identifier> :| <empty>
<operation part>   ::=  <operation> | <register operation>, <register number>
<address part>     ::=  <number> | <identifier> |
                        <register number>, <identifier> |
                        <register number>, (<unsigned integer>)
<register number>  ::=  <unsigned integer>
```

Instruction	Meaning
ADA	AC := AC + OP
ADI,n	$r(n)$:= $r(n)$ + OP
ANI,n	$r(n)$:= $r(n)\wedge$ OP
CALL,n	$r(n)$:= P + 1; P := OP
DVA	AC := AC/OP
JAN	jump if AC < 0
JAP	jump if AC > 0
JAZ	jump if AC $= 0$
JIN,n	jump if $r(n) > 0$
JIP,n	jump if $r(n) < 0$
JIZ,n	jump if $r(n) = 0$
JMP	unconditional jump
JPI	indirect jump, P := OP
LDA	AC := OP
LDI,n	$r(n)$:= OP
MLA	AC := AC $\times$ OP
MLI,n	$r(n)$:= $r(n)\times$ OP
ORI,n	$r(n)$:= $r(n)\vee$ OP
SBA	AC := AC $-$ OP
SBI,n	$r(n)$:= $r(n)-$ OP
STA	OP := AC
STI,n	OP := $r(n)$

Table 1.1: Illustrative computer instructions

Compiler directives to terminate the program, set the position of the compiled code, etc., are not defined (since they are not required in the examples). The machine operations either act upon the accumulator or other implicit variables in the computer or upon the register specified by the number immediately following the operation.

The address part can be just a number, in which case this is the operand to be used. It is not specified as to whether this is implemented by a literal mode of addressing or whether a word containing the literal must be set aside by the assembler. The literal can be any number as defined in 2.5.1 of the ALGOL Report. When the address part is a single identifier, the operand is this identifier. When the identifier is preceded by a register name, then the actual address of the operand is the address corresponding to the identifier plus the value of the register. In the last case, the address of the operand is given by the sum of the unsigned integer and the value of the register.

The instructions listed in Table 1.1 are very modest. No register manipulation orders are included (i.e. ones which do not address the main store of the machine). To overcome this (or rather to avoid introducing such stores), the registers can be referenced like memory locations using the reserved identifiers REG1, REG2, etc. In a few minor cases, it is assumed that limited compile-time address calculation can be performed by the assembler. For instance, the word following the one identified by B is referred to as B+1 in the address part of the instruction.

In Table 1.1, the unsigned integer n denotes the register, $r(n)$ the register value, OP the operand value, P the instruction address and AC the accumulator. Operations on the accumulator are assumed to be floating point.

Additional “instructions” are sometimes used which are more likely to be implemented as compiler macros or subroutine calls. These are explained in the context in which they are used. Unless otherwise stated, various conventions are adopted about the use of the first few registers. These conventions are as follows:-

REG1	Used for the calculation of integer expressions
REG2	Used for subscripts in integer expressions, the address of array variables on assignment, integer divide, etc.
REG3	Local procedure pointer
REG4	Non-local procedure pointer
REG5	Stack front.

Examples

```
    LDI,2        3,(0)
```

Loads register 2 with the word whose address is in register 3.

```
    LDA          1.0
```

Loads the accumulator with the floating point number 1.0.

```
    STA          3,6
```

This is invalid, since assignment to constants is not allowed.

```
    LDI,1        B + 1
```

Load register 1 with the word whose address is one more than that of B.

```
    SBI,1        REG3
```

Subtracts the value of register 3 from register 1.

```
    CALL,3       PROC
```

This is an example of a subroutine call to the instruction address PROC. Register 3 contains the address of the instruction after the CALL. Hence the return could be affected by

```
    JPI          REG3
```

assuming that the subroutine PROC does not use register 3.

Appendix 2

Statement Times and Estimated Values

This table lists the time taken in microseconds to execute the 41 statements and the **for** loop control code (in **step** 1 **until** n **do**). Any value which is not known, or any which has been measured but is suspect, has been replaced by an estimate using the model described in 2.2.4. Such estimates are marked with a asterisk. A few compilers did not produce any code for the dummy block **begin real** a; **end**. Since this introduces a singularity into the model, a time of about half an instruction time has been used instead and this is marked with a $\neq$.

IBM 370/165	ATLAS	1904A XALT MK5	1906A XALT MK5	ALGOL 60/RRE	RRE ALGOL 68	*statement*
1.4	6.0	9.0	1.4	15.0	14.0	x := 1.0
1.9	6.0	10.0	1.3	15.0	54.0	x := 1
1.4	6.0	9.0	1.4	*12.4	*12.6	$x := y$
1.4	9.0	14.0	2.0	23.0	23.0	$x := y + z$
1.4	12.0	20.0	3.2	39.0	39.0	$x := y \times z$
2.7	18.0	34.0	7.2	72.0	71.0	$x := y/z$
1.4	9.0	5.0	1.0	8.0	8.0	k := 1
4.5	18.0	5.0	0.9	8.0	121.0	k := 1.0
1.4	12.0	7.0	2.5	12.0	13.0	$k := l + m$
1.4	15.0	37.0	4.9	78.0	75.0	$k := l \times m$
1.9	48.0	24.0	6.7	56.0	45.0	$k := l \div m$
1.4	9.0	5.0	1.9	8.0	8.0	$k := l$
1.9	6.0	23.0	3.1	22.0	44.0	$x := l$
4.7	18.0	35.0	8.0	113.0	122.0	$l := y$
11.7	39.0	60.0	20.3	104.0	180.0	$x := y \uparrow 2$
12.3	48.0	71.0	23.0	120.0	213.0	$x := y \uparrow 3$
53.6	120.0	166.0	55.0	286.0	978.0	$x := y \uparrow z$
1.6	21.0	8.0	1.8	29.0	22.0	$e1[1]$:= 1
1.7	27.0	8.0	1.9	71.0	54.0	$e2[1, 1]$:= 1
1.7	33.0	8.0	1.9	120.0	106.0	$e3[1, 1, 1]$:= 1
1.6	15.0	8.0	2.4	29.0	22.0	$l := e1[1]$
19.4	45.0	$\neq$1.0	$\neq$0.4	$\neq$1.5	52.0	**begin real** a; **end**
130.0	96.0	321.0	80.0	664.0	242.0	**begin array** $a[1 : 1]$; **end**
242.0	96.0	321.0	86.0	4200.0	232.0	**begin array** $a[1 : 500]$; **end**
137.0	156.0	418.0	106.0	854.0	352.0	**begin array** $a[1 : 1, 1 : 1]$; **end**
138.0	216.0	514.0	124.0	1010.0	452.0	**begin array** $a[1 : 1, 1 : 1, 1 : 1]$; **end**
0.6	42.0	11.0	3.5	16.0	16.0	**begin goto** $abcd$; $abcd$: **end**
35.2	129.0	30.0	9.4	54.0	62.0	**begin switch** $s := q$; **goto** $s[1]$; q : **end**
20.1	210.0	235.0	73.0	394.0	692.0	$x := sin(y)$
19.4	222.0	235.0	73.0	404.0	462.0	$x := cos(y)$
1.4	84.0	53.0	17.3	84.0	22.0	$x := abs(y)$
23.6	270.0	264.0	71.0	494.0	562.0	$x := exp(y)$
19.6	261.0	71.0	24.7	116.0	462.0	$x := ln(y)$
24.7	246.0	256.0	73.0	464.0	432.0	$x := sqrt(y)$
29.2	272.0	315.0	91.0	564.0	622.0	$x := arctan(y)$
3.0	99.0	72.0	18.7	114.0	72.0	$x := sign(y)$
5.7	99.0	99.0	24.7	164.0	152.0	$x := entier(y)$
31.6	54.0	129.0	43.0	284.0	72.0	$p0$
60.7	69.0	176.0	57.0	414.0	92.0	$p1(x)$
83.6	75.0	195.0	65.0	464.0	132.0	$p2(x, y)$
108.0	93.0	216.0	71.0	504.0	162.0	$p2(x, y, z)$
*4.0	57.0	22.0	8.6	46.0	17.0	loop time

PDP10	B5500	B6500	B6700	X8 ALGOL	X8 Eindhoven	*statement*
5.93	12.1	6.5	5.5	19.0	10.8	x := 1.0
5.74	8.1	4.1	2.7	13.5	10.8	x := 1
7.58	11.6	5.1	3.9	15.0	16.0	$x := y$
14.80	18.8	7.5	5.5	25.0	24.0	$x := y + z$
20.10	50.0	13.0	11.3	30.0	31.3	$x := y \times z$
28.20	32.5	19.7	18.0	80.0	87.5	$x := y/z$
6.18	8.1	4.3	2.9	7.5	8.2	k := 1
6.20	25.0	4.3	2.7	17.5	8.2	k := 1.0
11.80	18.8	8.2	6.0	15.0	22.4	$k := l + m$
19.20	35.0	11.1	8.9	20.0	30.4	$k := l \times m$
28.20	34.6	12.3	10.3	105.0	171.0	$k := l \div m$
7.16	11.6	6.0	4.3	10.0	10.6	$k := l$
57.30	11.8	5.1	3.6	12.5	13.5	$x := l$
51.30	26.1	9.1	6.0	17.5	20.7	$l := y$
21.40	46.6	11.8	10.8	200.0	32.0	$x := y \uparrow 2$
123.00	85.0	21.1	20.2	220.0	352.0	$x := y \uparrow 3$
85.70	1760.0	139.0	108.0	168.0	186.0	$x := y \uparrow z$
10.60	24.0	8.4	5.3	50.0	129.0	$e1[1]$:= 1
14.70	42.8	12.0	7.7	100.0	284.0	$e2[1, 1]$:= 1
19.80	66.6	15.9	11.3	250.0	425.0	$e3[1, 1, 1]$:= 1
11.60	23.5	11.8	6.7	25.0	82.4	$l := e1[1]$
28.50	22.3	36.2	33.8	108.0	270.0	**begin real** a; **end**
280.00	2870.0	540.0	408.0	448.0	1250.0	**begin array** $a[1 : 1]$; **end**
277.00	2870.0	540.0	408.0	453.0	1490.0	**begin array** $a[1 : 500]$; **end**
395.00	8430.0	1710.0	1250.0	595.0	1940.0	**begin array** $a[1 : 1, 1 : 1]$; **end**
553.00	13000.0	1820.0	1330.0	1120.0	1650.0	**begin array** $a[1 : 1, 1 : 1, 1 : 1]$; **end**
2.33	31.5	1.9	1.7	3.5	2.8	**begin goto** $abcd$; $abcd$: **end**
96.30	98.3	*66.5	47.5	273.0	1260.0	**begin switch** $s := q$; **goto** $s[1]$; q : **end**
211.00	598.0	195.0	161.0	460.0	512.0	$x := sin(y)$
268.00	758.0	211.0	171.0	510.0	568.0	$x := cos(y)$
6.92	14.0	6.7	5.5	18.0	32.4	$x := abs(y)$
226.00	740.0	307.0	250.0	755.0	824.0	$x := exp(y)$
26.10	808.0	238.0	192.0	667.0	726.0	$x := ln(y)$
144.00	605.0	303.0	262.0	380.0	430.0	$x := sqrt(y)$
211.00	841.0	274.0	221.0	795.0	858.0	$x := arctan(y)$
75.50	37.5	13.7	11.5	20.0	36.0	$x := sign(y)$
82.80	41.0	22.6	15.4	25.0	41.0	$x := entier(y)$
224.00	31.0	27.6	26.4	173.0	374.0	$p0$
368.00	39.0	30.7	28.6	300.0	448.0	$p1(x)$
455.00	45.0	32.4	30.5	425.0	522.0	$p2(x, y)$
593.00	53.0	36.0	33.6	560.0	596.0	$p2(x, y, z)$
11.90	38.5	7.2	7.2	80.0	72.0	loop time

ALCOR 7094/1	KDF9 Egdon	KDF9 Mk2	Babel KDF9	CDC 6600	CDC 6400	*statement*
6.5	22.0	22.0	22.0	2.2	4.4	$x :=$ 1.0
10.0	26.5	21.0	22.0	2.1	4.5	$x :=$ 1
7.5	22.0	22.0	22.0	2.0	4.4	$x := y$
9.5	32.7	32.7	32.7	2.5	6.1	$x := y + z$
*15.3	32.7	32.7	32.7	2.9	10.8	$x := y \times z$
16.5	45.2	45.2	45.2	4.8	10.3	$x := y/z$
6.5	18.3	18.0	17.0	2.1	4.5	$k :=$ 1
30.0	54.6	92.0	37.0	1.9	4.3	$k :=$ 1.0
11.0	32.7	32.7	32.7	2.6	6.1	$k := l + m$
15.0	34.0	34.0	34.0	3.0	10.9	$k := l \times m$
45.0	134.0	122.0	132.0	6.0	14.4	$k := l \div m$
8.5	22.0	22.0	22.0	2.1	4.6	$k := l$
6.5	28.7	113.0	126.0	2.1	4.5	$x := l$
29.5	54.6	86.0	40.0	4.1	9.0	$l := y$
15.5	402.0	25.0	230.0	3.8	10.6	$x := y \uparrow 2$
33.5	435.0	41.0	258.0	4.6	18.4	$x := y \uparrow 3$
381.0	360.0	882.0	1350.0	124.0	299.0	$x := y \uparrow z$
25.0	38.7	46.0	41.6	2.8	7.7	$e1[1] :=$ 1
21.0	67.8	89.0	75.7	2.8	7.8	$e2[1, 1] :=$ 1
22.0	247.0	361.0	138.0	4.6	9.6	$e3[1, 1, 1] :=$ 1
9.0	42.7	48.0	44.8	3.5	8,4	$l := e1[1]$
7.0	16.3	34.0	1380.0	41.0	72.2	**begin real** a; **end**
308.0	226.0	758.0	1950.0	72.4	126.0	**begin array** $a[1 : 1]$; **end**
308.0	226.0	763.0	1950.0	71.0	127.0	**begin array** $a[1 : 500]$; **end**
418.0	509.0	891.0	2100.0	75.1	130.0	**begin array** $a[1 : 1, 1 : 1]$; **end**
526.0	600.0	1030.0	2440.0	75.7	131.0	**begin array** $a[1 : 1, 1 : 1, 1 : 1]$; **end**
*8.1	10.0	25.0	1910.0	9.6	14.4	**begin goto** $abcd$; $abcd$: **end**
205.0	465.0	582.0	174.0	60.5	108.0	**begin switch** $s := q$; **goto** $s[1]$; q : **end**
265.0	547.0	629.0	713.0	60.2	180.0	$x := sin(y)$
311.0	595.0	664.0	751.0	58.8	177.0	$x := cos(y)$
11.5	22.7	124.0	21.0	4.2	7.2	$x := abs(y)$
206.0	434.0	647.0	724.0	61.5	179.0	$x := exp(y)$
80.0	301.0	753.0	820.0	60.2	175.0	$x := ln(y)$
168.0	269.0	396.0	455.0	49.5	138.0	$x := sqrt(y)$
336.0	1040.0	1080.0	1180.0	62.6	184.0	$x := arctan(y)$
16.0	32.0	219.0	23.0	7.0	9.6	$x := sign(y)$
28.0	63.1	302.0	162.0	39.5	71.7	$x := entier(y)$
281.0	276.0	74.0	1510.0	89.5	167.0	$p0$
407.0	295.0	85.0	1600.0	94.3	174.0	$p1(x)$
503.0	325.0	114.0	1630.0	99.8	185.0	$p2(x, y)$
598.0	356.0	136.0	1660.0	105.0	195.0	$p2(x, y, z)$
*23.8	*49.3	129.0	42.0	13.5	28.9	loop time

NU ALGOL 1108	ALGOL 60 4/50	ICL 4/70	ALGOL W 4/75	ALGOL W 360/67	IBM 360/65	*statement*
1.5	19.0	6.9	3.38	2.09	5.6	x := 1.0
1.5	443.0	19.8	13.00	2.25	21.7	x := 1
1.5	25.7	4.6	3.28	2.17	4.2	x := y
3.4	52.7	9.2	7.01	4.36	4.4	x := $y + z$
*4.1	245.0	17.5	9.86	6.47	6.3	x := $y \times z$
10.3	181.0	24.7	14.00	9.46	7.3	x := y/z
1.0	15.1	2.5	2.71	1.55	6.0	k := 1
1.0	222.0	48.5	80.80	43.30	32.4	k := 1.0
2.3	28.4	5.1	5.04	3.69	15.7	k := $l + m$
3.4	63.4	9.7	11.70	11.10	22.6	k := $l \times m$
12.5	172.0	17.2	16.30	14.70	26.5	k := $l \div m$
1.5	18.0	3.1	3.03	2.10	12.4	k := l
2.7	460.0	22.2	15.30	11.80	31.6	x := l
3.1	222.0	49.5	81.90	43.40	31.5	l := y
9.7	635.0	102.0	8.71	6.35	49.6	x := $y \uparrow 2$
9.3	835.0	110.0	134.00	10.60	54.2	x := $y \uparrow 3$
9.7	5810.0	582.0	184.00	236.00	131.0	x := $y \uparrow z$
2.7	87.0	14.2	11.10	4.54	6.7	$e1[1]$:= 1
5.8	156.0	25.2	20.50	11.10	6.9	$e2[1,1]$:= 1
9.0	613.0	120.0	31.40	17.70	6.8	$e3[1,1,1]$:= 1
3.4	104.0	21.5	10.40	5.10	5.1	l := $e1[1]$
46.3	18.6	3.2	26.30	31.20	327.0	**begin real** a; **end**
161.0	789.0	155.0	59.90	74.60	649.0	**begin array** $a[1:1]$; **end**
918.0	789.0	155.0	59.70	74.20	708.0	**begin array** $a[1:500]$; **end**
181.0	879.0	188.0	89.40	101.00	685.0	**begin array** $a[1:1,1:1]$; **end**
161.0	1160.0	214.0	117.00	127.00	661.0	**begin array** $a[1:1,1:1,1:1]$; **end**
0.7	51.0	7.5	4.07	2.57	*7.4	**begin goto** $abcd$; $abcd$: **end**
97.4	824.0	175.0	23.80	5.56	344.0	**begin switch** s := q; **goto** $s[1]$; q : **end**
38.6	4450.0	564.0	152.00	106.00	83.7	x := $sin(y)$
39.4	4460.0	571.0	146.00	102.00	72.0	x := $cos(y)$
1.5	668.0	158.0	4.43	2.87	2.8	x := $abs(y)$
25.7	4190.0	585.0	144.00	122.00	78.0	x := $exp(y)$
34.9	4580.0	587.0	132.00	87.40	71.5	x := $ln(y)$
33.1	2720.0	470.0	113.00	83.10	68.5	x := $sqrt(y)$
58.1	7110.0	755.0	135.00	89.00	54.0	x := $arctan(y)$
5.8	1110.0	174.0	*21.10	10.20	25.6	x := $sign(y)$
13.4	1480.0	233.0	98.10	55.20	43.3	x := $entier(y)$
113.0	568.0	135.0	43.50	31.60	373.0	$p0$
127.0	598.0	143.0	111.00	45.60	500.0	$p1(x)$
137.0	618.0	150.0	178.00	61.60	603.0	$p2(x,y)$
148.0	658.0	158.0	241.00	76.70	726.0	$p2(x,y,z)$
8.7	192.0	49.5	9.30	7.98	*22.0	loop time

Appendix 3

Ratios of Observed and Expected Statement Times

These ratios are calculated from the model given in 2.2.4. They give an easy method of pinpointing the strength and weaknesses of particular statements in relation to the overall performance of any system. For instance, a ratio of 2 or greater indicates poor implementation (by hardware or software), whereas a ratio of less than $\frac{1}{2}$ implies a good implementation of the relevant statement. Statements whose times have been estimated, or falsely put to a small non-zero value are marked by * and $\neq$ respectively, as in Appendix 2.

3.0.1 Residuals

IBM 370/165	ATLAS	1904A XALT MK5	1906A XALT MK5	ALGOL 60/RRE	RRE ALGOL 68	*statement*
1.240	0.936	1.490	0.903	1.160	1.130	$x := 1.0$
1.160	0.647	1.140	0.508	0.802	3.030	$x := 1$
1.290	0.974	1.550	0.939	*1.000	*1.000	$x := y$
0.811	0.921	1.520	0.846	1.170	1.220	$x := y + z$
0.540	0.817	1.440	0.901	1.320	1.380	$x := y \times z$
0.693	0.816	1.630	1.350	1.620	1.670	$x := y/z$
1.640	1.860	1.100	0.857	0.822	0.861	$k := 1$
1.870	1.320	0.388	0.273	0.291	4.610	$k := 1.0$
0.911	1.380	0.851	1.190	0.684	0.776	$k := l + m$
0.510	0.965	2.520	1.300	2.490	2.510	$k := l \times m$
0.383	1.710	0.905	0.987	0.989	0.833	$k := l \div m$
1.330	1.510	0.890	1.320	0.667	0.699	$k := l$
0.786	0.438	1.780	0.936	0.797	1.670	$x := l$
1.080	0.734	1.510	1.350	2.280	2.580	$l := y$
1.920	1.130	1.840	2.440	1.500	2.720	$x := y \uparrow 2$
1.080	0.747	1.170	1.480	0.927	1.720	$x := y \uparrow 3$
1.320	0.524	0.767	0.992	0.619	2.220	$x := y \uparrow z$
0.650	1.510	0.607	0.534	1.030	0.820	$e1[1] := 1$
0.455	1.280	0.400	0.371	1.660	1.330	$e2[1,1] := 1$
0.274	0.941	0.241	0.224	1.700	1.570	$e3[1,1,1] := 1$
0.667	1.110	0.624	0.731	1.060	0.842	$l := e1[1]$
4.510	1.850	≠0.044	≠0.068	≠0.031	1.110	**begin real** a; **end**
2.580	0.336	1.190	1.160	1.150	0.440	**begin array** $a[1:1]$; **end**
3.980	0.279	0.987	1.030	6.050	0.350	**begin array** $a[1:500]$; **end**
1.880	0.379	1.080	1.070	1.030	0.445	**begin array** $a[1:1,1:1]$; **end**
1.610	0.445	1.120	1.060	1.040	0.484	**begin array** $a[1:1,1:1,1:1]$; **end**
0.440	5.440	1.510	1.880	1.030	1.080	**begin goto** $abcd$; $abcd$: **end**
1.960	1.270	0.313	0.383	0.264	0.318	**begin switch** $s := q$; **goto** $s[1]$; q : **end**
0.491	0.906	1.070	1.300	0.843	1.550	$x := sin(y)$
0.463	0.935	1.050	1.270	0.844	1.010	$x := cos(y)$
0.526	5.570	3.720	4.750	2.760	0.758	$x := abs(y)$
0.535	1.080	1.120	1.180	0.982	1.170	$x := exp(y)$
0.629	1.470	0.424	0.576	0.325	1.350	$x := ln(y)$
0.706	1.240	1.370	1.520	1.160	1.130	$x := sqrt(y)$
0.557	0.917	1.120	1.270	0.942	1.090	$x := arctan(y)$
0.556	3.240	2.500	2.530	1.850	1.230	$x := sign(y)$
0.545	1.670	1.770	1.720	1.370	1.330	$x := entier(y)$
1.630	0.492	1.240	1.620	1.280	0.341	$p0$
2.380	0.477	1.290	1.630	1.420	0.331	$p1(x)$
2.770	0.439	1.210	1.570	1.350	0.401	$p2(x,y)$
3.100	0.470	1.160	1.480	1.260	0.425	$p2(x,y,z)$
*1.000	2.500	1.020	1.560	1.000	0.387	loop time

PDP10	B5500	B6500	B6700	X8 ALGOL	X8 Eindhoven	*statement*
0.916	1.020	1.550	1.670	1.500	0.634	x := 1.0
0.613	0.471	0.674	0.565	0.735	0.438	x := 1
1.220	1.020	1.260	1.230	1.230	0.977	x := y
1.500	1.040	1.170	1.090	1.290	0.924	x := $y + z$
1.360	1.840	1.350	1.490	1.030	0.802	x := $y \times z$
1.270	0.795	1.360	1.580	1.830	1.490	x := y/z
1.270	0.905	1.360	1.170	0.785	0.639	k := 1
0.450	0.989	0.481	0.384	0.648	0.226	k := 1.0
1.340	1.170	1.440	1.340	0.870	0.969	k := $l + m$
1.220	1.210	1.090	1.110	0.650	0.736	k := $l \times m$
1.020	0.665	0.668	0.712	1.890	2.290	k := $l \div m$
1.190	1.050	1.540	1.400	0.840	0.670	k := l
4.140	0.465	0.568	0.510	0.461	0.371	x := l
2.070	0.574	0.565	0.475	0.360	0.317	l := y
0.615	0.730	0.522	0.609	2.930	0.350	x := $y \uparrow$ 2
1.900	0.714	0.501	0.610	1.730	2.060	x := $y \uparrow$ 3
0.370	4.140	0.925	0.917	0.370	0.305	x := $y \uparrow z$
0.753	0.929	0.918	0.738	1.810	3.480	$e1[1]$:= 1
0.688	1.090	0.865	0.707	2.390	5.050	$e2[1, 1]$:= 1
0.559	1.020	0.691	0.626	3.600	4.560	$e3[1, 1, 1]$:= 1
0.846	0.934	1.320	0.958	0.930	2.280	l := $e1[1]$
1.160	0.495	2.270	2.700	2.240	4.180	**begin real** a; **end**
0.971	5.430	2.880	2.780	0.792	1.640	**begin array** $a[1 : 1]$; **end**
0.797	4.500	2.390	2.300	0.665	1.630	**begin array** $a[1 : 500]$; **end**
0.951	11.100	6.330	5.880	0.730	1.770	**begin array** $a[1 : 1, 1 : 1]$; **end**
1.130	14.400	5.710	5.310	1.160	1.280	**begin array** $a[1 : 1, 1 : 1, 1 : 1]$; **end**
0.286	2.200	0.375	0.428	0.229	1.137	**begin goto** $abcd$; $abcd$: **end**
0.940	0.523	*1.000	0.909	1.360	4.670	**begin switch** s := q; **goto** $s[1]$; q : **end**
0.901	1.390	1.280	1.350	1.000	0.831	x := $sin(y)$
1.120	1.720	1.360	1.400	1.080	0.900	x := $cos(y)$
0.455	0.501	0.677	0.708	0.603	0.809	x := $abs(y)$
0.897	1.600	1.880	1.940	1.530	1.240	x := $exp(y)$
0.146	2.460	2.050	2.100	1.900	1.540	x := $ln(y)$
0.720	1.650	2.330	2.570	0.969	0.817	x := $sqrt(y)$
0.704	1.530	1.410	1.450	1.350	1.090	x := $arctan(y)$
2.450	0.663	0.684	0.731	0.331	0.444	x := $sign(y)$
1.380	0.374	0.581	0.505	0.213	0.260	x := $entier(y)$
2.020	0.152	0.383	0.467	0.796	1.280	$p0$
2.520	0.146	0.324	0.384	1.050	1.170	$p1(x)$
2.640	0.142	0.289	0.346	1.260	1.150	$p2(x, y)$
2.970	0.145	0.277	0.330	1.430	1.130	$p2(x, y, z)$
0.517	0.911	0.481	0.613	1.770	1.190	loop time

ALCOR 7094/1	KDF9 Egdon	KDF9 Mk2	Babel KDF9	CDC 6600	CDC 6400	*statement*
0.971	1.480	1.230	0.688	1.200	1.070	x := 1.0
1.030	1.230	0.815	0.476	0.791	0.754	x := 1
1.170	1.540	1.280	0.716	1.130	1.110	$x := y$
0.931	1.440	1.200	0.671	0.893	0.970	$x := y + z$
*1.000	0.958	0.801	0.447	0.689	1.140	$x := y \times z$
0.717	0.882	0.737	0.411	0.760	0.726	$x := y/z$
1.290	1.630	1.340	0.706	1.520	1.450	k := 1
2.110	1.720	2.430	0.544	0.486	0.400	k := 1.0
1.210	1.620	1.350	0.754	1.040	1.090	$k := l + m$
0.924	0.941	0.787	0.439	0.673	1.090	$k := l \times m$
1.540	2.050	1.430	0.943	0.746	0.797	$k := l \div m$
1.370	1.590	1.330	0.742	1.230	1.200	$k := l$
0.455	0.902	2.970	1.850	0.535	0.511	$x := l$
1.150	0.957	1.260	0.327	0.583	0.570	$l := y$
0.431	5.020	0.261	1.340	0.385	0.478	$x := y \uparrow 2$
0.500	2.920	0.230	0.806	0.250	0.445	$x := y \uparrow 3$
1.590	0.676	1.380	1.190	1.890	2.030	$x := y \uparrow z$
1.720	1.190	1.190	3.670	0.701	0.558	$e1[1]$:= 1
0.951	1.380	1.510	3.260	0.462	0.573	$e2[1, 1]$:= 1
0.601	3.040	3.700	2.760	0.458	0.425	$e3[1, 1, 1]$:= 1
0.635	1.350	1.270	3.750	0.900	0.961	$l := e1[1]$
0.276	0.288	0.503	11.400	5.880	4.610	**begin real** a; **end**
1.030	0.341	0.955	1.370	0.885	0.686	**begin array** $a[1 : 1]$; **end**
0.837	0.282	0.797	1.140	0.720	0.573	**begin array** $a[1 : 500]$; **end**
0.974	0.533	0.779	1.020	0.637	0.491	**begin array** $a[1 : 1, 1 : 1]$; **end**
1.040	0.532	0.764	1.010	0.544	0.419	**begin array** $a[1 : 1, 1 : 1, 1 : 1]$; **end**
*1.000	0.558	1.170	49.700	4.340	2.900	**begin goto** $abcd$; $abcd$: **end**
1.940	1.970	2.060	0.344	2.080	1.650	**begin switch** $s := q$; **goto** $s[1]$; q : **end**
1.090	1.020	0.976	0.617	0.906	1.210	$x := sin(y)$
1.260	1.080	1.010	0.635	0.865	1.160	$x := cos(y)$
0.731	0.648	2.960	0.279	0.972	0.742	$x := abs(y)$
0.791	0.748	0.933	0.582	0.860	1.110	$x := exp(y)$
0.432	0.730	1.530	0.928	1.190	1.530	$x := ln(y)$
0.813	0.585	0.720	0.461	0.873	1.080	$x := sqrt(y)$
1.080	1.510	1.310	0.796	0.736	0.963	$x := arctan(y)$
0.502	0.451	2.580	0.151	0.800	0.489	$x := sign(y)$
0.453	0.458	1.830	0.548	2.330	1.880	$x := entier(y)$
2.450	1.080	0.242	2.760	2.850	2.360	$p0$
2.700	0.878	0.212	2.210	2.280	1.870	$p1(x)$
2.820	0.817	0.240	1.910	2.040	1.650	$p2(x, y)$
2.900	0.774	0.247	1.680	1.850	1.530	$p2(x, y, z)$
*1.000	*1.000	2.030	0.369	2.070	1.970	loop time

NU ALGOL 1108	ALGOL 60 4/50	ICL 4/70	ALGOL W 4/75	ALGOL W 360/67	IBM 360/65	*statement*
0.846	0.358	0.811	0.796	0.781	0.906	x := 1.0
0.585	5.760	1.610	2.610	2.310	2.430	x := 1
0.880	0.503	0.566	0.803	0.812	0.707	x := y
1.260	0.651	0.710	1.080	1.120	0.467	x := $y + z$
*1.000	2.010	0.894	1.010	1.060	0.445	x := $y \times z$
1.690	0.993	0.842	0.958	1.020	0.343	x := y/z
0.749	0.377	0.384	0.847	1.040	1.290	k := 1
0.265	1.960	2.670	8.940	6.730	2.460	k := 1.0
0.956	0.394	0.436	0.874	0.959	1.870	k := $l + m$
0.791	0.492	0.468	1.140	1.690	1.510	k := $l \times m$
1.610	0.739	0.459	0.876	1.260	0.979	k := $l \div m$
0.912	0.365	0.396	0.769	0.842	2.160	k := l
0.713	4.050	1.220	1.690	1.570	2.390	x := l
0.457	1.090	1.510	5.040	3.740	1.330	l := y
1.020	2.220	2.220	0.382	0.429	1.490	x := $y \uparrow$ 2
0.524	1.570	1.290	3.150	3.450	0.873	x := $y \uparrow$ 3
0.153	3.060	1.900	1.210	2.170	0.591	x := $y \uparrow z$
0.701	0.753	0.767	1.200	0.957	0.498	$e1[1]$:= 1
0.992	0.893	0.896	1.460	1.410	0.338	$e2[1, 1]$:= 1
0.928	2.110	2.570	1.350	1.330	0.201	$e3[1, 1, 1]$:= 1
0.906	0.925	1.190	1.160	0.983	0.390	l := $e1[1]$
6.880	0.092	0.098	1.630	2.090	13.900	**begin real** a; **end**
2.040	0.334	0.409	0.317	0.513	2.360	**begin array** $a[1 : 1]$; **end**
9.650	0.277	0.339	0.262	0.426	2.130	**begin array** $a[1 : 500]$; **end**
1.590	0.258	0.344	0.328	0.509	1.730	**begin array** $a[1 : 1, 1 : 1]$; **end**
1.200	0.288	0.331	0.364	0.561	1.410	**begin array** $a[1 : 1, 1 : 1, 1 : 1]$; **end**
0.328	0.798	0.730	0.796	0.680	*1.000	**begin goto** $abcd$; $abcd$: **end**
3.470	0.980	1.290	0.354	0.275	3.520	**begin switch** s := q; **goto** $s[1]$; q : **end**
0.602	2.310	1.830	0.989	0.996	0.374	x := $sin(y)$
0.600	2.270	1.810	0.928	0.917	0.315	x := $cos(y)$
0.360	5.350	7.860	0.444	0.474	0.193	x := $abs(y)$
0.373	2.030	1.760	0.871	0.993	0.324	x := $exp(y)$
0.712	3.120	2.490	1.120	1.050	0.418	x := $ln(y)$
0.605	1.660	1.780	0.862	0.892	0.359	x := $sqrt(y)$
0.708	2.890	1.910	0.687	0.680	0.189	x := $arctan(y)$
0.687	4.400	4.290	*1.000	0.835	0.870	x := $sign(y)$
0.818	3.010	2.950	2.500	1.860	0.758	x := $entier(y)$
3.720	0.624	0.926	0.598	0.463	3.530	$p0$
3.180	0.499	0.743	1.160	0.691	3.590	$p1(x)$
2.900	0.436	0.661	1.570	0.855	3.660	$p2(x, y)$
2.710	0.401	0.599	1.840	0.973	3.810	$p2(x, y, z)$
1.380	1.020	1.630	0.616	0.515	*1.000	loop time

Appendix 4

Table of Machine Factors

ALGOL processing speed, ignoring the relative importance of the different statements, can be found from the statement factors given in the model discussed in 2.2.4. This is not a true measure of performance, since it is clear that some statements are used much less than others. The inverse of the model values are listed, to give larger values for more powerful computers.

1/machine factor	machine
1.00	Atlas
1.06	1904A XALT Mk5
4.13	1906A XALT Mk5
0.496	ALGOL 6O/RRE
0.519	RRE ALGOL 68
0.539	B5500
1.52	B6500
1.94	B6700
0.505	X8 ALGOL
0.376	X8 Eindhoven
0.43	KDF9-Edgon
0.36	KDF9-Mk2
0.201	Babel-KDF9
3.49	CDC6600
1.55	CDC 6400
0.121	ICL 4/50
0.75	ICL 4/70
1.51	ALGOL W 4/75
2.38	ALGOL W 360/67
1.04	IBM 360/65
5.66	IBM 370/165
0.99	PDP1O
0.958	ALCOR 7094/1
3.62	NU ALGOL 1108

Appendix 5

Average Statement Times

A further byproduct of the model used in 2.2.4 is the time taken to execute the statements on a machine of the same speed as ATLAS, but having a performance ratio of 1 throughout. This machine is therefore a mythical average machine. The number of instructions executed is roughly the time divided by 3, since ATLAS takes 3 microseconds to execute an "average" instruction. These figures allow one to compare any new compiler with the average on a statement by statement basis.

The times have been used to give program execution estimates given in chapter 3.

6.4	x := 1.0
9.3	x := 1
6.2	x := y
9.8	x := $y + z$
14.7	x := $y \times z$
22.1	x := y/z
4.8	k := 1
13.6	k := 1.0
8.7	k := $l + m$
15.5	k := $l \times m$
28.1	k := $l \div m$
5.9	k := l
13.7	x := l
24.5	l := y
34.4	x := $y \uparrow 2$
64.2	x := $y \uparrow 3$
229.0	x := $y \uparrow z$
13.9	$e1[1]$:= 1
21.2	$e2[1, 1]$:= 1
35.1	$e3[1, 1, 1]$:= 1
13.6	l := $e1[1]$
24.3	**begin real** a; **end**
285.0	**begin array** $a[1 : 1]$; **end**
344.0	**begin array** $a[1 : 500]$; **end**
411.0	**begin array** $a[1 : 1, 1 : 1]$; **end**
485.0	**begin array** $a[1 : 1, 1 : 1, 1 : 1]$; **end**
7.7	**begin goto** $abcd$; $abcd$: **end**
101.0	**begin switch** s := q; **goto** $s[1]$; q : **end**
232.0	x := $sin(y)$
237.0	x := $cos(y)$
15.1	x := $abs(y)$
249.0	x := $exp(y)$
177.0	x := $ln(y)$
198.0	x := $sqrt(y)$
297.0	x := $arctan(y)$
30.5	x := $sign(y)$
59.2	x := $entier(y)$
110.0	$p0$
145.0	$p1(x)$
171.0	$p2(x, y)$
198.0	$p2(x, y, z)$
22.8	loop time

Appendix 6

Interpretive Instruction Counts

This table gives the data upon which most of the statistical information on ALGOL 60 has been deduced. Each instruction of the interpretive system of Randell and Russell is listed in order of decreasing usage is given as instructions executed per million obeyed, and the variance of this figure with seven samples is given (as a percentage). The static frequency of usage is also given. Listed with this is the dynamic effect of the instruction on the stack. Some of the figures given for this are not strictly correct, but are shown so that the net result on the stack is zero. The nett effect on the stack gives an equation for d, the average array dimension (see 2.3.1.9).

Interpretive instruction counts

Frequency per million	Variance	Static frequency per thousand	Effect on stack	Operation
12 9000	10	73.5	1	Take Integer Result
84 200	20	44.2	-1	LINK (for loop control, see 2.3.15)
68 000	17	29.3	1	Take Real Result
55 100	18	48.4	1	Take Real Address
52 200	12	31.0	1	Take Integer Address
49 000	6	47.5	-2	STore
43 200	4	39.5	1	Take Integer Constant 1
39 700	10	26.4	$-d$	INDex Result (see 2.3.1.9)
36 200	9	18.9	-1	$\times$ (integer and real)
31 000	14	15.7	-1	+ (integer and real)
26 300	11	13.4	-1	$-$ (integer and real)
24 600	20	64.8	0	Unconditional Jump
24 400	39	10.2	1	Take Formal Address
24 200	12	21.8	$-d$	INDex Address (see 2.3.1.9)
23 500	22	24.8	0	TRACE (diagnostic, see 5.4.2)
22 500	33	37.6	1	Take Integer Constant
20 100	26	18.7	0	Procedure Entry
19 400	29	46.1	1	Call Function
19 300	18	10.2	0	For Return
18 100	26	12.9	0	Check and Store Real

Frequency per million	Variance	Static frequency per thousand	Effect on stack	Operation
17 800	21	9.7	0	FOR S2 (step-until)
15 400	16	12.8	−1	If False Jump
13 400	26	13,6	0	Block Entry (beginning of thunk)
13 200	27	12.3	−1	End Implicit Subroutine (end of thunk)
13 100	26	15.9	0	Check and Store Integer
11 000	27	6.7	−1	/
7450	13	33.1	−1	REJECT
6980	26	5.9	−1	=
6630	25	10.0	1	Take Integer Constant 0
6280	41	14.0	0	Real DOWN (code body)
5450	44	9.8	1	Take Real Constant
5230	21	3.2	0	NEGate (unary minus)
3600	24	2.7	−1	$>$
3530	59	2.1	−1	$\uparrow$ (see 1.1.4.2)
3380	28	12.5	−1	Can Function Zero
3290	38	2.1	−1	$<$
2730	44	3.6	0	RETURN
2700	24	10.2	0	For Block Entry
2690	40	1.3	1	Take Formal Real
2510	27	10.2	−1	For Statement End
2480	79	2.6	0	Check Arithmetic
2330	29	9.7	0	FOR S1 (step-until)
2230	56	1.8	−1	$\neq$
2180	32	1.2	−1	$\vee$
2010	51	8.9	1	Take Label
2010	51	7.4	−1	GoTo Accumulator
1890	42	1.8	1	Take Formal Address Integer
1820	109	3.5	1	Take Formal Integer
1790	76	2.0	0	Check STring
1750	49	0.22	0	SQRT
1700	93	1.2	1	Take Formal Address Real
1490	101	0.06	0	COS
1390	40	0.19	0	ABS
1370	207	1.6	0	Decrement Switch Index
1330	125	4.6	0	FOR Arithmetic
1030	70	1.2	−1	$\leq$
1020	108	0.06	0	SIN
987	154	0.81	0	Integer DOWN (code body)
977	48	2.3	−1	STore Also
909	122	0.17	0	ENTIER
914	86	0.59	−1	$\geq$
890	120	1.6	0	UP1 (code body)
884	74	1.3	−1	$\wedge$
831	92	0.09	0	EXP

Frequency per million	Variance	Static frequency per thousand	Effect on stack	Operation
817	85	0.85	1	Take Boolean Address
753	61	0.93	1	Take Boolean Result
664	112	1.6	0	UP2 (code body)
656	34	4.6	0	Check Array Real
644	83	0.13	0	LN
591	149	0.03	0	ARCTAN
481	70	1.0	-1	DIV (see 1.1.4.1)
471	180	0.63	0	Check Array Integer
377	91	0.10	0	FOR While
197	51	1.3	0	Can Block
157	49	3.4	$-2d$	Make Storage Function
163	194	0.36	1	Call Formal Function
124	68	0.31	1	Take Boolean Constant False
115	107	0.24	0	$\neg$
113	93	0.49	0	Check and Store Boolean
103	92	0.22	0	DUMMY
94	105	0.08	1	Take Switch Address
85	143	0.03	1	Take Formal Boolean
82	82	0.05	0	SIGN
78	126	0.27	1	Take Boolean Constant True
59	91	0.06	0	Check Label
43	85	0.08	0	Copy Real Formal Array
24	170	0.05	0	Check Boolean
9	153	0.04	0	Check PRocedure
4	56	0.52	0	FINISH
3	169	0.65	0	Check Function Real
2	91	0.59	0	Boolean DOWN
2	208	0.03	0	Check and Store Label
1	241	Not Found	-1	EQUILVALENT
0	0	0.08	0	End Switch List
0	223	Not Found	0	Copy Integer Formal Array
0	223	Not Found	0	TEST (diagnostic)
0	151	0.09	1	Take Formal Label
0	0	Not Found	0	Check Function Boolean
0	0	Not Found	0	Check Function Integer
0	0	Not Found	0	Avoid Own Array
0	0	Not Found	1	Call Formal Function Zero
0	0	Not Found	-1	IMPlies
0	0	Not Found	$-2d$	Make Own Storage Function
0	0	Not Found	0	Call Segment
0	0	Not Found	0	Check Switch
0	0	Not Found	0	Check Array Boolean
0	0	Not Found	0	Copy Boolean Formal Array

Frequency per million	Variance	Static frequency per thousand	Effect on stack	Operation
Not Executed		Not Found	—	Parameter Switch
		Not Found	—	Parameter Boolean Array
		Not Found	—	Parameter Boolean Constant
		6.5	—	Parameter Real Array
		0.93	—	Parameter Real Constant
		0.80	—	Parameter Integer Array
		33.5	—	Parameter Integer Constant
		0.12	—	Parameter Procedure
		1.9	—	Parameter Formal
		0.16	—	Parameter Label
		14.1	—	Parameter String
		0.12	—	Parameter Boolean
		8.6	—	Parameter Real
		19.4	—	Parameter Integer
		12.2	—	Parameter SubRoutine
		Not Found	—	Parameter Function Boolean
		0.70	—	Parameter Function Real
		0.04	—	Parameter Function Integer

Appendix 7

Table of ALGOL Basic Symbols

The frequency of ALGOL basic symbols as produced from 209 programs on KDF9 (see 2.3.3).

Programs can use both lower and upper case alphabetic characters but because of the one case on the line printer listing, the use of upper case characters is discouraged.

The statistics are listed as follows

1. Lower case letters
2. Upper case letters
3. Digits
4. Other basic symbols in order of decreasing use

Lower case alphabet: total frequency 286 539/million made up as follows:

a	21101	j	7882	s	14540
b	7707	k	5144	t	24 664
c	11 877	l	8786	u	5106
d	15597	m	12121	v	3518
e	20 537	n	18652	w	6446
f	9064	o	10172	x	12136
g	3115	p	9776	y	6225
h	4621	q	2685	z	1917
i	24809	r	18341		

Upper case alphabet: total frequency 50 270/million made up as follows:

A	3938	J	1093	S	2282
B	1706	K	661	T	2703
C	2208	L	2853	U	1270
D	2002	M	2210	V	1393
E	3721	N	2912	W	489
F	2008	0	2330	X	1166
G	951	P	2318	Y	1085
H	1123	Q	404	Z	269
I	3831	R	3344		

The digits were used with a total frequency of 71 105/million as follows:

0	16402
1	23 002
2	10518
3	9233
4	2789
5	3241
6	2079
7	1417
8	1298
9	1126

The remaining basic symbols are:

space	143 506	**integer**	2062
tab	125 162	**else**	1872
new line	54 026	**value**	1502
;	42 232	**goto**	1465
,	40 770	**comment**	1438
:=	20 618	for code body **KDF9**	1277
(	17 307	for code body **ALGOL**	1277
)	17 306	**array**	1211
[	15 317	$<$	775
]	15 377	$>$	737
$-$	7377	$\neq$	722
$\times$	7041	$_{10}$	712
+	7023	$\uparrow$	478
(open string quote) ‘	6405	$\wedge$	365
(close string quote) ’	6404	vee	257
(string space) ˽	4775	leq	233
end	4474	geq	222
begin	4473	(end program) $\rightarrow$	209
if	4137	$\div$	206
then	4137	(system) **library**	200
for	3534	**string**	106
do	3534	**true**	102
step	3411	**false**	100
until	3411	**Boolean**	95
=	2807	$\neg$	85
:	2795	**label**	51
.	2721	**while**	31
/	2541	**switch**	19
procedure	2169	(system) **segment**	8
real	2117	**own**	3
		$\equiv$	0
		$\supset$	0

Appendix 8

Statement Weights and Mix Figures

The program statistics were split into seven samples to illustrate the variation to be expected (see 2.4.2). The overall sample weight is given in the first column, the subsequent columns giving the seven individual weights.

								statement
10000	10000	10000	10000	10000	10000	10000	10000	x := 1.0
7000	7000	7000	7000	7000	7000	7000	7000	x := 1
10000	10000	10000	10000	10000	10000	10000	10000	$x := y$
26700	22000	25400	26900	29900	34600	26400	21700	$x := y + z$
31200	29100	36200	25000	29100	35400	31500	32200	$x := y \times z$
11000	11800	6470	13400	11800	15400	10800	7130	$x := y/z$
3000	3000	3000	3000	3000	3000	3000	3000	k := 1
500	500	500	500	500	500	500	500	k := 1.0
4300	4370	2360	5470	5570	3970	4410	3950	$k := l + m$
4980	5800	3360	5080	5420	4060	5260	5860	$k := l \times m$
481	725	910	224	947	193	287	83	$k := l \div m$
5000	5000	5000	5000	5000	5000	5000	5000	$k := l$
4000	4000	4000	4000	4000	4000	4000	4000	$x := l$
500	500	500	500	500	500	500	500	$l := y$
2780	5970	4480	1550	1920	2710	1160	1680	$x := y \uparrow 2$
309	664	497	172	214	302	129	187	$x := y \uparrow 3$
442	948	710	246	305	431	184	267	$x := y \uparrow z$
23800	23300	23900	25700	24500	20500	23000	25600	$e1[1] := 1$
16000	15600	16100	17300	16400	13700	15400	17200	$e2[1, 1] := 1$
296	289	198	320	304	255	285	319	$e3[1, 1, 1] := 1$
23800	23300	23900	25700	24500	20500	23000	25600	$l := e1[1]$
0	0	0	0	0	0	0	0	**begin real** a; **end**
59	32	119	73	66	32	46	40	**begin array** $a[1 : 1]$; **end**
59	32	119	73	66	32	46	40	**begin array** $a[1 : 500]$; **end**
39	22	80	49	44	22	31	27	**begin array** $a[1 : 1, 1 : 1]$; **end**
0.73	0.40	1.50	0.91	0.82	0.40	0.57	0.50	**begin array** $a[1 : 1, 1 : 1, 1 : 1]$; **end**
2010	1450	1710	2410	3110	1260	3640	508	**begin goto** $abcd$; $abcd$: **end**
94	0	131	50	306	13	131	29	**begin switch** $s := q$; **goto** $s[1]$; q : **end**
1020	2600	104	539	646	2860	264	119	$x := sin(y)$
1490	4430	883	555	683	3200	482	232	$x := cos(y)$
1390	747	741	1090	1710	1750	2420	1310	$x := abs(y)$
831	1700	317	449	493	377	2310	167	$x := exp(y)$
644	172	888	31	130	1530	592	1160	$x := ln(y)$
1750	719	1120	2200	1420	3600	1630	1580	$x := sqrt(y)$
591	257	40	1	370	2710	385	372	$x := arctan(y)$
82	103	47	2	65	119	22	216	$x := sign(y)$
909	132	20	827	1180	3430	767	9	$x := entier(y)$
788	308	1020	798	2630	91	573	89	$p0$
2320	5430	6780	2150	1770	340	192	2070	$p1(x)$
2320	5430	6780	2150	1770	340	192	2070	$p2(x, y)$
6050	7380	2520	6010	7440	6540	7750	4960	$p2(x, y, z)$
17800	13400	21000	17400	15200	14100	18700	24800	loop time

The mix figures on the different machines
Summary of the ALGOL mix figures

Mix figure	Machine	Comment
1.00	Atlas	see 9.3
0.91	1904A XALT Mk5	see 9.8
3.32	1906A XALT Mk5	see 9.8
0.406	ALGOL 60 RRE	
0.526	ALGOL 68 RRE	
0.527	B5500	see 9.5
1.55	B6500	
1.93	B6700	
0.37	X8 ALGOL	
0.261	X8 Eindhoven	
0.408	KDF9-Edgon	
0.393	KDF9 Mk2	see 9.4 and 10.3
0.127	Babel KDF9	
2.69	CDC6600 version 2.0	see 9.7
1.21	CDC6400 version 2.0	see 9.7
0.095	ICL 4/50	
0.645	ICL 4/70	
1.21	ALGOL W 4/75	
2.36	ALGOL W 360/67	
0.662	IBM 360/65	32-bit reals
4.07	IBMM 370/165	cache store
0.645	PDP10	
0.618	ALCOR 7094/1	
2.35	NU ALGOL 1108	see 9.6
0.014	KDF9 Whetstone	very slow, interpretive system
0.95	IBM 360/65	32-bit reals, with improvement package
0.881	IBM 360/65	64-bit reals, with improvement package
0.095	IBM 360/65	with slow core
0.163	IBM 360/50	32-bit reals
0.148	IBM 360/50	64-bit rea1s
0.013	IBM 360/50	32-bit reals, with slow core
0.068	IBM 360/50	64-bit reals, with slow core
0.284	KDF9	see 9.4
0.334	ICL 4130	2 μ store
0.170	ICL 4130	6 μ store
0.189	ICL 503	
0.167	RREAC	obsolete machine
0.804	CDC3600	Norwegian compiler
0.617	ICL 1907	2 μ store, obsolete compiler
1.24	Univac 1108	obsolete compiler, note speed improvement
1.12	CDC6600 version 1.0	(obsolete), note speed improvement
1.08	CDC6500/6400	version 2.0
2.95	CDC6600	version 2.0, 10% faster for no known reason
0.292	EL-X8	original compiler for Electrologica X8.

Caution: The mix figure should not be taken as a measure of processor performance without ensuring that the current version of the compiler has been used. It is important to remember that the processing speed of ALGOL 60 only is being measured. A difference in the mix figure of less than 20% should not be regarded as significant. The difference between the mixes is also illustrated in Fig. 2.1, page 90.

Appendix 9

This electronic copy

This copy was produced by scanning a printed copy and making minimal changes to re-format within the standard LaTeX book style. All floating elements have been renumbered, but the citations are not linked. The index has been redone and no attempt has been made to relate it to the original.

Typographical errors have been corrected, but the substance remains as in the original.

Using the original pages numbers, my record of the typographical errors are as follows:

Page	*Line no.*	*Was*	*Should be*
10	5	$<$	$>$
10	7	$\vee$	$\wedge$
10	7	$<$	$>$
10	21	$<$	$>$
12	-5	columns	rows
24	-5	AZ	JAZ
25	14		(Tab S3 to left)
42	-19		BLOCK 4 (under BLOCK 3)
48	-11.5	(new line)	102 0
49	14.5	(new line)	JIN,3 L2 Jump for deeper level
63	-12	word	work
70	-9	$\sum_i$	$\sum_j$
70	-9	LT_{j*}	LT_{i*}
70	-8	$\sum_j$	$\sum_j$
70	-8	nLS_j	nLM
72	19	nett	net
74	-5	would	could
75	13	rate	rate.
80	20	Trontheim	Trondheim
81	2/3	(Replace sentence by)	The optimisation is performed even if **step** 1 is included.
92	-1	TFA/	TFA/(

Page	*Line no.*	*Was*	*Should be*
96	-7	-17	17
101	17	That, is	That is,
104	8	text	text.
104	11	here.	here (see Table 2.10).
105	Entry 4.	+	+,
105	-8	GTA	; GTA
109	1	Fig. 1	Fig. 2
110	-17	Fig. 2	Fig. 3
120	12	(as	(at
127	17	par	par-
131	1	contain an error	contain errors
131	-8	or	of
136	16	used	use
140	13	or	for
142	-8	6.3)	6.3).
150	11	PL/1	PL/I
154	-4	re-put	re-input
156	-5	necessary	necessary.
158	3	call,	calls,
162	-16	ower	lower
165	-11	that	require that
170	2	he	the
172	-20	mean	means
173	(Diagram)	(first arrow on right)	(point to <b2>)
173	(Diagram)	(new arrow from second JMP **false**)	(point to LDA X)
173	(Diagram)	(align 3.0 with X)	
176	6	now	not
177	31	format-actual	formal-actual
177	-5	$\uparrow 0$	$\uparrow 0.0$
178	-1	is	in
185	7	Wieglosz	Wielgosz
195	-17	is not	was not
201	1	nett	net
205	3	:–	–
213	8	illustrates	1 illustrates
213	10	$[e2]'$	$[e2']$
215	-6	B4	B4;
219	3	multiple	multiply
219	19		(align CURRENT)
221	11	Manor	Man or
222	-8	optiminal	optimal
223	-4	KDF9	KDF9,
227	1	procedure	procedures
228	11	optiminal	optimal
228	22	nett	net
229	5	for	**for**
231	24	auto increment	auto-increment
233	12	for	**for**
233	14	for	**for**
234	-17	nett	net
234	-9	polish	Polish

Page	*Line no.*	*Was*	*Should be*
235	3	Estimate space	Estimated space
236	14	must contain	contains
236	-17	sequencies	sequences
238	1	the for	the **for**
249	-2	convenient	inconvenient
251	-9	(1966)	(1967)
256	16	language	language version
267	21	:= 1	:= -1
268	4	for	**for**
273	-12	I.F.E.	I.E.E.
274	Gries(1965)	use in	used in
274	Gries(1971)	Computer	Compiler
276	Lowry(1969)	(1969)	and Medlock, C. W. (1969)
280	10	report	Report
304	8	$1n$	ln
305	10	nett	net
305	(heading)	stuck	stack
307	20	1.1.41.	1.1.1.4
310	2	71 105	71 105/million
310	(2nd col, 712)	$\hat{10}$	$_{10}$
315	-7/8	(delete sentence)	
319	16	*Samset*	*Samet*

This misprints on page 10 were covered by an errata slip. The text above refers to complete words.

Index