The design of the MAX macroprocessor

D. Nudds

Computing Laboratory, University of Bradford, Bradford, West Yorkshire BD7 1DP

A new macroprocessor has been designed for general purpose text manipulation. The system aims to provide the user with a completely flexible system free of any syntactic rules for the specification of macro calls. Although the lack of restriction implies greater system overheads in the form of more extensive scanning and comparison of the scanned text with the set of defined macros, a block structure together with a tree structured macro definition list enable a practicable working system to be implemented which is not unrealistically slow.

(Received October 1974)

1. Introduction

In this article we consider the specification and application of a macroprocessor MAX. Work on this system was motivated by the following considerations.

First, a practical system was required which could be used for a variety of purposes: particular examples include:

1. The development of high level machine oriented languages via a macroprocessor. Macro definitions were required which could be applied to a source text having a relatively complex syntax.
2. Work on the translation between assembly languages of different machines. Here the syntactic structure of the source text undergoing processing is somewhat simpler, but the semantics had to be taken into account in defining appropriate macros, so again a powerful and comprehensive macroprocessor was desired.
3. Further work in the field of applications oriented languages indicated a requirement for a system with easily written macro definitions and free of restrictions concerning input or output conventions, character sets and formats.

A second motivation was to examine the role of a formal definition in producing a simple system which would be not only easy to use but relatively straightforward to implement. Actual experience in building the MAX system was not, of course, a matter of proceeding in a linear progression from formal definition through implementation to application: use of the system led to developments in the implementation and hence to the underlying formal model. But a formal description gives a clear, succinct statement of the system, which is readily modifiable, and encourages a modular implementation. This formal description will be found in the Appendix.

The basic principle of all macroprocessors, whether called macroassemblers or macrogenerators (special or general purpose) is the substitution of symbols within a text, presented as data to the processor according to rules specified by macro definitions. A macro definition is a string of symbols too, which may be incorporated in the same test. The simplest macroassemblers have the restriction that these definitions may not themselves be processed. A general purpose processor, on the other hand, allows the conversion process to apply to arbitrary strings in the text, and the conversion of textual macro definitions may take place.

General purpose processors, such as GPM (Strachey, 1965), the TRAC system (Mooers, 1966), and ML/I (Brown, 1967), usually have the property, also present in macroassemblers, that the macro calls are made by placing the name of the macro prior to the parameters on which the macro expansion is to be carried out. The system we have adopted, paralleling the LIMP system and STAGE2 (Waite, 1967; Waite, 1970), is more general; effectively a distributed macro name is used, and template matching is required of the macro definition against the text being processed. At the same time in MAX, no assumed conventions are built in regarding the parameter names or abbreviations. This is achieved by distinguishing between the internal representation of the individual atoms, or symbols, which are macroprocessed, as against their external representation, whereby a symbol is represented by a string of characters. Externally symbols are delimited by a specified separator character. This separator itself may be redefined during processing.

The MAX system presents, then, an attempt to overcome the restrictions which Brown (1971) describes as being imposed on the macroprocessors dealt with in his survey. There is no fixed start or end to macro calls, as in STAGE2 for example. Nor are the start and end of the macro required to be explicitly defined, as in ML/I and GPM. At the same time, the matching mechanism is implemented with a tree structured macro definition set so as to give an acceptable processing speed. The user however need not be aware (as he should be with STAGE2) of the storage system. He simply considers his macro definitions as a chronologically ordered list.

Certain predefined control symbols are necessary in any system. In MAX there is a group of about 30 such symbols; their external representations all begin with a special character (e.g. '£'), which again may be replaced by any other character at any stage if required. Thus the system has about the same level of flexibility of character set over which it might operate as ML/I, except that the MAX system has just one 'punctuation' rule: a predefined separator character separates symbols.

2. Matching with MAX

At any stage in the processing of a string by macroprocessors there will be a set of macro definitions which may be applied to the string. These definitions are effectively rewriting rules: the whole process in fact may be considered as a form of Markov algorithm. The general Markov algorithms allow for repeated scanning of a string, on each scan the first rewriting rule being applied. In contrast, macroprocessors generally are essentially one-pass systems in which the string is passed through the processor just once, and the rewriting is performed as (in general) keywords are recognised which cause the application of the rewriting rules.

Such systems may always be considered as Markov algorithms by suitable expression of the rewriting rules in such a way that they incorporate a marker which corresponds to that point up to which the input string has currently been scanned.

For example, suppose we consider a simple system in which the alphabet **A** consists of the letters $A, B, \ldots Z$. Suppose there is a single macro which implies that if the current scanned symbol is an A then it and the following symbol will be interchanged, and that scanning will then continue with the next

symbol after the rewritten A. That is the production

$$A\,x \rightarrow x\,A \quad \text{for all } x \text{ in } \mathbf{A}$$

is to be applied but the string must not be rescanned so as to cause the interchange of the rewritten A with the next letter.

The above conditions can be applied in a general Markov algorithm by adding a special marker symbol to the alphabet. Here we shall use '□' as the marker. Incidentally we shall ignore the conventional use within Markov algorithms of a period symbol to indicate a terminating rule. Such explicit terminating rules will be dispensed with in our production schema, as termination instead will be indicated by the non-applicability of any of the production rules: this is equivalent to having a single production containing the terminator symbol as the last one of the set of productions.

First the initial string to be processed is preceded by this marker (which may again have been indicated explicitly by an appropriate production, but instead we shall adopt this as a convention).

The productions are now written:

$$\square\,A\,x \rightarrow x\,A\,\square \quad \text{for all } x \text{ in } \mathbf{A}$$
$$\square\,x \rightarrow x\,\square \quad \text{for all } x \text{ in } \mathbf{A}\ .$$

The second production moves the marker through the string whenever the first production cannot be applied, i.e. acts as a marker.

In the above macro expansion we had a highly simplified example of the standard form which most macro processors use: the keyword followed by a list of arguments causes an expansion according to the macro definition. The next piece of text will then be scanned.

More generally, then, simpler macro processors may be defined by Markov algorithms in which the production schema are of the form:

$$\square\,A\,B \ldots I\,a\,b \ldots j \rightarrow \alpha\,\square$$

where $A \ldots I$ are fixed symbols in the alphabet, $a \ldots j$ are variables in the alphabet, and α is a string containing the variables $a \ldots j$ and some fixed symbols in the alphabet. The movement of the marker prevents repeated scanning.

The system which we have adopted for macro expansion is, however, quite different. Rescanning is permitted, because all the standard macros are defined by schema of form:

$$\alpha\,\square\,x \rightarrow \square\,\beta$$

where both α and β are arbitrary strings of symbols (fixed or variable) in the alphabet, and x is a single symbol. In addition the production schema

$$\square\,x \rightarrow x\,\square \quad \text{for all } x \text{ in } \mathbf{A}$$

occurs as a means of moving the marker one symbol to the right. Productions of the form $\alpha\,\square\,x \rightarrow \square\,\beta$ we shall call 'macros' and write them $\alpha\,x = \beta$, for all strings α and β and symbols x.

It follows that the matching mechanism required for this schema is one in which symbols are compared one by one working from right to left, starting with the symbol immediately to the right of the marker. For example with a macro $A\,x\,B\,y\,C = x\,y\,D$ i.e. a production schema:

$$A\,x\,B\,y\,\square\,C \rightarrow \square\,x\,y\,D \quad (\text{for all } x, y \text{ in } \mathbf{A})$$

and the string

$$A\,B\,E\,A\,\square\,C\,A\,B\,E\,;$$

first the C's to the right of the marker are compared, then the A in the string is matched with the y in the definition. Since y may stand for an A, the comparison is again successful, and so next the E in the string is compared with the B in the definition. They differ, so the match of the string with this production schema is unsuccessful and the string will not be transformed.

If all the macros in the environment similarly fail to match, then the production schema

$$\square\,x \rightarrow x\,\square$$

applies and the string becomes $A\,B\,E\,A\,C\,\square\,A\,B\,E$. After this transformation a further set of comparisons are now made to determine whether any of the macros may be applied to the new string.

Comparing the individual symbols from right to left in this way might seem at first hand to be an awkward and unnatural system. It does, however, give a neat and simple method of dealing with the repeated application of macros. For example, with the macros

$$X = A$$
$$Y = B$$
and $$Z = X\,Y$$

the text	$\square\,Z$	is converted first to
	$\square\,X\,Y$	then to
	$\square\,A\,Y$	then to
	$A\,\square\,Y$	which becomes
	$A\,\square\,B$	and finally
	$A\,B\,\square$.	

This description of the action of a macroprocessor in terms of a marker moving through a single text has, it is felt, the advantage of simplicity. At least when a user of the system is uncertain as to the effect of a complex set of macros, he can work through an example in the manner shown above. It might be pointed out, incidentally, that similar systems could be applied to the description of other macroprocessors. GPM, for example, effectively requires the marker to move up to the closing symbol (a semicolon for example) of the macro call. ML/I, on the other hand, requires the marker to move as far as the opening symbol (the macro name) of the call. The MAX system thus can be said to use call-by-value for its parameters, like GPM, as opposed to call-by-name as in ML/I.

Implementation of the MAX system may model the description given above by means of a pair of stacks, one for the text on each side of the pointer. No further stacks are needed for various levels of recursion as with other systems.

3. Definitions and blocks

In the above examples the individual symbols are represented by single characters. A more powerful general system is in fact used in MAX whereby groups of individual characters represent symbols. A single (but user definable) character is used to separate symbols. Here we shall use a space, or a string of spaces, as a separator. Special symbols are used for system control and for other predefined purposes. For example, **BEG** starts a block, **DEF** starts a definition, **+12** represents the positive integer twelve, and Φ represents a variable (universal symbol) within a definition.

The definitions of the previous section may then be written:

DEF Z **IS** $X\ Y$ **ENDD**
DEF X **IS** A **ENDD**
DEF Y **IS** B **ENDD** .

The order in which these definitions are presented to the processor is critical here. Reversing their order would cause the expansion of X and Y within the text of the first definition, whilst the definition is read in by the macroprocessor. To shield this from expansion on input, one of several possible scope restricting symbols may be used. The simplest way, perhaps, is to place the text of the macro within a block:

DEF BEG Z **IS** $X\ Y$ **ENDQ ENDD**

or, in abbreviated form:

MAC Z **IS** $X\ Y$ **ENDM** .

Within the brackets **BEG . . . ENDQ** the text is evaluated using only the macros defined within the block (together with global, system defined macros). Thus in the text

□ **DEF BEG** *Z* **IS** *X Y* **ENDQ ENDD**

the marker is moved from left to right, and within the block no evaluation takes place. When the marker reaches the symbol **ENDQ,** the text becomes

DEF *Z* **IS** *X Y* □ **ENDD**

and this causes the definition $Z = X\ Y$ to be created and stored at the current block level.

Alternative brackets are the pair **BEG . . . END**, with the same effect as **BEG . . . ENDQ**, except that additionally on reading the end-of-block symbol **END**, the marker is moved back to the beginning of the text enclosed within the block, and the text is re-evaluated at the outer block level (the brackets being removed) with all macros defined within the block discarded.

The following simple example illustrates some successive stages in the processing of text with two block levels. Some intermediate stages where the marker simply moves from left to right are indicated by the broken lines.

The following simple example illustrates some successive stages in the processing of text with two block levels. Some intermediate stages where the marker simply moves from left to right are indicated by the broken lines.

□ **BEG DEF** *A* **IS** *B* **ENDD** *A B* **BEG** **DEF** *B* **IS** *A C* **ENDD** *A B C* **END** *B* **END**

BEG DEF *A* **IS** *B* □ **ENDD** *A B* **BEG** **DEF** *B* **IS** *A C* **ENDD** *A B C* **END** *B* **END**

----------→

BEG □ *A B* **BEG** **DEF** *B* **IS** *A C* **ENDD** *A B C* **END** *B* **END**

A macro ($A = B$) is now applicable in the outer block. However when the marker enters the inner block, the macro is inapplicable.

BEG *B B* **BEG** **DEF** *B* **IS** *A C* □ **ENDD** *A B C* **END** *B* **END**

----------→

BEG *B B* **BEG** *A* □ *B C* **END** *B* **END**

---------→

A macro ($B = A\ C$) is now applicable in the inner block.

BEG *B B* **BEG** *A A C C* □ **END** *B* **END**

----------→

Now, in the next, outer, block level again:

BEG *B B* □ *A A C C* *B* **END**

The macro *A* The macro $A = B$ leads to

BEG *B B* *B B C C* *B* □ **END**

----------→

and then to

□ *B B* *B B C C* *B*

and so on according to the macros defined in the now current block.

Parameters within a macro definition are represented by:

(*a*) the universal symbol Φ on the left-hand side of a definition—the template;

(*b*) the symbols **1, 2, 3** . . . in the right-hand side of a macro definition—the macro body.

For example with a macro defined by

DEF A Φ *B* Φ **IS 4** FFF **2 ENDD**

the text . . . *A X B* □ *Y* . . . will be matched against the template. The symbol **2** in the macro body represents the second symbol of the matching text, whilst **4** represents the fourth symbol. Hence the text is replaced by . . . □ *Y* FFF *X* . . .. The pointer is positioned ready for re-evaluation of the generated text. This text will now be processed as required by the definitions of the current block.

Supposing, however, it were required to prohibit such a re-evaluation by an existing definition. One way of doing this would be to surround the above text by the block-quote symbols **BEG . . . ENDQ**. That is, the macro is required to generate the text . . . □ **BEG** *Y* FFF *X* **ENDQ** . . .. The macro needed for this is $A\ \Phi\ B\ \Phi$ = **BEG 4** FFF **2 ENDQ**. However, if the textual definition

□ **DEF** $A\ \Phi\ B\ \Phi$ **IS BEG 4** FFF **2 ENDQ ENDD**

is presented to the macroprocessor, it will be converted into

DEF $A\ \Phi\ B\ \Phi$ **IS** **4** FFF **2** □ **ENDD**

before the macro is defined and stored away. In several other macro systems, this sort of problem is resolved by allowing for quoting brackets equivalent to our **BEG . . . ENDQ** to be nested, and for the processor, on each scan, simply to strip off the outer layer of brackets. This is not possible with the **BEG . . . ENDQ** symbols, however, as processing is required to continue inside the brackets. Indeed these symbols are not the standard 'skip-quotes' or literal brackets, and so an alternative quoting system is employed in addition.

In the MAX system each symbol is designed to be treated uniformly. Instead therefore of introducing further special separate symbols which would have the effect of suspending the normal effect of the processor, the system incorporates the quotation mark within the symbol itself. We indicate this quotation property in our description by following a symbol by apostrophes equal in number to the level of quotation applied. Absence of apostrophes is equivalent to a zero-level quote.

Quoted symbols of this form are regarded simply as further symbols in the alphabet of MAX. A quoted control symbol like **DEF′** is not in itself a control symbol. Likewise quoted parameter symbols such as **2′** are not parameter symbols.

A quote-level is dropped from a symbol at the time when a macro is applied. Hence the text

DEF $A\ \Phi\ B\ \Phi$ **IS BEG′ 4** FFF **2 ENDQ′ ENDD**

causes the macro $A\ \Phi\ B\ \Phi$ = **BEG′ 4** FFF **2 ENDQ′** to be stored. When this matches the incoming text, . . . *A X B* □ *Y* . . ., then the text is changed to . . . □ **BEG** *Y* FFF *X* **ENDQ** . . . which in turn becomes . . . *Y* FFF *X* □ . . . the **BEG** and **ENDQ** having had the required effect of shielding the enclosed text from matching against any previously defined macros.

As a further example of the use of quoted symbols, consider the situation in which we wish to create definitions using a different notation. It may be required to write, for example, DEFINE NEXT = 1; in place of **DEF** NEXT **IS** 1 **ENDD.**

This may be achieved using the macro defined by

DEF DEFINE Φ = Φ; **IS DEF' 2 IS' 4 ENDD' ENDD** .

4. Further facilities

Additional control symbols are defined which allow for binary representation of integers, and arithmetic operations upon them. Output onto a special output stream is controlled by further symbols; such facilities are required because the nature of the scanning process implies that symbols cannot be automatically output during macroprocessing. Only at the termination of the process may it be assumed in general that symbols to the left of the pointer cannot match a stored macro definition. Hence if intermediate output is required, which is highly desirable not only for storage economy but also for debugging purposes, then it must be demanded explicitly by the programmer.

A full description of all the control symbols which have been built into the current implementation of MAX is to be found in the Appendix. These symbols may be regarded as calls to macros which are valid within all block levels: these macros are in effect defined in a special block, the *outer block*. The user, too, can also define macros in this outer block, and thus build up whatever special control symbols he requires. These special symbols are recognised by the MAX processor by the initial character of their external representation, just as are the built-in control symbols.

In the following examples will be seen the use of several of these control symbols.

5. Examples

First, we consider the conversion of an algebraic expression into postfix form.

It is simple to write macro definitions in MAX which will convert a fully parenthesised algebraic expression into postfix form, for example $((A + (B * C)) * D)$ into $A\ B\ C * D * +$. A number of macros of form $(\Phi\ \theta\ \Phi)$ = **2 4 CONC** θ **CONC** are required, where θ stands for one of the arithmetic operators. These macros may be defined by a set of macro definitions such as, for example,

MAC (Φ * Φ) **IS 2 4 CONC * CONC ENDM**

and similar ones for each operator.

The effect of the **CONC** control symbol is to concatenate the characters representing the two previous symbols into a single symbol. However it is only applicable in the case where the two previous symbols are unquoted character strings. Otherwise it has no effect (but remains in the text). Thus, for example, **2 4 CONC** is not changed when input as part of the definition so the **CONC** symbol does not need to be quoted.

The template matches only when a pair of parentheses contains just two operand symbols separated by an operator symbol. The string in the example is scanned until the pointer reaches the symbol *C*, i.e.

((*A* + (*B* * □*C*)) * *D*)

which is converted to

((*A* + □ *B C* **CONC** * **CONC**) * *D*) .

This in turn becomes

((*A* + □*BC* * **CONC**) * *D*)

and then (*A* + *BC* * □) * *D*)

which becomes (□ *ABC* *+ * *D*)

similarly, and finally *ABC**+ *D**.

The macro definitions required for this need not be explicitly written in the text. Instead they may be generated by the macro defined by

MAC *GEN* Φ **IS MAC'** (Φ **2** Φ) **IS**
2' 4' CONC 2 CONC ENDM' ENDM

together with the calls

GEN + *GEN* − *GEN* * *GEN* /

which will produce the required macro definitions.

This example has assumed that the initial source text consists of symbols which are represented by single characters, whilst it forms a final text which is a single symbol represented by a character string. The control symbol **DSEP** allows redefinition of the separator character, or for symbols to be represented by single characters. We shall see a use of this in the next example.

Suppose, as part of a macro assembler, we wish to convert a line of form

MOVE *aa . . . a, bb . . . b* into the text
LOAD *aa . . . a*
STO *bb . . . b* .

Here the newline will be regarded as a terminator. The strings *aa . . . a* and *bb . . . b* represent any sequences of characters not containing a comma.

This is tackled by converting *MOVE* into *LOAD* and scanning through the subsequent characters as far as a comma. During scanning each character is to be treated as a separate symbol. In

MAC *M O V E* **IS** *LOAD* **OUT'** *ML* **ENDM**

the symbol **OUT** has the effect of printing out the previous symbol *LOAD*. It is quoted to delay the output until the macro has been applied. *ML* is a single symbol used in effect as a special marker to be moved through the subsequent characters as far as the comma:

MAC *ML* Φ **IS 2 OUT'** *ML* **ENDM**
MAC *ML* **, IS N/L'** *STO* **OUT'** *MS* **ENDM** .

Note here the order of the macro definitions; in matching text the latter will be tried first, and thus when a match occurs it will override the first, more general, one. The control symbol **N/L** (also quoted) has the effect of producing a new line on the output stream. The symbol *MS* is used to scan through the subsequent text as far as the end of the line. This is here represented by the symbol *EOL*:

MAC *MS* Φ **IS 2 OUT'** *MS* **ENDM**
MAC *MS EOL* **IS N/L' ENDM**

To insert the symbol *EOL* at the end of each subsequent line of input the macro call *EOL* **DEOL** is made. The latter control symbol has the effect of defining the previous one, whatever it may be, as the inserted one. Processing of the text must treat all subsequent characters as individual symbols: this may be achieved by a call of the built in macro: *NULL* **DSEP**.

This example illustrates one difference between MAX and many other macroprocessors: because the matching is made by comparing symbol against symbol, a device is needed to scan through an arbitrary number of symbols in the input stream. The above example illustrates one, quite general, way. An alternative, more efficient, way is to redefine the meaning of a separator in the context of scanning a statement of the above form. Thus we define:

MAC *MOVE* **IS** *LOAD* **OUT', DSEP'** *ML* **ENDM**

On recognising the characters MOVE (followed by a space separator), the processor outputs LOAD and inserts ML at the head of the subsequent text, which is scanned with a comma regarded as the separator. When the next symbol after ML (i.e. *aa . . . a*) is recognised, then it too must be output:

MAC *ML* Φ **IS 2 OUT' N/L'** *STO* **OUT'** *MS* **SPSS**
DSEP' ENDM .

There follow the characters *STO* on a new line. Subsequent scanning requires a switch back to a space as separator (the space being represented by **SPSS** which is the separator at the time the definitions are input), Finally, when the symbol

bb . . . b following *MS* is scanned, it too is output:

MAC *MS* Φ IS 2 OUT' N/L' ENDM .

Note here that we have not used a fixed terminator at the end of this macro call. It would be quite possible to incorporate further macros to deal with text between *bb . . . b* and the end of the line using this latter method.

6. Conclusion

The preceding sections have indicated the ease with which the MAX system may be used. Although conditional macro expressions and other expansion time evaluations are not present, we have somewhat simpler, if more primitive, facilities which produce the same effect by allowing for multiple definitions. This need not make the additional demands on storage space, or on matching time, that might seem, at first sight, to be demanded. In our implementation at Bradford for ICL 1900 machines, a tree structure is used for storage of all macros. This leads to economies of space used in storing the templates by which the macros are compared with the scanned text, and also economies in time used for these comparisons. A hashing technique, as used in ML/I implementations, is not practicable in MAX, as there is no fixed position within a template which may be identified as the key-word.

Speed of matching is of great importance for a practicable implementation of MAX, as a scan is made each time the pointer is advanced through the source text; in this scan the current text must effectively be compared with all of the currently active macros. With a tree structure, this can be achieved by comparing the current symbols of the text in turn against the symbols in the macro template tree. Once a point in the tree is established further symbols are compared, the process continuing until either a complete template has been matched, or the tree has been exhausted. The situation is complicated by the existence of universal elements in the macro definitions, which can lead to more than one definition being applicable at any given position within a scan. Thus a sequence of searches may be required, but the number of alternative subtrees to be searched will be small compared with the total number of definitions each of which would be individually searched if a simple linear scan were used.

This system has been used at Bradford in text conversion applications. Speed depends, of course, on the amount of macro conversion involved, and on the total number of macros. With small sets a practicable interactive system may be constructed, even though the system was originally developed with a batch environment for lengthy texts. A translator for converting between assembly languages of differing machines was written as a set of 245 MAX definitions (Nudds, 1974). This program effectively performed the functions of an input routine besides simply converting machine code instructions, and so the conversion processes were quite complex. Although the conversion speed proved to be relatively slow, the match and conversion of a single macro was performed in about one second.

Appendix A formal definition of MAX

1. *Basic symbols*

The following are constituents of a text which is processed by the MAX system.

(*a*) the set *W* of words (strings of characters from a predefined alphabet)

(*b*) the set *I* of integers **+0, +1, −1, +2, −2,** . . .

(*c*) the set *P* of parameters **1, 2, 3,** . . .

(*d*) the set *U* containing the universal symbol ∅

(*e*) the set *C* of control symbols **BEG, END, ENDQ, DEF, IS, ENDD,** and so on.

The set of basic symbols is $B = W \cup I \cup P \cup U \cup C$.

2. *Symbols*

Any basic symbol may be combined with a quotecount; a quotecount is a member of the set $Q = \{q0, q1, q2, \ldots\}$. A *symbol* is a member of the set $S = B\,X\,Q$, e.g. if $b \in B$ and $q = qi$ then $s = (b, qi)$ is a symbol. We shall write such a symbol as $s = b^i$, and we shall write $b^\circ = b$, where no confusion might arise. If $S = (b, qi) = b^i$ then the symbol $S^- = b^{i-1}$ when $i \geqslant 1$, $S^- = S = b^\circ$ when $i = 0$ and the symbol $S' = b^{i+1}$.

3. *Markers*

There is a further class of markers

$$M = \{\Box 0, \Box 1, \Box 2, \Box 3, \ldots\} .$$

4. *Text*

A *text* (or *internal text*) is an ordered set of symbols containing one marker. An *initial text* is a text in which the marker $\Box 0$ appears as the first element of the text. Additionally an input stream of characters (the input text) is available, and an output stream of characters may be produced (the output text). Each symbol in a text may be considered to be represented by a configuration of characters in an external text (input text or output text). The rules for the interpretation of input texts and representation in output texts will be described in Sections 9 and 10.

5. *Production and Macros*

A *production* is a rewriting rule of form $t_1 \rightarrow t_2$ where t_1 and t_2 are texts. It therefore has the general form

$$U_1\,U_2 \ldots U_a\,\Box m\,U_{a+1} \ldots U_{a+b} \rightarrow V_1 \ldots V_c\,\Box n\,V_{c+1} \ldots V_{c+d}$$

where all U_i, V_i are symbols. In this case $a, b, c, d > 0$, and $\Box m$ is a marker.

The appropriate sequences of symbols are understood to be omitted where any of *a*, *b*, *c*, or *d* are zero.

A *macro* is a production with $m = n$, $b = 1$, $c = 0$. Its form is therefore $U_1 \ldots U_a\,\Box m\,U_{a+1} \rightarrow \Box m\,V_1 \ldots V_d$.

We shall abbreviate this as

$$U_1 \ldots U_a U_{a+1} = V_1 \ldots V_d\,(m) .$$

6. *Production sets*

The MAX system consists of an ordered set of productions. This production set contains

(*a*) control productions, a set *Pc*

(*b*) a variable set *Pv*

(*c*) the production set *Pm*

$\Box m\,\Phi \rightarrow 1\,\Box m$ for all $m = 0, 1, 2, \ldots$.

7. *Transformations*

These productions are used to transform a given text. Transformation of text is carried out according to the following procedure. The text is compared with productions in sets *Pc*, *Pv*, and P_m in turn.

If the text is $\ldots S_{-3}\,S_{-2}\,S_{-1}\,\Box m\,S_0\,S_1\,S_2 \ldots$ then a *match* occurs with the production

$$U_1\,U_2 \ldots U_a\,\Box m\,U_{a+1} \ldots U_{a+b} \rightarrow V_1 \ldots V_c\,\Box n\,V_{c+1} \ldots V_{c+d}$$

for which, for all i, $1 \leqslant i \leqslant a + b$,

either $U_i = \Phi$

or $U_i = S_{i-a-1}$.

The first production for which a match occurs specifies that the text is transformed to

$\ldots S_{-a-1}\ W_1 \ldots W_c\ \Box n\ W_{c+1} \ldots W_{c+d}\ S_b\ S_{b+1} \ldots$

where

$$W_i = \begin{cases} S_{j-a-1} \text{ if } V_i = j\ (j \in P) \\ V_i \text{ otherwise} \end{cases}$$

If a match occurs with some control productions, the set P_v may also change, or some other events concerned with communication with the external environment may take place.

8. *Basic control productions*

The following control productions with $m = 0, 1, 2, \ldots$ are in the set Pc. Here m^+ stands for $m + 1$, for any integer m. The same notation applies for all symbols $S \in I \cup P$

(*a*) $\Box m$ **BEG** $\rightarrow$ **BEG** $\Box m^+$

(*b*) **BEG** $S_1\ S_2 \ldots S_n\ \Box m$ **REPT** $\rightarrow$ **BEG** $\Box m\ S_1^-\ S_2^- \ldots S_n^-$

(*c*) **BEG** $S_1\ S_2 \ldots S_n\ \Box m^+$ **END** $\rightarrow \Box m\ S_1^-\ S_2^- \ldots S_n^-$
for all $S_1, S_2, \ldots S_n$ in S.

(*d*) **BEG** $S_1\ S_2 \ldots S\ \Box m^+$ **ENDQ** $\rightarrow S_1\ S_2 \ldots S_n\ \Box m$
for all $S_1\ S_2 \ldots S_n$ in S.

When productions of form (*c*) or (*d*) are applied, all productions with markers $\Box m'$ are deleted from the set P_v.

(*e*) **DEF** $S_1\ S_2 \ldots S_n$ **IS** $S_{n+1} \ldots S_t\ \Box m$ **ENDD** $\rightarrow \Box m$ for all symbols $S_1 \ldots S_t$ in S. When this production is applied, the production $S_1\ S_2 \ldots S_n = S_{n+1} \ldots S_t\ (m)$ is placed at the beginning of P_v, when $m \neq 0$. If $m = 0$ then the production $S_1 S_2 \ldots S_n = S_{n+1} \ldots S_t\ (l)$, for $l = 0, 1, 2 \ldots$ is placed in Pc.

(*f*) **DEF** $S_1\ S_2 \ldots S_{n-1}\ \Box m$ **ARG** $\Phi \rightarrow$ **DEF** $S_1\ S_2 \ldots S_{n-1}\ \Phi$ $\Box m$ **DEF'** $\mathbf{n}^+$ **1 IS' n' ENDD'** for all $S_1, S_2, \ldots S_{n-1}$ in S, and for all $n \geqslant 2$. Also ($n = 1$) **DEF** $\Box m$ **ARG** $\Phi \rightarrow$ **DEF** $\Phi\ \Box m$ **DEF' 2 IS' 1' ENDD'**.

(*g*) $\Box m$ **MAC** $\rightarrow \Box m$ **DEF BEG**

(*h*) $\Box m$ **ENDM** $\rightarrow \Box m$ **ENDQ ENDD**

(*i*) $S\ \Box m$ **INC** $\rightarrow S^+\ \Box m$ for S in $I \cup P$

(*j*) $S^+\ \Box m$ **DEC** $\rightarrow S\ \Box m$ for S in $I \cup P$

(*k*) $S_1\ S_2\ \Box m$ **ADD** $\rightarrow S_3\ \Box mx$ where, if $S_1 = a^\circ \in I$

(*l*) $S_1\ S_2\ \Box m$ **SUB** $\rightarrow S_4\ \Box m$ $S2 = b^\circ \in I$,
$S_3 = t^\circ, t = a + b$
$S_4 = U^\circ, U = a - b$

(*m*) $+\mathbf{n}\ \Box m$ **PZ** $\rightarrow$ **TRUE** $\Box m$ for $n = 0, 1, 2, \ldots$
$-\mathbf{n}\ \Box m$ **PZ** $\rightarrow$ **FALSE** $\Box m$ for $n = 1, 2, \ldots$
$S\ \Box m$ **PZ** $\rightarrow$ **TRAP** $\Box m$ for any other S.

(*n*) $\mathbf{0}\ \Box m$ **ZE** $\rightarrow$ **TRUE** $\Box m$
$+\mathbf{n}\ \Box m$ **ZE** $\rightarrow$ **FALSE** $\Box m$
$-\mathbf{n}\ \Box m$ **ZE** $\rightarrow$ **FALSE** $\Box m$ for $n = 1, 2, \ldots$
$S\ \Box m$ **ZE** $\rightarrow$ **TRAP** $\Box m$ for any other S.

(*o*) $S_1\ S_2\ \Box m$ **CONC** $\rightarrow \Box m\ S_3$ where
if $S_1 = a_1^\circ, a_1 \in W$
and $S_2 = a_2^\circ, a_2 \in W$
then $S_3 = a_3$
where $a_3 = a_1 a_2 \in W$

$S_1\ S_2\ \Box m$ **CONC** $\rightarrow S_1\ S_2\ \Box m$ **TRAP**
for any other S_1, S_2.

(*p*) $S_1\ S_2\ \Box m$ **CDB** $\rightarrow S_3\ \Box m$ where $S_1 = a_1^\circ, a_1 \in I$,
$S_2 = a_2^\circ, a_2 \in W$
and is a representation of a decimal digit d. Then
$S_3 = a_3^\circ, a_3 \in I$ and $a_3 = 10a_1 + d$.
$S_1\ S_2\ \Box m$ **CDB** $\rightarrow S_1\ S_2\ \Box m$ **TRAP** otherwise.

(*q*) $S_1\ \Box m$ **CBD** $\rightarrow S_2 S_3\ \Box m$
where

(i) if $S_1 = a_1^\circ$ and $0 \leqslant a_1 = 10c + d$ where $0 \leqslant d \leqslant 9$, then $S_2 = a_2^\circ$ and $a_2 \in W$ is the representation of the digit d, $S_3 = a_3^\circ$ where a_3 is the integer c if $c \neq 0$ or if $c = 0$, a_3 is the character $+$.

(ii) correspondingly if $0 \leqslant -a_1 = 10c + d$, with a_2 representing d, and a_3 representing c or the character $-$.

$S_1\ \Box m$ **CBD** $\rightarrow s_1\ \Box m$ **TRAP** otherwise.

9. *Output control productions*

In the latter three sets of productions it is assumed that basic symbols in W may be representations of particular external characters. Each symbol has an external representation and strings of external representations may be produced (output) using the following productions.

(*r*) $S\ \Box m$ **OUT** $\rightarrow \Box m$
the representation of S is output

(*s*) $\Box m$ **SP** $\rightarrow \Box m$
a separator character is output

(*t*) $\Box m$ **N/L** $\rightarrow \Box m$
a newline character is output, i.e. the output device is placed at character position 1 on the next line.

(*u*) $+\mathbf{i}\ \Box m$ **TAB** $\rightarrow \Box m$.
The output device is moved onwards to character position i on the same or the next line.

10. *Input control productions*

External representations may be converted into internal representations by means of the control symbol **INP**.

(*v*) $\Box m$ **INP** $\rightarrow \Box m\ x$ **INP**
where x is the next symbol which is externally represented in the input stream of characters.

In the following productions the interpretation of external representations of symbols may be controlled. It is assumed that words are represented externally by strings of characters other than the control character. The external representation of a symbol is the external representation of the basic symbol followed by that number of control symbols equal to its quote level.

Basic symbols other than words are represented as follows:
integers: their signed form preceded by the control character
parameters: an unsigned number preceded by the control character
the universal symbol: the control symbol alone.
control symbols: any other string of characters preceded by the control character.

Any number of separators may be used to separate external representations of symbols. At the end of each line of input a specified symbol may be placed on the text, when $\Box m$ **INP** occurs. The following productions control input

(*w*) $S\ \Box m$ **DSEP** $\rightarrow \Box m$
If $S = w^\circ$ where $w \in W$ and represents single character, then henceforth that character will be treated as the separator (**SPSS**). If not, then all subsequent characters will be regarded as separate words.

(*x*) $S\ \Box m$ **DCON** $\rightarrow \Box m$
Similarly the character w ($S = w^\circ$) will be henceforth treated as the control character. If $S \neq w^\circ$ for a character w, there will be no subsequent control character.

(*y*) $s\ \Box m$ **DEOL** $\rightarrow \Box m$
The symbol s is henceforth regarded as being represented at the end of each line of input, unless s is the universal symbol in which case no symbol is recognised.

(*z*) $\Box m$ **C** $\rightarrow \Box m$
All characters on the current line of input are to be ignored, (regarded as comment).

It should be noted in the above that these productions control

the effect of future applications of the **INP** production. It is possible that text to be processed after an application of one of these productions may not necessarily start with the symbol **INP**, although this will generally be the case.

References

BROWN, P. J. (1967). The ML/1 Macroprocessor, *CACM*, Vol. 10, pp. 618-623.

BROWN, P. J. (1971). A Survey of Macroprocessors, *Ann. Rev. Automatic Programming*, Vol. 6, pp. 37-88, Pergamon Press, Oxford.

MOOERS, C. N. (1966). TRAC, Procedure-describing Language for the Reactive Type-writer, *CACM*, Vol. 9, pp. 215-219.

NUDDS, D. (1974). The Simulation of a Digital Computer and its Languages on another Computer, Ph.D. Thesis, University of Bradford.

STRACHEY, C. (1965). A General-Purpose Macrogenerator, *The Computer Journal*, Vol. 8, pp. 225-241.

WAITE, W. M. (1967). A Language Independent Macro Processor, *CACM*, Vol. 10, pp. 433-440.

WAITE, W. M. (1970). The Mobile Programming Programming System: STAGE 2, *CACM*, Vol. 12, pp. 507-510.

Book reviews

Formal Languages and Programming, edited by R. Aguilar, 1976; 129 pages. (*North-Holland*, US$15.00)

One might think to use formal language theory to describe exponential growth in colonies of cells or simple animals; symbols are attached to strings one at a time by production systems, whereas all the animals reproduce themselves simultaneously, however. What is needed instead is a variant theory called Lindenmayer systems. Readers who wish to explore this tempting byway in formal language theory should certainly join the tour led by Arto Salomaa at this symposium, for he has a delightful way of describing recent theoretical work to newcomers. Thus we learn about the animals in IL systems who discuss their reproduction problems together, and those in OL systems who do not. It seems some animals are more context-free than others.

Recent work in transporting compilers across 'families' of computers, in discovering equivalent grammars from which faster compilers might be written, and in describing 'extensible' languages (i.e. families of languages) such as ALGOL 68, appears to entail the study of 'families of grammars'. These are sets of related grammars whose common features can be abstracted by grammar-generating functions called 'grammar forms'. The prophet Seymour Ginsburg came down from the mountain for just long enough to present to the symposium the tablets on which he had briefly though clearly inscribed the main results obtained by his colleagues in this new work.

Doubts are being voiced increasingly about the way in which compound data types have to be declared in the current crop of very high level languages. It is being suggested that what we ought to be offered instead is provision to declare sets or 'clusters' of functions to access the data. The data themselves should remain hidden behind the functions. Melkanoff explains how this proposal might serve to relate linguistic mechanisms for data type declaration to hardware mechanism for memory protection, and how formal language theory and proofs of program correctness might then be used to design both mechanisms more neatly.

Many of us are still arguing over the best way for pairs of co-routines to share variables in order to pass information 'horizontally' between shared storage areas. Adin Falkoff explains how files are shared in time-sharing-APL to the satisfaction of its users apparently, although his solution seems to me far less general than the better-known ones of Dijkstra and Per Brinch Hansen.

The paper by Kupka is flatulent and the one by Indermark is marred by incomplete definition of terms. Intending readers of the three minor papers should bear in mind that their French authors work in an algebraic tradition that refers to contextfree and regular languages as 'algebraic' and 'rational' languages.

R. EDWARDS (Egham)

Revised report on the algorithmic language ALGOL 68, edited by A. van Wijngaarden et al, 1976; 236 pages. (*Springer-Verlag*, DM 24) or (*Stichting MC Tract*, 50, Dfl. 25)

This report is the final definition of the programming language ALGOL 68. It is a substantial revision of the original report (Mathematisch Centrum Amsterdam, MR 101 February 1969) both in style and content, and was first published as a supplement to *ALGOL Bulletin* 36 in March 1974. The draft was subjected to meticulous scrutiny, and numerous corrections were made before the first publication of the revised report in *Acta Informatica* Volume 5, parts 1, 2 and 3 (December 1975).

The two volumes which are the subject of this review are identical reprints of the *Acta Informatica* text. For the first time, the report has been able to be printed in several typefaces, which as readers of the draft will confirm adds greatly to its readability. In addition, the review of the draft by so many people seems to have been worthwhile; by August 1976 only one minor error was known to the editor of *ALGOL Bulletin*.

The revised report is a very specialised document, for an audience of specialists. It is a formal and rigorous definition of a complex language, and as such of interest to limited classes of people. Theoretical linguists and some computer scientists may find it interesting. University lecturers with the time and inclination to study the metalanguage may find it reassuring to have on their bookshelf, as the final authority for settling arguments. Compiler writers will find it essential reading, although arduous. It is most definitely *not* suitable as an introduction to ALGOL 68 for programmers—even experienced programmers—since the formal descriptive technique is not easy to follow, even with practice.

Anyone who wishes to learn ALGOL 68 (which I would strongly recommend) should read one of the introductory texts which are increasingly available. Anyone who needs the report, particularly anyone who is still working from the draft version, should hasten to order one of these reprints. The choice between them would seem to be a matter of convenience, or the current state of the currency markets.