An efficient system for user extendible languages

by MALCOLM C. NEWEY

University of Colorado
Boulder, Colorado

INTRODUCTION

The recent survey by Feldman and Gries on Translator Writing Systems ¹ devotes space to discussion of "extendible" compilers but fails to explain fully the motivation. The reasons for an extendible high level language (and hence for an extendible compiler) are as follows.

i) It is not feasible to foresee all the programming techniques which have not been developed at the time of language definition, and which might deserve a place in a language.

ii) It is not practical to incorporate all the data structures, operations and statement forms currently known into one colossal language since this language will be barely teachable, and implementation may be uneconomical or maybe impossible.

Extendible compilers have, of course, not been available and it has been shrugged off as a fact of life that most programmers constantly come across problems to which the language available is really not suited. The normal "solutions" of this difficulty are both unattractive. The first way out is to abandon the general purpose language for assembly code or change to some special purpose language (which may require a compiler being written for it). This line of attack has given rise to a multitude of special purpose languages such as string processors, list processors, etc. The other way out of the problem is to re-sign oneself to tedious, inefficient, complicated code, and is illustrated by the way many ALGOL 60 programmers incorporated complex variables into their programs.

i) represent each complex quantity by a 2 element array.

ii) operations between complex quantities are performed by procedures and so $z := (u + v)^* (u - v) / (u * v)$ is computed as follows:

add complex $(a, u, v)$;
subtract complex $(b, u, v)$;
multiply complex $(c, a, b)$;
divide complex $(a, c, u)$;
divide complex $(z, a, v)$;

This awkward dilemma can be avoided completely if we have a language with very strong basic structure (such as ALGOL) and a facility for extension by the user of the language. The problem with this scheme is that an extendible compiler must be devised. This is a problem of automating something that compiler writers are old hands at; that is, amending the compiler to recognize extensions to the base language.

There are two semi-automatic schemes for extendible compilers that deserve a mention:

i) Syntax Directed Compiler Approach

The Syntax Directed Compiler ² is a compiling system accepting first the translation rules of a language and then a source program in that language, to produce a target program according to the rules supplied. The system has great merit where researchers are experimenting with various source languages. The system could be made to process an extended source language by altering the translation rules appropriately,

ii) The Compiler-Compiler Approach

This genus of translator writing system (usually associated with Brooker, Morris, et al. ³) accepts a definition of a phrase structure language including the meaning of each of the phrases (in terms of the target language) and produces a compiler. As with the Syntax Directed compiler the description of the source language can be altered to accommodate the extended language.
This paper describes an approach to this subject which is claimed superior to each of these possible attacks and also superior to the approaches described by Cheatham, Galler, Perlis.

It must also be noted that although ALGOL 60 is used as a basis for discussion (as did Galler and Perlis, the notions can be extended to extendible compilers for other high level, powerful, general purpose languages such as ALGOL 68 and PL/1. Its use is justified by the fact that ALGOL 60 is more familiar.

Nature of useful extensions

It must be realized that if one has an automatic means of extending a compiler then there is precisely the same problem (as encountered with languages such as ALGOL) that led to the need for extendible languages. That is, it is probably not possible to be simple to learn and operate, readily implementable and completely general. This difficulty is offset, as will be indicated in this section, by the fact that there are a small number of discrete areas which contain most extensions. The difficulty is also offset by the fact that there are often many ways to extend a language but only certain of these can be said to be in "harmony" with the language, and it appears that these harmonious extensions are the most powerful and most easily describable. To illustrate this last point, consider a typical example of a feature to be added to ALGOL: string processing as used in SNOBOL.*

1) It is quite likely that someone wanting both string processing and arithmetic power could take the main SNOBOL statement (which is very powerful) and incorporate it in ALGOL. This would be syntactically recognizable provided each such statement was flagged by some symbol such as asterisk.

2) Alternatively one could revise the type string or ALGOL 60, add a concatenation operator, etc and introduce a statement of a form such as

\[
\text{search < string > for < string exprn> :=}
\text{<str exprn> else <Roman>}
\]

Comparing these two alternatives we note the following points in favor of the second which is, of course, in harmony with ALGOL 60.

i) Because string is added in the second case with the same status as real or Boolean, both string functions and strings as procedure parameters are possible.

ii) In the first case there will be confusion and contradiction of labelling conventions.

iii) Because the second implies string arrays, indirect addressing of strings is made simpler.

iv) Implementation of the first scheme is tantamount to including most of a SNOBOL compiler inside an ALGOL 60 compiler which is of course nasty for implementation.

The above example does emphasize that some methods of including a concept into ALGOL are in harmony and some are not. Furthermore, it illustrates that harmonious extensions make use of and work in with the various concepts of ALGOL that are well founded (the concepts of type, statement, label, variable, etc.).

It is with this principle in mind that the following basic methods of extending ALGOL-like languages are proposed.

Provision for additional data types

This was the technique that was used in the above example for incorporation of string manipulation and which made a very attractive extension. In fact it is probably the most powerful device possible since it opens large numbers of non-numeric fields to the scope of Algorithmic languages in a very simple manner. An example of the type complex was mentioned in the introduction and it may readily be seen that introduction of a type complex with the same status as real, integer, and Boolean is the only satisfactory way of introducing the concept of complex arithmetic.

A flood of other extensions to ALGOL 60 come to mind by means of this device of providing additional types. It is the author's opinion that the bulk of the shortcomings of ALGOL 60 that people feel at various times could be removed if they could define additional types. The following examples illustrate the wide application for this technique.

i) The type double real would enable double precision hardware, presently available on most current computers, to be utilized by a programmer.

ii) the type half real would sometimes be useful when space is at a premium rather than accuracy.

iii) the type polynomial would make polynomial manipulation programs (Hamblin, Collins etc.) quite redundant since the inclusion of the routines...
into the framework of ALGOL allows the more comfortable programming speech of the more sophisticated language.

iv) The types note or chord, etc., could be a great boon to the researcher in music by making music text manipulation available in the ALGOL framework.

The implications of additional types
The extensibility of ALGOL-like languages in the direction indicated in the previous section is obviously very powerful but in order that its full value may be realized the following additional types of extension must accompany any provision of additional types.

i) Additional significance for existing operators. If the concept of subtraction, for example, is applicable to the extra data type then the significance of the operator “minus” must be defined for operation on quantities of the new type.

ii) Additional Transfer Functions
Consider the case where the extra type is complex. Just as it is necessary for an inbuilt rule to be available for conversion of any integer quantity to the equivalent real quantity, so it is necessary that the transformation from real to complex be defined. The combination of these two transfer functions should automatically, then, be available for conversion from integer to complex.

There are actually two classes of transfer functions which I shall label “generalizing” and “degeneralizing.” It is readily seen that real may be considered a more general type than integer since each integer can be transformed to a real quantity without loss of information but most real quantities have no integer counterpart. Hence it is appropriate that the transfer integer → real is called “generalizing” and the transfer real → integer is termed “degeneralizing.”

It is obvious that “generalizing” transfer functions are essential for most new types. Wirth and Hoare express clearly the major argument for using operators and functions instead of degeneralizing transfer functions.

“. . . and he (the programmer) is thereby encouraged to select the alternative which corresponds to his real requirements, rather than rely on a built-in “default” conversion about which he may have only vague or mistaken ideas.”

The opposing point of view is that such functions and operators can clutter up a program. I don’t propose to add anything to this debate and so will just proceed anticipating either. Therefore we should allow extendibility in the direction of both generalizing and degeneralizing transfer functions.

iii) Additional Operators
It is readily seen that if entirely new data types were introduced then new ways of combining quantities of those types could also be required. For instance if the new type were complex then monadic operators ip and rp suggest themselves for extracting real and imaginary parts of a complex quantity. Similarly, if the new data type were polynomial then a diadic operator deriv would be useful for taking the derivative of a polynomial with respect to some variable.

iv) Additional Forms of Constant
For every type in a language there must be some format, for a primary of an expression, which can be used to specify constants of that type. The types of ALGOL 60 include real and Boolean and the corresponding constant forms are numeric constants such as 5.2, 123.4 and logical constants true and false.

Provision for additional statements
This is another very powerful way of grafting onto the framework of any general purpose language additional special purpose features. Because harmonious extensions are ones which make the most of existing parts of the compiler, it is reasonable to specify that the components of the statement are either basic symbols or the various important syntactic units already found in the language. These are the components of existing statements and include <statement>, <variable>, <Boolean expression>, <array>, <label>. The delimiters would be chosen from the standard set of the language or manufactured with some mnemonic significance. The following example of a parallel processing statement illustrates that extra statement forms can be the only convenient solution for some tasks. The construction “do <statement> also <statement> . . . also <statement>”; is almost essential for making use of parallel processing hardware via ALGOL 60.

A language for extensions to ALGOL
The motivation for a compiler expansion
scheme and the directions in which extensibility is required have now been expounded. The extension scheme described in this paper consists of two programs. The main one is the skeleton of a compiler without information about types, operators, etc. The second part of the system is a program which will accept as input the information necessary to describe the extensions desired to the language and use this information to pad out the skeleton compiler to accept and compile any programs written in the new extended language. The information that the extender program requires can be considered to be a program in the language LACE (Language for ALGOL Compiler Extension). A LACE program will be a sequence of statements—each one being a specification of some new addition to ALGOL. There will, of course, be statements associated with each of the directions for extension that were described in PART II.

The following ten LACE statement forms are proposed to allow descriptions of extensions to be specified.

- **TYPE**
- **CONSTANT FORM**
- **MONADIC OPERATOR SIGNIFICANCE**
- **DIADIC OPERATOR SIGNIFICANCE**
- **GENERALISING TRANSFER FUNCTION**
- **DEGENERALISING TRANSFER FUNCTION**
- **OPERATOR PRIORIT Y**
- **STATEMENT FORM**
- **BASIC SYMBOL**
- **CONTROL ROUTINE**

Each of these statements or functions require parameters and so a convenient form for each statement is its name (either full name or abbreviated) followed by the parameters required. These parameters will be separated by commas. A description of the parameters required follows.

i) **TYPE** <name>, <length>

To add a new type to ALGOL, a name, which is a new basic symbol, must be specified so it can be used in declarations, specification parts etc. The parameter 'length' is required to specify the amount of memory space that is required to contain the value of any quantity of that type. This will normally be a number of bits required but for certain types of data, such as polynomial, a variable length field is required for values and so instead of an integer another symbol such as 'V' (for variable length) would be used.

ii) **GTF and DTF** <initial type>, <final type>, <code>

The parameter 'CODE' is the body of code that is required to transform quantities of the initial type to quantities of the final type. The body of code must be surrounded by appropriate symbols. We will use "and".

iii) **MOS** <operator>, <T1>, <TR>, <code>

and

**DOS** <operator>, <T1>, <T2>, <TR>, <code>

In the statements to define new operators T1 and T2 are the names of the types of the arguments of the operator. TR is the name of the type of the result of the operation and the parameter 'CODE' is the body of code necessary to perform the operation.

iv) **Operator priority** <operator>, <priority value>

The priority value for an entirely new operator would be obtained by referring to a list of priorities for use with the system. The compiler requires an ordering of priorities for all operators.

v) **Constant Form** <type>, <syntax>, <semantics>

Each constant form for a particular type in the language will have some syntax associated with it. The form of syntax strings in LACE is discussed later. The semantics required must be target code that will generate a constant of the appropriate value for use by the program.

vi) **Basic Symbol** <symbol>, <internal value>

This statement can be used when a particular value is desired for new basic symbols in the language. It is also useful where equivalent symbols are desired, as is the case with some versions of ALGOL—the underlined words can be in several languages.

vii) **Statement Form** <syntax>, <semantics>

Refer to the later section on syntax and semantics strings.

viii) **Control Routines** <code>

This statement is to allow the specification of a routine that will be available at run-time of an ALGOL program.
The strings that represent syntax

To represent the syntax of statements and constants we use an extension of Backus Normal Form in which the following nine symbols are reserved as meta-symbols.

```
< >  for surrounding syntactic units.
|   for use as a choice operator.
{   for denoting a sequence of zero or more repetitions of the enclosed string.
[   for bracketing a series of choices together.
“ ” for delimiting the whole string.
```

These characters will not be available on all computing equipment but for any given system suitable replacements can be made. The extensions to BNF are minor and should be readily understood with the aid of an example. The syntax string for the conditional statement of ALGOL 60 follows

```
"if <Boolean expression> then <unconditional statement> [else <statement> | <nil>]"
```

It has previously been mentioned that the syntax string is composed of the syntactic units of ALGOL and basic symbols, either new or old. This is, of course, an oversimplification. Firstly, there are many syntactic units in the ALGOL 60 Report that are included merely as steps in the development of the syntax; terms such as 'Boolean secondary' or 'compound tail' are units we need not consider useful in this scheme. Secondly, the grammar should be unambiguous and all developments of the syntax string should be recognisable in a left to right scan; this means that such syntax strings as

```
abc [ <variable> | <expression> ];
```

are made illegal because it is both ambiguous for `abc alpha` and involves backup for `abc beta [1+m×n]+2`;

Thirdly it should never be the case that two syntactic units can be run together, either directly or indirectly, to remove further difficulties, or even ambiguities, for the compiler. Hence it is not allowed to write a syntax string

```
pqr <variable> <integer> xyz or the string
pqr <variable> [lmn] <integer> ] xyz
```

The strings that represent semantics

For the LACE statements for defining new forms of statements in ALGOL and new forms of constant, there was a parameter for denoting the syntax and a parameter for denoting the semantics. This pair of parameters forms a translation rule just as in the Syntax Directed Compiler or the Compiler-Compiler. Just as the nine symbols

```
" [ ] { } < >
```

were reserved for use as meta-symbols in syntax strings, so they are reserved here. Of course, there is no syntax units in the semantics string so the symbols and are used in constructing semantic units. The symbol * is also reserved for this purpose. It should be clear that the semantics string will have an identical structuring of the metasymbols " " [ ] { } to the corresponding syntax string. Apart from the metasymbols the semantics string is made up of a sequence of semantic units. There are 3 types of semantic units as follows:

i) A semantic unit may be any symbol of the target language. Special conventions will have to be made for representing symbols which are reserved as metasymbols.

ii) A semantic unit may be a pointer to one of the syntactic units in the syntax string and has the form `<integer>`, for example, `<6>` or `<19>`

The syntactic unit to which it refers is that one with ordinal number equal to the quoted integer. The target code that this initiates is the code normally associated with the syntactic unit. This code is well defined for such units as `>` statement>, `<expression>` but for others like `<label>` would be empty, and so a pointer to `<label>` would be useless.

iii) A semantic unit may also be a call of a system function designator which will have certain parameter requirements. A call on a function has the following form.

```
*function name (<parameter list>)
```

The parameters which form the list will normally be pointers, symbols or integers. There will be
functions for all the attributes that the various syntactic units have and so an example of a function is

*ADDRESS (<2>) which gives as a result the address of syntactic unit number 2 in the syntax string which may be <variable> or <label>.

The effectiveness of the system will, of course, depend to a great extent on a suitable set of functions. However, experience derived from an actual implementation would affect the choice so much that no more than a few examples are included in this paper.

A simple illustration of the correspondence between the syntax strings and the semantics strings is now presented in such a way as to demonstrate an instance of the usefulness of LACE. Many programmers realize that the time spent in duplicate address calculation in the following types of statement is wasted but there is no way out in current ALGOL.

\[
xyz[i, j, k, l] := xyz[i, j, k, l] + 1;
\]

With a compiler extension scheme available the frustrated programmer could write a more efficient and more suggestively worded

\[
\text{step } xyz[i, j, k, l] \text{ by } 1;
\]

The following text would be a suitable LACE definition.

NSF ("\text{step} <variable> <comma> <expression>",

*load address (XR1, <2>)
*fetch indirect (XR1) <4> *apply diad(+, *type (<2>), *type(<4>))
/store indirect (XR1)"

where XR1 may be an index register.

A second and more complex example is now given which is the definition of a statement to permute the contents of any number of variables.

NSF ("\text{rotate} <variable> <comma>

{ <variable> <comma> } <variable>",

*load address (XR1, <2>)
*fetch indirect(XR1)
*load address (XR2, <4>)
*fetch indirect(XR2)
/store indirect (XR1)
/store indirect(XR2)"

In the above example the string CPYREG is an instance of the object code output required. In both examples some notional stack is assumed.

Implementation

This part is concerned with the design of an ALGOL compiler in such a way that it will be extendible; obviously special considerations are required in formulating its structure. ALGOL, in particular, is chosen because of the difficulty of talking in the abstract about translation techniques. The schemes are modifications of well known ones and consequently much familiar material is presented, but only in the interest of a clear description.

Only two areas of the system will be discussed; these are the ones that present most of the challenge. These areas are expression analysis/compilation and statement analysis/compilation. Other areas affected are the basic symbol input routine, simple variable declarations, array declarations and procedure specification parts.

It should be noted that the following are demonstrations that the problems that arise in implementation are soluble. It may be that the techniques proposed are not immediately applicable to certain languages which are not ALGOL-like or not immediately compatible with certain optimisation requirements for the compiler. The proposals should, in any case, suggest the method of revision for alternate techniques.

Expression analysis routine

The technique that seems most convenient is the operator priority technique for translation to Reverse Polish notation. Early references to the
method are DIJKSTRA, HAMBLIN and HIGMAN (all appeared in 1962).

The technique is normally described in terms of an input string (in source code) and an output string (in target code). At the start of an expression a left parenthesis is put on the operator stack to protect the contents of a previous recursion and if the first symbol is "if" then appropriate action is taken; otherwise there is a normal simple expression. All primaries encountered during the left to right scan generate output code immediately (including sub-expressions) but operators may be stacked according to the following rules.

i) Operators which can be either monadic or diadic are treated as diadic if they follow a primary, else they are treated as monadic.

ii) Operators which are monadic are added to the operator stack, provided there is not an operator of higher priority on the stack. This condition would correspond to a failure.

iii) Operators which are diadic must occur immediately after a primary or else an error exists. Also a monadic operator or another primary must follow. Diadic operators first displace each operator on top of the stack which is of equal or higher priority. It is then added to the stack with the current value of the variable, BETA, which indicates the type of the left argument of the operator.

iv) An expression terminator displaces all operators until it finds a left parenthesis which it removes. It then causes exit from this level of the expression routine indicating that it found an expression of type given by the value of BETA.

v) A monadic operator is displaced as follows:—The type of quantity to which the operator is to be applied is given by the variable BETA. The appropriate operator significance for that operator and the type of quantity is then added to the output. The type that is given is the type of the result of the operation is placed in BETA and the operator removed from the operator stack.

vi) A diadic operator is displaced as follows:—The type of the 1st argument of the operation is stored with the operator and the value of BETA gives the type of the second argument. The operator significance appropriate to that operator and that pair of types is selected and added to the output. This may be one of the operator significances specified by the programmer or may be one which was specified, but compounded with one or more generalising transfer functions.

The expression analysis routine described above is one pass, left-to-right scan and does not produce optimum code. The ideas could, however, be extended to other expression analysis techniques.

It has been noted earlier that there are transfer functions which are defined for the extended language both explicitly and implicitly. If a new data type abed is defined and a generalising transfer function from type real to type abed is also defined, then a new generalising function is implicitly defined from type integer to type abed.

It is clear that it is more efficient for the extender program, after having absorbed all the transfer functions which are explicitly defined, to generate, as well, all the other transfer functions that are implicitly defined. New generalising transfer functions are generated as minimum combinations of explicit generalising transfer functions. Implicit degeneralising transfer functions are generated as minimum combinations of explicit degeneralising functions or as the combination of a generalising transfer function (which need not be explicitly defined) and a string of degeneralising transfer functions.

It is also profitable for the extender program to generate, and form as a list in the compiler, all operator significances which are implicit. An implicit operator significance is formed when a generalising transfer function is applied to convert the quantity to an appropriate type for some operator significance, which is given for the operator, to act on.

Statement analysis

The extender program has the task of constructing, from the syntax and semantics strings which are supplied, that part of the compiler which will analyse the statements of the extended-ALGOL programs. We shall talk about this process assuming that the compiler is constructed along the lines commonly known as syntax routine method of compiling. This means that wherever a syntactic unit appears in a syntax string this transforms into compiler code as a call on that syntax routine. The syntax routine will scan the code till it exhausts that syntactic unit.

A basic symbol which appears in a syntax string will be transformed to a test or a check in the compiler code generated. Because the choice operator, | , separates alternatives, a series of tests
must be made with one branch made to a section of code for each alternative. At the end of each of the alternatives, there must be jumps to a common point.

The transformation of a pair of braces \{ \} which denotes optional repetition, is a loop in the compiler code. The contents of the braces, when transformed, form the body of the loop. The decision to jump out of the loop is made at the start with a test to see whether the next symbol to be fetched from the input is appropriate to continuing or discontinuing the loop.

The above is a description of how the syntax string imparts structure, tests and syntax routine calls to the code generated for the compiler. The semantics string, of course, has the same structure and so where the tests and routine calls are made for some particular element of the syntax string, the corresponding element of the semantics string is designated for output.

CONCLUSIONS

This paper has been an introduction to a solution of the problem of general purpose languages being inflexible; a two segment system which allows the user to introduce new data types and new data manipulations. The scheme is intended for high-level general purpose languages to make them more general but was illustrated with ALGOL. A generalisation to other appropriate languages should not be difficult.

There are several questions that should be asked of this compiler extension scheme. The first of these is whether the direction of extra data types and extra operations on these types is the right way to provide for extensions. The second question is whether it is ambitious enough (i.e., powerful enough), and the third is whether it is feasible (i.e., not too ambitious). The author's opinion on these questions is that they should all be affirmed. However, the areas are subjective since there has been no implementation of such a system yet, but the contents of the paper explain and give weight to the point of view.

A fourth and less subjective question asks how it compares with rival schemes. We compare it to four other systems which can be used to provide compilation of extended languages.

1) Syntax Directed Compiler (SDC).

The use of a SDC for compiling an ALGOL-like language requires a standard primary 'program' which defines the whole syntax and semantics of the language, rigorously and completely.

To make use of SDC for user extensions, the primary program would require suitable alteration. This would demand knowledge of the target code (as does the scheme here presented) but also demands quite intimate knowledge of the primary program—both its language and structure. This is probably significantly easier than amending a hand coded compiler but suffers from the same disadvantages.

ii) Compiler-Computer (CC)

This is also a semi-automatic method of extending a compiler and suffers the same disadvantages as SDC.

iii) Cheatham's approach

This is an attempt to provide macro facilities for high level languages, and, in the terms of this paper, only in the area of new statement forms. That is, no direct attack is made on the area of provision of extra data types, which is crucial. It is readily admitted that macro facilities are very useful and a powerful macro preprocessor such as LIMP (WAITE 15) can help to remove some of the problems of tedious programming speech. However, it is also true that if it is impossible to write code for some given task (in some language) that will be compiled both efficiently and compactly, then macro processing is not going to help. It is contended, therefore, that the present paper has a better answer as regards efficiency.

iv) ALGOL C—Galler, Perlis

As with the above scheme (Cheatham), considerable importance is given to provision of macro facilities in ALGOL-like languages. For the same reasons it is potentially less efficient than the approach of extended compiler generation. Recall it was mentioned in the introductory section, that the case of complex numbers is a classic example of tedious and inefficient ALGOL; yet Galler and Perlis still have complex numbers implemented as \(2\) element arrays. On the other hand their approach means that extensions are not machine dependent.

A second disadvantage is that it stops short of providing new statement forms. A last and quite significant disadvantage is that all the definitional facilities are in the compiler and therefore the same work is done over and over again, wastiw time and compiler space.

REFERENCES

1 J FELDMAN  D GRIES
Efficient System for User Extendible Languages

2 E T IRONS
The structure and use of the syntax directed compiler

3 T CHEATHAM
The introduction of definitional facilities into higher level programming languages

4 B GALLER A J PERLIS
A proposal for definitions in ALGOL
Comm ACM 10 April 1967 pp 204–219

5 R A BROOKER et al.
The compiler-compiler
Annual review in Automatic Programming Vol 3 1963 p 229

6 D J FARBER et al.
SNOBOL A string manipulation language
J ACM 11 Jan 1964 p 21

7 C L HAMBLIN
An algorithm for polynomial operations
Computer Journal 10 Aug 1967

8 G E COLLINS

PM A system for polynomial manipulation
Comm ACM 9 Aug 1966 p 578

9 N WIRTH C A ROARE
A contribution to the development of ALGOL
Comm ACM 9 June 1966 pp 413–432

10 E W DIJKSTRA
Making a translator for ALGOL 60
Annual Review in Automatic Programming Vol 3 1963

11 C L HAMBLIN
Translation to and from Polish notation
Computer Journal 5 Nov 1962 p 210

12 B HIGMAN
Towards an ALGOL translator
Annual Review in Automatic Programming Vol 3 1963

13 B RANDELL D J RUSSELL
ALGOL 60 implementation
Academic Press Inc LONDON 1964

14 P NAUR
Revised report on the algorithmic language ALGOL 60
Comm ACM 6 Jan 1963 pp 1–17

15 W M WAITE
A language independent macro processor
Comm ACM 10 July 1967 p 443