Part I — A Survey of Macro Processors

1 Introduction

Macro processors have recently received a considerable amount of attention. It is the purpose of this discussion to try to evaluate the uses and the limitations of macro processors and to consider the essential features of the design and, to some extent, the implementation of a macro processor with special reference to those macro processors that have already been implemented or proposed.

The first problem that arises in a discussion such as this is the problem of notation and terminology. Terminology for macros is even more variable than most terminology in computing. For example "replacement text" in one terminology is equivalent in various other terminologies to "macro definition", "macro skeleton", "defining string", "macro body", and "code body". In order to avoid confusion, a fixed terminology and notation will be used throughout this discussion. This terminology, which should be self-explanatory, is the terminology used in the literature describing ML/I [4, 5]. (It is not claimed that this terminology is "better" than any other; it is simply that if one is forced to adopt one terminology it is most natural to adopt the same terminology as one has used in previous papers.)

1.1 What is a Macro Processor?

A very wide range of pieces of software have been described as macro processors and it is not possible to draw exact boundary lines which determine what is a macro processor and what is not. In particular most processors for symbol manipulation languages can be used to some extent as macro processors and vice versa. Thus any definition of macro processor must, of necessity, be somewhat unsatisfactory.

Two dictionaries of computer terms [26, 37] define macros only in terms of their use with assembly languages, but nowadays macros are regarded in much more general terms.

For the purpose of this discussion a macro processor will be defined as "a piece of software designed to allow the user to add new facilities of his own design to an existing piece of software".

1.2 Why "Macro Processor" rather than "Macro Language"?

It is much more usual in computing to assign a name to a language rather than to the pseudo-machine that executes programs in the language. Thus one speaks of *languages* when using the terms ALGOL, FORTRAN, etc. When dealing with macros, however, it is normal to adopt the opposite approach and apply the name to the macro processor rather than to the macro language. Examples are XPOP, LIMP, GPM, ML/I. Macro processors do, of course, have associated macro languages. These languages determine the syntax of macro calls, macro definitions, etc. However, since the input to a macro processor does not normally consist purely of statements in the macro language but rather of statements in the macro language interspersed with sections of text which have no meaning to the macro processor, it is usually more convenient to describe the effect of macros in terms of the actions of a processor rather than the semantics of a language.

1.3 Uses of Macro Processors

It is not possible to separate the uses of macro processors into watertight compartments, but the following are the four main application areas:

a. Language extension.
b. Language translation.
c. Text generation.
d. Systematic editing.

In cases a) and b), the language is normally an artificial language rather than a natural language. These areas will be discussed in more detail later.

Clearly some applications of macro processors overlap the above areas and a few applications might be considered not to fall into any of the four areas. However, it is still felt that these four application areas are sufficiently comprehensive for a general discussion such as this.

1.4 Organisation of this Discussion

The sections which follow discuss in more detail these four application areas, and consider the jobs in these areas that could usefully be performed by macro processors. This part of the discussion may be considered as setting some design goals. However, a single macro processor can hardly be expected to achieve all these goals, and a macro processor cannot necessarily be considered as inferior if it has been designed with only a limited number of these goals in mind.

After the four application areas have been considered, the rest of this Chapter will be devoted to introducing the main factors that go into the design of a macro processor. These design factors do not correspond in any fixed way to the application areas, and the capabilities of a macro processor in each area will normally depend on a variety of these factors.

In Chapter 2, Chapter 3, Chapter 4, Chapter 5 and Chapter 6, each design factor will be considered in more detail. In each case general considerations affecting the choice of design will be discussed and the designs used in various individual macro processors will be considered.

Chapter 7 will sum up the preceding material. Each of the four application areas will be re-considered, and the capabilities and limitations of existing and hypothetical macro processors will be evaluated.

1.5 Language Extension

The most basic application of any macro processor is in allowing the user to extend an existing programming language to make it more suitable for his particular application. This includes the familiar use of a macro processor in providing a shorthand notation, i.e. in allowing the user to write a short sequence of characters in place of a long sequence that occurs extensively in his application. Ideally the user should be able to extend any syntactic class of a language by macro replacement, but in practice it is often only possible to add new statements and not to extend other syntactic classes.

Another goal in this field is that the macro processor should be easy to use, and it should be possible for the run-of-the-mill programmer to make full use of its facilities.

1.6 Language Translation

Given a language A and a language B, together with some rules for translating from A to B, then an application for a macro processor might be to translate from A to B. Clearly this goal can only be realized in a limited number of cases. In particular, macros are not suitable for translating between natural languages, since this area requires very specialised techniques. However, taking A to be a high-level programming language and B to be the assembly language for some machine, then it might be possible for a macro processor to perform the compilation from A to B.

1.7 Text Generation

A macro processor should be useful to a user who wishes to write "a program to generate a program", or, more generally, a program to generate any piece of text. Applications in this area are:

a. *Conditional generation.* For example a macro processor might be used to provide the optional inclusion within a program of statements required only during the debugging stages of that program.

b. *Repetitive generation.* A macro processor should be useful for the generation of repetitive text, particularly of repetitive text involving some numerical sequence. Examples are a series of data constants representing the powers of 2 or the initial value for an array where the numbers involved follow some pattern.

c. *Program parameterisation.* An alternative name for this application would be "delayed binding". As an example, assume a programming language requires array bounds to be constants. Then it might be convenient to delay the binding of array bounds until compile time by using variables for array bounds and using a macro processor to substitute values for these variables.

d. *Report generation.* A macro processor cannot hope to match the power of a special-purpose report generator. However, it might be used in a limited way. For example, one of the intended uses of TRAC [35] is as an aid to stenographers.

e. *Library facilities.* An auxiliary function of a macro processor might be to communicate with a library of (pieces of) programs and data.

1.8 Systematic Editing and Searching

A macro processor may be very useful for making systematic corrections to programs. The simplest example of this is the replacement of one variable name by another. Another example would arise if an ALGOL-oriented programmer, when writing in PL/I, omitted semicolons from all his DO statements. A macro processor might well be able to correct this error. However, macro processors are not basically suitable for making non-systematic corrections to programs, for example the correction of logical errors or of isolated syntactic errors.

In addition a macro processor might be used to search a program for some given construction, say a certain type of statement or a certain variable, and to point out all occurrences of this construction. Applications in editing and searching arise not only when dealing with programs but with any kind of text.

1.9 Introduction to the Design of a Macro Processor

The fundamental facilities offered in every macro processor are the same. Each macro processor involves the concept of a *macro call*, which is used to identify the fact that a certain piece of replacement text is to be inserted. The macro call has arguments and there are facilities for specifying where the arguments are to be inserted into the replacement text. In addition every macro processor has the facility for the user to write "macro-time" instructions, i.e. instructions that are executed during macro processing.

However, this is not to say that all macro processors are much the same. There is, in fact, an immense difference in the power and range of application of different macro processors, and the purpose of this Chapter is to introduce the areas of design where macro processors fundamentally differ.

A good starting point in a discussion of the design of macro processors is a classic paper by McIlroy [32], published in 1960, which introduced a large number of previously unpublished ideas, arising from a variety of sources. Before McIlroy's paper, the published material on macro processors consisted mainly of descriptions of simple macro assemblers, all of which were much the same. McIlroy considerably broadened the horizons. He proposed that the syntax of macro calls should not be as rigid as in the conventional macro assembler, that all text should be treated in a uniform manner, and that there should be a wide range of macro-time statements. This represents contributions in three basic areas, namely syntax, text evaluation, and macro-time facilities. He also mentioned that the use of macro processors is not confined to assembly languages but can be applied to other languages as well. This introduces a fourth consideration, namely the language into which the macro processor maps, which is called the *base language*.

Adding implementation methods as the last area of consideration, this makes up the five factors which are fundamental to the design of a macro processor. These design factors will be considered in turn in the following Chapters. They will be considered in the following order: base language; syntax; text evaluation; macro-time facilities; and implementation methods.

2 Base Language

A macro processor may only be applicable to a single base language, in which case it will be called *special purpose*, or it may be applicable to a wide range of base languages or even to any base language, in which case it will be called *general purpose.* An example of a special-purpose macro processor is Macro FAP [25], while an example of a general-purpose macro processor is GPM [38], as its name implies. Obviously a special-purpose macro processor, within its limited field of application, is normally easier to use and perhaps more powerful than a general-purpose macro processor. Some examples of how special-purpose macro processors take advantage of their base language dependence will be considered later.

In addition to macro processors that require a particular language to be the base language, there exist macro processors that depend on the base language being implemented in a special way. This subject is discussed by Cheatham [6]. In the case where the base language is a high-level programming language, Cheatham distinguishes three classes of macro as follows:

a. *Text macros.* These are macros implemented by a preprocessor to the compiler for the base language. Outside of macro calls little or no syntactic analysis of the source text is performed.
b. *Syntactic macros.* These are macros called during the syntactic analysis stage of a compiler. The syntax of the entire source text is analysed. In defining syntactic macros the user makes use of the syntactic classes built into the base language.
c. *Computation macros.* These are macros called during the stages of the compiler where machine code or some intermediate code is being generated.

Syntactic and computation macros, which are considered in more detail later, will be called compiler-dependent since they cannot be attached to an arbitrary compiler. Instead the compiler has to be designed around the macro processor.

If the macro processors that form part of macro-assemblers are examined to find whether they are compiler-dependent (or, more exactly, assembler-dependent) then most of them are not, since normally the macro processor is logically a preprocessor to the assembler and no syntactic analysis is performed in parts of the source text not involving macro replacement. However, two pieces of software that qualify as macro-assemblers, namely Lampson's IMP [28] and Ferguson's meta-assembler [11], are compiler-dependent since their design is such that macro processing and assembly must be simultaneous, with considerable interaction between the two.

In the discussion that follows, a macro processor that deals with text macros only will be called a *text macro processor*, and similarly for the two other types of macro.

2.1 Syntactic Macros

Syntactic macros are macros that are expanded during the syntactic analysis of the source text. Syntactic macros normally map into text in the base language in the same way as text macros. Existing schemes for syntactic macros require that the compiler for the base language should be syntax-directed. The macro facility is to some extent a natural extension of the syntax-directed compiler. Syntax-directed compilers as such, however, cannot be considered to be syntactic macro processors since, though compilers generated by them can be modified more easily than conventional compilers, the person performing the modification usually needs to get at the syntax tables (and the associated semantics) of the compiler that he wants to modify and to know a good deal about how these tables are designed and how entries are interrelated. This kind of operation comes into the category of modification by patching rather than modification by macro replacement, though it must be admitted that it sometimes becomes difficult to draw a dividing line between the two. However, some syntax-directed compilers, the Compiler Compiler [3] for example, do permit a certain amount of modification to be made without resort to patching.

A feature of syntactic macros is that the user can specify the syntax of the entire macro call, including its arguments. A notation such as Backus Normal Form is normally used.

Cheatham proposes two different classes of syntactic macros, called SMACROs and MACROs. Each SMACRO is associated with a given syntactic class of the base language, and calls of the SMACRO are only recognised in a context where that syntactic class may occur. SMACROs behave as if they had been built into the original syntax of the base language and their syntax can be chosen to fit in neatly with the syntax of the rest of the base language. This compares favourably with macros which require some special marker to announce their occurrence and/or require a special syntax for macro calls, which might clash with that of the base language.

Cheatham supplies the following example of an SMACRO in his paper:

```
LET N BE INTEGER
SMACRO MATRIX(N) AS ATTRIBUTE
MEANS ‘ARRAY(1:N, 1:N)’
```

where ATTRIBUTE is a previously defined syntactic class of the base language. This definition specifies that if text of form

`MATRIX(`*`argument`*`)`

occurs in a context of the base language where an ATTRIBUTE is to be expected, then the macro processor is to check that the argument is an integer and replace the original text by

`ARRAY(1:`*`argument`*`, 1:`*`argument`*`)`

Cheatham's MACROs differ from his SMACROs in that they do not have to form a syntactic class of the base language and can occur anywhere in the source text. They do, however, have to be preceded by a special marker. MACROs are, therefore, similar in many ways to some existing schemes for text macros. However, the advantage of MACROs is that the syntax of their arguments is specified and hence, in the case of an error in the form of an argument, a message can be produced when the macro is called. In the case of text macros it is often the case that error messages are produced in a stage of compilation after macro processing has been completed, and it may be difficult to relate an error message with the macro call that produced it.

Unfortunately no syntactic macro processors have yet been fully implemented, so it is rather difficult to judge their capabilities. In addition to Cheatham [6], Galler and Perlis [14] and Leavenworth[29] have also produced proposals. Leavenworth's scheme, like Cheatham's, is independent of the base language, except that it must be possible to analyse the syntax of the base language by syntax-directed techniques.

In theory any syntactic macro processor could be made compiler-independent by making it a preprocessor rather than part of a compiler. In this case, the compiler, being independent of the macro processor, need not be implemented by syntax-directed techniques. Indeed, the macro processor could be an afterthought to the compiler rather than an integral part of its design. However, in practice it would be excessively slow to analyse the syntax of the source text twice, and it is unlikely that a piece of software as elaborate as a syntactic macro processor would be written simply to serve as a preprocessor to an existing compiler.

To conclude the discussion of syntactic macros it can be said that, as far as can be judged at this time, they are powerful but difficult to use and to implement. In fact both Cheatham and Galler make the point of saying that syntactic macros would be a tool for the systems programmer rather than the run-of-the-mill user. Although syntactic macros and text macros have overlapping fields of application, they should be regarded as partners rather than competitors.

2.2 Computation Macros

Computation macros are macros which are called during intermediate stages of compilation and which are replaced by the particular form of machine code or pseudo-code used by the compiler concerned. This code can be specified explicitly by the user or it can be precompiled from some higher-level form specified by the user. Cheatham describes the second method, but most existing schemes for computation macros use the first, MAD [1] being a good example. In the MAD compiler for the IBM 7090 new operators and data types can be defined by adding sections of machine code to a table used by the compiler.

Computation macros are highly specialised, being dependent on the base language, the method of compilation, and probably on the object machine. It is hard to make any general statements about them beyond saying that some individual schemes have proved very useful.

Due to their highly specialised nature, computation macros are not considered in the rest of this discussion, and henceforth the word “macro” can be taken to mean either “text macro” or “syntactic macro”.

2.3 Difficulties in Categorising Compiler-Dependent Macros

One of the problems with compiler-dependent macros is that it becomes very hard to distinguish what is a macro and what is not. For instance a facility to define new operators in a language might be called a macro facility. However, all high-level languages allow the user to define his own operators by writing function definitions, and functions are not regarded as a macro facility.

In this discussion a facility will only be regarded as a macro if it is of form “Replace X by Y” where X and Y are, within certain limits, defined by the user. With this definition, certain built-in facilities of high-level languages still qualify as macro facilities, for example the DEFINED facility in PL/I [24].

The thinness of the dividing line between macro facilities and non-macro facilities is well illustrated by a study of a language called GPL [15]. The letters stand for “General Purpose Language”, and it has been designed as a language that any user can modify to fit his own special needs. GPL allows the user to define new functions, infix operators, and data types. These functions and infix operators may be polymorphic (i.e. applicable to many data types), and existing polymorphic operators may be extended to cater for new data types. Functions and operators may either be replaced by in-line code defined by the user, in which case the compiler for GPL acts as a macro processor, or by a function call inserted by the compiler, in which case no macro activity is involved.

2.4 Advantages of Base Language Dependence

The purpose of this section is to see what can be gained from attaching a macro processor to a single base language.

Several different special-purpose macro processors will be discussed, each of which takes advantage of its base language dependence in a different way.

The System/360 Macro-Assembler [23], the macro processor for which is described by Freeman [13], is a good example of a powerful macro-assembler. The macro processor acts as a two-pass preprocessor to the assembler. The first pass builds up a dictionary of all the assembly language variables and their attributes. (It also collects all the macro definitions, but this is not a feature of its base language dependence.) Macro replacement is performed on the second pass. Within the replacement text of a macro the attributes of variables appearing as its arguments may be examined and code generated accordingly. This facility adds considerable power to the macro processor. It is an example of a more general concept, called *context-dependent replacement*, which will be discussed later.

A different example of a macro-assembler exploiting its base language dependence is illustrated by the IMP system described by Lampson. In IMP the macro processor and assembler operate simultaneously, both using the same dictionary. The assembler is one-pass, and the loader deals with the problem of forward references. There is no difference between macro variables and assembly language variables. The EQU statement, familiar in many assemblers, acts as a macro-time assignment statement, and the value of a variable may be redefined any number of times using this statement. The occurrence of a variable name in the label field of an instruction causes the current value of the location counter to be assigned to the variable in the normal way. When the name of a variable occurs as part of an execution-time instruction, the current value of the variable is substituted in its place.

The macro processor for PL/I [24] uses its base language dependence in an interesting way. Here, following a view expressed in McIlroy's paper, the macro-time statements have been designed to have the same syntax as the corresponding statements in the base language. This saves the user the bother of learning two different syntaxes. However, the result is not an unqualified success, since as a consequence of this philosophy, the PL/I macro processor has unnecessarily powerful arithmetic facilities and poor replacement facilities. For example, since macro calls have the same form as PL/I procedure calls, each macro must have a fixed number of arguments. This is a severe restriction. The basic flaw in this philosophy seems to be that the facilities desirable in a macro processor do not correspond closely with those desirable in a high-level language.

The PL/I scheme can be generalised by allowing a compiler to make two (or more) passes through itself. On the first pass the macro-time statements in the source text would be compiled. These statements would then be executed and the resultant value of variables and functions would be inserted dynamically into the pieces of text separating the macro-time statements in the same way as in the PL/I scheme. This text, as modified, could then be fed back as input to the compiler. This approach allows all the base language facilities to be available at macro-time, and is economical to implement in that the same piece of software is used to compile both the macro-time statements and the final base language statements.

Leroy's [30] proposed macro facilities for ALGOL are similar to those in PL/I, though Leroy goes further than PL/I by allowing macro-time subscripting and expression evaluation within base language statements. Leroy analyses the complete syntax of the source text, and only allows macro-time statements to appear where base language statements can appear. Similar restrictions apply to other syntactic classes.

Leroy's scheme is, in fact, somewhere between a text macro processor and a syntactic macro processor. The syntax of the entire source text is analysed but this is mainly for checking purposes and the user has no control of the syntax of macro facilities.

Meta-assemblers, which may be regarded as a special class of macro processor, should also be mentioned at this point. Descriptions of meta-assemblers have been published by Ferguson [11] and by Graham and Ingerman [16]. A meta-assembler is a generalised assembler where the formats of the instructions in the assembly language are defined by the user. These "formats" can be regarded as macros with many alternative names. The base language into which these macros map is either absolute machine code or code for a loader. Each macro has, within its replacement text, output statements to generate the code to replace it. When a macro has alternative names each name is replaced by a number defined by the user. This number will normally be the numerical code for the machine instruction represented by the macro.

3 Syntax

There are three primary syntactic considerations in the design of macro processors. These are:

a. The syntax of macro calls.
b. The way in which formal parameters are specified within the replacement text of a macro.
c. The syntax of macro-time statements.

A macro is, in general, defined once and called thousands or even millions of times. Hence syntactic considerations that only apply to the defining of a macro and the specification of its replacement are relatively unimportant. Of the three considerations above, the syntax of macro calls is far and away the most important. As well as affecting the convenience of use of a macro processor, the syntax of macro calls, as will be seen, can determine its power and range of application. On the other hand, considerations b) and c) above are largely a matter of personal taste and hence do not merit much general consideration.

3.1 Syntax of Macro Calls

The syntax of macro calls involves, in general, choosing symbols to indicate the beginning and end of the call and to separate the arguments. In this discussion the word *symbol* will be taken to mean any predefined sequence of characters (or, in a context where text is not treated in units of characters, any predefined sequence of text units), and it will be assumed that the character "newline" occurs at the end of each line of text. "Newline" and similar characters indicating the layout of text will be called *control characters* and other characters will be called *explicit characters*, since these latter will have been explicitly written by the programmer.

The syntax of a macro call must be chosen so that:

a. The call can be recognised as such.
b. The arguments (if any) of the call can be recognised and separated from one another.

These two considerations can reasonably be separated from one another and hence will be considered separately.

3.2 Recognition of Macro Calls

This section will describe how macro calls can be recognised in the source text. As has been said, the syntax of replacement text is relatively unimportant, and this applies equally to the recognition of macro calls within replacement text. The same recognition method may be used as in the source text or special rules may be made as to the format of replacement text so that macro calls can be recognised more easily. It is quite acceptable to have rigid rules on the form of replacement text but much less acceptable to have such rules governing the form of source text.

The first point to be considered with respect to the recognition of macro calls is the scope of the search. General-purpose macro processors scan the entire source text looking for macro calls, though there is normally a facility for specifying that certain strings are to be treated literally (see “Skips” in Chapter 4). On the other hand, special-purpose macro processors might only search for macros in particular contexts in the base language. For example, a macro-assembler might only recognise a macro name in the operation field of an instruction, and Cheatham’s SMACROs are only recognised in a specified syntactic class of the base language.

The second and more important point is the recognition method. The source text will, in general, consist partly of text to be copied and partly of macro calls, i.e. text to be replaced. The macro processor must be able to recognise these macro calls and separate them from the text surrounding them. This involves determining the following information about each macro:

a. The start.

b. The end.

c. The macro to be called.

A macro processor knows it has encountered a macro call when it has ascertained one of the three above pieces of information, and from this deduces the two other pieces of information. The interrelation between these pieces of information determines the character of the recognition method. The recognition methods used by various macro processors are discussed below in the light of this. However, before these methods are discussed it is necessary to consider briefly how a macro can be identified.

There are basically two ways of identifying the macro to be called. These will be called *name recognition* and *pattern matching*. In the name-recognition method a macro is identified by a unique symbol associated with the macro. This symbol is called the *macro name*. In the pattern-matching method, each macro has associated with it a pattern, which consists of a sequence of fixed strings interspersed with strings which, within certain limits, are arbitrary. A macro is identified by the occurrence of its pattern. There may be elaborate rules, as in LIMP [39], for identifying the macro if a macro call matches more than one pattern. The various recognition methods are now considered.

3.3 Recognition of Start First

In many macro processors the processor is told when it has reached the start of a macro call by the occurrence of a predefined symbol, the same symbol being used for all macro calls. This symbol will be called a *warning marker*. Examples of macro processors which use warning markers are GPM, TRAC, Cheatham’s MACRO facility, and, if it is in “warning mode”, ML/I. In all these cases except the MACRO facility, the macro is subsequently identified by the name recognition method, and the macro name must follow immediately after the warning marker.

Once the start of a macro call has been identified, the end and the macro to be called can be decided in either order. ML/I identifies the macro first and, as will be discussed later, this determines the syntax of the rest of the call, including where the end is. GPM is to some extent the opposite as instead of giving freedom in the choice of symbol to delimit the end of a call, it uses a fixed universal symbol (the semicolon) for this purpose but gives more freedom in specification of the macro name. In GPM the macro name is given by the text from the warning marker up to the next comma or semicolon. This text may contain macro calls and hence the macro name can be constructed dynamically during macro processing. This is a very useful facility.

3.4 Macro Identification First

In many macro processors the macro processor does not know it has a call on its hands until it has identified the macro to be called. As has been said, a macro may be recognised by its name or by a pattern. If a macro is recognised by its name it is necessary in practice to place some limitation on the recognition process otherwise the user would find many pieces of text taken as macro calls when they were not intended as such. The two most common methods of limitation are:

a. *Restricted scope.* This is the method adopted by most macro-assemblers, which only recognise macro names in the operation field of the instruction.
b. *Larger text units.* The PL/I macro processor and ML/I do not scan text character by character but rather in larger units called *tokens* or *atoms*. In each case an identifier is treated as a single unit rather than as a sequence of individual characters. Thus if DO were a macro name it would not be recognised as such in an identifier such as DOG or RANDOM. (ML/I is mentioned in this section as well as the last since it has two modes of working, “warning mode” and “free mode”. The mode of working determines the recognition method.)

Similarly, if a macro is recognised by a pattern-matching process it is usual to make some limitation on the recognition process. In the case of pattern matching, recognition would be incredibly slow if every sequence of characters in the source text had to be compared with every pattern. Thus one of the two following limitations is made in practice:

a. *Restricted scope.* This is the method used in Cheatham’s SMACRO proposal, and the proposal for syntactic macros by Galler and Perlis.
b. *Fixed start and end.* In LIMP, WISP [40], and SYGMA [10] each pattern must begin and end with fixed characters. In LIMP and WISP patterns must occupy exactly one line of text. This is equivalent to tacking the character newline onto the beginning and end of each pattern. SYGMA, on the other hand, requires that patterns be enclosed in parentheses.

Once the macro has been identified it is necessary to find the beginning and end of its call. This presents no problem if the macro is identified by the pattern-matching method since the recognition of a pattern automatically determines the beginning and end. The beginning and end may be predefined symbols, or, in the case of syntactic macro processors, they may be determined by the syntactic analysis of the source text surrounding the macro call. In the proposal of Galler and Perlis, the new operators that can arise within macro calls are given a priority relative to other operators and this determines the boundaries of macro calls.

If a macro is identified by its name, then it is still necessary to find the beginning and end of its call.

In ML/I and the PL/I macro processor, the macro name must come at the start of the call, and hence the start is immediately deduced on recognising the macro name.

In most macro-assemblers, on the other hand, the start of the call is the “newline” at the start of the line on which the macro name occurs. XPOP [17, 18] is probably unique in that, in one mode of operation, the macro name need not precede its arguments but may occur anywhere in a call. Most other macro-assemblers only allow a label to precede the macro name, and this label is usually treated by a special mechanism and cannot be regarded as an argument of the call.

As in the case where macro calls were identified by a warning marker, the symbol denoting the end of a macro call can be a fixed universal symbol or a symbol dependent on the macro being called. The latter method, which is clearly more flexible, is used by XPOP, ML/I, and Leavenworth’s proposed syntactic macro scheme.

Since this latter method of specifying the end of a macro call is so much more flexible, it is worth considering whether a similar method cannot be applied to specifying the start of a call since macro processors are rather inflexible in this respect. To take a specific example, ML/I gains in flexibility by allowing the user to specify the syntax of the macro call, but all the syntax specified by the user must follow the macro name. It would be an improvement if the user could also specify the syntax of a part of the macro call to precede the macro name. This would allow arguments to precede the macro name, a facility already offered by XPOP, and, more important, it would allow the symbol starting the macro call to be dependent on the macro being called, which would be a unique facility for a text macro processor.

The practical reason why this desirable facility is not offered to the user is, of course, the problem of implementation. It is much easier if a macro processor only requires forward scanning and if a backward search of arbitrary length is possible, then the entire source text must be preserved until macro processing is completed.

There will be more discussion about the merits of the various recognition methods later in this Chapter.

3.5 End First

In theory it would be possible to design a macro processor which commenced the recognition process for a macro call by identifying the end of the call. However, this method has no obvious advantages and it presents some practical difficulties since it is normally easier to scan text forwards rather than backwards. Moreover, nearly all programming languages have been designed for forwards scanning, and programmers have become accustomed to writing in this kind of notation.

3.6 Separation of Arguments

In a syntactic macro processor, the macro processor is tied in with a syntax-directed compiler and this syntax-directed compiler can be used to analyse the entire piece of text representing a macro call. Normally the syntax-directed compiler will be sufficiently powerful to allow the user to choose any notation he pleases for writing a macro call.

Text macro processors are very different, since the syntax of arguments to macro calls is not usually analysed. Arguments are arbitrary (or almost arbitrary) pieces of text. Arguments are separated by predefined symbols called *separators*, and the last argument is followed by a symbol called a *terminator*. Separators and terminators are called *delimiters*. In this section the ways in which delimiters can be specified in various different text macro processors are discussed and compared.

Many macro processors have a very rigid syntax for writing delimiters. There is often a universal separator, usually a comma, and a universal terminator, and each macro must have a fixed number of arguments. A notation such as this will be called *basic notation*.

It is desirable to be able to specify macros with an indeterminately long list of arguments, and basic notation is often extended to allow for this. One way of doing this is to allow each argument to be a single string or a parenthesised list of strings, separated by commas. This notation is used, for example, in Macro FAP. In macro processors where this facility is available, there must, of course, be macro-time statements for iterating through lists of arguments.

Another useful way to extend basic notation is to allow an argument to be optionally omitted, and, if this happens, a default value to be assumed. "Keyword parameters" in the System/360 Macro-Assembler offer this facility. As an example, to illustrate the use of keyword parameters, assume it was desired to have a macro called INCREASE which increased the contents of a storage location by a constant. Assume further that it was desired to assign a default value of one to this constant. To achieve this, a keyword would be chosen for this constant. Let this keyword be AMOUNT. Then in the declaration of the INCREASE macro, AMOUNT would be declared as a keyword parameter with default value one. If it was desired to increase the location XYZ by one, then the call of INCREASE would be written:

```
INCREASE XYZ
```

whereas if it was desired to increase XYZ by a different amount, say two, then the call would be written:

```
INCREASE XYZ, AMOUNT = 2
```

These improvements to basic notation are very desirable, but some macro processors go further and allow a much more general syntax than basic notation. The idea of allowing a more general syntax was propounded in McIlroy's paper. The schemes that will be considered in some detail here are those of XPOP, LIMP, and ML/I. In each of these the user can make up a language of his own by designing suitable macros for his own application in the notation most suitable for that application.

3.7 XPOP Syntax

Of all macro processors, XPOP is probably the one which allows the most flexibility in the writing of macro calls. In fact the designer of XPOP, Halpern [20], even claims that the user of XPOP can get quite close to writing in the English language. In XPOP each macro can have its own set of symbols for delimiting arguments. The user may define for each of his macros:

a. A set of symbols, any of which is recognised as a separator.
b. A set of symbols, any of which is recognised as a terminator.
c. A set of symbols, called *noise words*, which are completely ignored if they occur in a call of the macro.

As an example of the notation possible in XPOP, a macro call that in basic notation would be:

```
STORE, ALPHA, BETA, GAMMA
```

could be in XPOP:

```
STORE INTO CELL ``ALPHA'' THE LOGICAL SUM ...
FORMED BY OR'ING THE ...
BOOLEAN VARIABLES ``BETA'' AND ``GAMMA''.
```

or any of the wide variety of possible forms derived from permuting the separators and noise words in the above form.

In a later version of XPOP, a further notational convenience has been introduced. This is called the QWORD facility and is rather similar to the scheme for keyword parameters described earlier. However, QWORDs are a device to allow the arguments of a macro to be written in any order, whereas keyword parameters were used in assigning default values to arguments.

Although the XPOP user has such considerable freedom in specifying his notation for writing macro calls, the notation has no meaning. As far as the generation of text to replace a macro call is concerned, the notation used in writing the call is immaterial, and the effect is as if basic notation had been used in every case.

A very much simpler and less notationally powerful system that adopts a similar philosophy to XPOP is the macro processor for the Elliott 803 [9]. In this the user can write any comment between the arguments of macro calls. A typical call might read:

```
``LOOP'' FOR (A) := (B) STEP (C) UNTIL (D) DO (E)
```

3.8 LIMP Syntax

In LIMP, as has been said, macros are identified by the pattern-matching method. Hence each macro may be regarded as having its own pattern of delimiters. When a macro call is written, the notation that is used determines which macro is to be called and hence which piece of replacement text is to be substituted. Thus in LIMP, unlike in XPOP, notation has a meaning. However, the flexibility of notation in LIMP is much more limited than in XPOP. In XPOP it is quite easy to design a macro with a large number of alternative forms of writing its call, whereas in LIMP it would be necessary to enumerate all the possible patterns, which would be a tedious business.

There is no facility in LIMP for having a macro with an indefinitely long list of arguments, for example a macro of form:

```
X = A1 (-) A2 (-) ... (-) An
       (+)    (+)      (+)
```

However, this limitation can be got round by the use of LIMP's powerful string manipulation facilities or by the use of recursive techniques, though each of these methods would be slow and somewhat inconvenient to use.

3.9 ML/I Syntax

In ML/I each macro has its own *delimiter structure*, which defines the patterns of delimiters that can occur in calls of the macro. Each macro can have a wide range of possible delimiter patterns, and indefinitely long sequences of delimiters are possible. A delimiter structure is in fact a directed graph where the nodes represent the delimiters and the paths leading from a node determine which delimiters can follow the delimiter represented by the node. There are facilities in ML/I for causing the text replacing a macro call to be dependent on the pattern of delimiters that occurred in the call. It is worth comparing the ML/I method with that of LIMP in the case where a macro has N possible delimiter patterns, where N exceeds one. (In LIMP the macro would be represented as N different macros.) It is assumed that each of the N delimiter patterns requires a different replacement. If N is small then LIMP wins, since it is necessary in ML/I to write statements within the replacement text of the macro to test which pattern occurred in its call, whereas in LIMP each pattern is uniquely associated with its own piece of replacement text. As N becomes larger, however, it becomes increasingly tedious to enumerate all possible patterns, as is required by required by LIMP, and the ML/I method becomes better. It would, for instance, be almost impossible in LIMP to implement a macro of form (where the notation is that of Brooker and Morris [3], which is also used in the description of ML/I):

```
IF relation [ (|) relation *? ] THEN ... [ ELSE ... ? ] END
            [ (&)             ]
```

where a *relation* was of form:

```
         (=   )
argument (>   ) argument
         (<   )
         (etc.)
```

On the other hand, it is relatively easy to write such a macro in ML/I. Another feature of ML/I is that it allows *exclusive* terminators. This is a means of saying that a call is to be the text up to but not including a given symbol. This has several applications, one of which is to allow a symbol to serve the dual purpose of signifying the end of one call and the beginning of another, as the symbol "newline" does in LIMP.

3.10 Notation for Formal Parameters

Formal parameters are used in the replacement text of a macro to indicate where the arguments are to be slotted in. The two methods most commonly used to designate formal parameters are *designation by number* and *designation by name.*

In the first method a formal parameter is represented by a number, which indicates the position of the corresponding argument in the macro call. This number is usually preceded by a unique marker. ML/I uses this method for specifying formal parameters and to some extent generalises it by using the same mechanism for denoting the inserting of delimiters, macro variables, and macro labels.

In the "designation by name" method, each formal parameter is represented by a unique symbol, normally an identifier. The correspondence between these symbols and the arguments is defined either in the macro declaration, or by means of separate statements as in TRAC. In TRAC formal parameters are defined using a general string-segmentation statement, a method offering considerable flexibility. In “designation by name”, formal parameters might be regarded as local macros, the replacement text for each formal parameter being the corresponding argument.

“Designation by name” is probably the method that would most appeal to the user, although when a macro can have an indefinitely long list of arguments, some sort of numbering system is always necessary. XPOP extends designation by name in an interesting way. In XPOP if no argument is supplied corresponding to a given formal parameter then that formal parameter stands for itself (unless the user of XPOP takes special action to inhibit this facility). Hence the name of the formal parameter is its default value. However, this scheme for default values has its pitfalls and the method of keyword parameters mentioned earlier is probably preferable.

3.11 Syntax of Macro-Time Statements

Some macro processors, for instance GPM, regard macro-time statements in the same way as macros, except that the action for macro-time statements is built into the system rather than defined by the user. In these systems macro-time statements can be regarded as system macros; they have the same syntax as calls of ordinary macros. This method makes it possible for the user to build up his own macro-time statements in terms of existing ones by using macro techniques.

TRAC adopts the opposite approach to GPM by treating macro calls as a special kind of macro-time statement. Moreover, there are processors which treat the two concepts as entirely different entities with completely different syntaxes. In this latter case the syntax of macro-time statements does not have fundamental importance and any syntax that is reasonably easy to use and to implement is entirely satisfactory.

3.12 Comparison of Syntaxes

The purpose of the rest of this Chapter is to consider some general points that are of relevance in the choice of syntax of a macro call, and to consider what is gained by the more general forms of syntax. It must be emphasised that it is not intended to show that one method is “better” than some other method, as this is rarely, if ever, completely true.

3.13 Notation-Independence

Macro processors requiring a rigid syntax for macro calls suffer from the disadvantage that the text to be fed to them has to be written with the macro processor in mind. Macro processors allowing a more general syntax have applications in performing symbol manipulation on arbitrary pieces of text. For example, ML/I and LIMP can be used for applications in context editing; in particular, ML/I has been used as an aid to converting between FORTRAN IV and FORTRAN II (see [4]). Macro processors which can work on arbitrary text will be called *notation-independent*. Strictly speaking the term “fairly notation independent” should be used when talking about ML/I and LIMP, since each has its limitations. However, macro processors such as these, which give a fair degree of freedom on the form of the source text will be included under the heading “notation-independent”. Note that a macro processor can be general purpose but not notation-independent. An example is GPM.

Over and above their use as editors, macro processors that are notation-independent have a more important advantage. This advantage is the capability of “working behind the user’s back”, and thus effecting “context-dependent replacement”. This is discussed in the next section.

3.14 Context-Dependent Replacement

Context-dependent replacement is the replacement of a macro call by a piece of text, the form of which is dependent on the base-language context in which the call occurs. An example of this, which has been mentioned already, is the case where the text replacing a macro call depends on the data attributes of the variables supplied as arguments; in this example the context is supplied by declarative statements. There are many other examples where context-dependent replacement is desirable. For instance it might be desired to know whether or not a macro call occurs within a base language subroutine, what the name of the current “control section” is, whether or not the call is within a DO loop and, if so, what the controlled variable is, and so on. In fact most forms of optimising performed by compilers represent context-dependent replacement. It is clearly impractical for the user to supply extra arguments to macro calls in order to pass all this information across. Indeed the information might not be known at the time the macro call was written. Hence the macro processor should glean all the required information for itself by examining the appropriate base language statements and remembering information by setting macro variables.

A macro processor that is notation-independent is capable of doing exactly this, though it may only be capable of using the context supplied by the text preceding a macro call, and not the context supplied by the text following a call. (Multi-pass macro processors are discussed in Chapter 6.) Assume, therefore, that it is desired to use such a macro processor to “intercept” base language statements of a given form in order to keep a record of context. To achieve this, a macro is defined with the same syntax as the base language statement so that each occurrence of the base language statement results in a call of the macro. The action of the macro is to set those macro variables that are used to keep a record of the base language statements and to replace the macro call by a copy of itself so that the original text is not changed. (A concrete example of this technique is given in section 7.4.6 of Appendix A.) The user can be entirely unaware of this behind-the-scenes activity of the macro processor. Indeed, if code written by one programmer were placed in the middle of some code written by another programmer, then statements written by the second programmer could be intercepted by the macro processor and could affect macros written by the first programmer. The second programmer could be completely unaware of the existence of the macro processor.

3.15 Use of Newline

Many macro processors, including LIMP, WISP, and most macro-assemblers, require that each macro call occupy exactly one line of text (or more strictly one record of text since there is often a facility for overflowing to the next physical line if one physical line is filled up). This has the following advantages:

a. The user does not have to write explicit characters at the beginning and end of each call.
b. The text is easier to read.
c. A single character “newline” can serve a dual purpose, namely to terminate one call and commence another.
d. Some degree of notation-independence is achieved.

Needless to say, however, there are some compensating disadvantages. These include:

a. Calls cannot be nested within other calls.
b. For the macro processor to be practically useful, it is almost imperative that base language statements should be written one to a line. Certainly they should not be allowed to straddle lines.
c. As a result of b), each macro must represent a series of base language statements. It would not be possible for a macro to represent any other syntactic class, for instance a variable name.

3.16 Notational Restrictions

All the text macro processors that have been considered have required that each macro call should begin and end with predefined symbols. (These symbols are either universal to all macros as in GPM, or dependent on the macro being called, as in ML/I.) Hence all calls must be bracketed within fixed delimiters. This notation will be called *bracketed notation*.

Many people regard the requirement for bracketed notation as a great disadvantage of macro processors. For example, they would like to write a macro call as *arg1* + *arg2* rather than some of the following alternatives, which are offered by various macro processors:

a. ADD (*arg1*, *arg2*)
b. + (*arg1*, *arg2*)
c. $ +, *arg1*, *arg2*;
d. The alternatives offered by macro processors that would require each macro to be written as a statement.

Various techniques can alleviate the awkwardness of bracketed notation, in particular the use of the character “newline” as a delimiter and the use of “exclusive” terminators. However, the only macro processors that manage to dispense with bracketed notation altogether are the syntactic macro processors of Cheatham and of Galler and Perlis. Even these have not been implemented, though they are feasible to implement. The reason why these macro processors are able to get away from bracketed notation is that there is no problem in separating macro calls from arbitrary pieces of text in between them, since the syntax of the entire source text is analysed. Macros are restricted to occurring in certain syntactically determined positions and the delimiters enclosing these positions also serve to delimit macro calls.

The question arises as to whether it is possible for a text macro processor to get away from bracketed notation. In other words, if bracketed notation is to be avoided, does the syntax of the entire source text need to be analysed? The only way of avoiding bracketed notation is to allow the syntax of arguments to be specified and to recognise macro calls by the pattern matching method. If the user were allowed much freedom in defining this syntax, it would be necessary to use syntax-directed techniques to analyse macro calls. It would, however, be incredibly slow to analyse the attempt to match every sequence of characters in the source text. Moreover, several ambiguities would arise. Hence it is necessary in practice to use a method such as Cheatham's which analyses the entire source text and only searches for macros in specified syntactic classes.

It might be possible, however, to allow the user to specify the syntax of his arguments by placing them in one of a number of predefined classes, such as "identifier", "bracketed text", etc. Provide these classes were very easy to recognise, it might be possible for a text macro processor to recognise macro calls not in bracketed notation without being too slow. However, it remains to be seen how simple these classes would need to be in practice and, as a result of this, how useful the macro processor was.

3.17 Errors and Error Recovery

The effect of errors in writing the delimiters of macro calls depends on the recognition method and on the syntax of macro calls. If a macro call is recognised only when a macro is recognised, an error in writing a macro call might lead to the macro call not being recognised as such. This applies especially if macros are identified by pattern matching. The end result of a macro not being recognised will be an error during compilation or assembly, or worse still, during execution.

In macro processors that use warning markers or macro processors that recognise a macro by its name, some syntactic errors will be detected at macro-time, which is, of course, the most desirable time. However, even if an error is detected it may considerably upset subsequent macro processing, especially in macro processors which allow nested macro calls. In GPM or ML/I, for instance, if the terminator of a call is omitted then the entire source text might be scanned to search for the terminator, thus effectively ending macro processing. In ML/I this applies even if an intermediate delimiter is incorrectly specified.

As far as error detection and recovery is considered, it is best if a processor has a lot of restrictions and a lot of redundancy in specifying syntax.

In macro processors which recognise a macro by its name, there is a danger of a piece of text being taken as a macro call when this was not intended by the user. However, this problem is worse in theory than in practice. In support of this assertion it can be said that ML/I offers the user the choice of writing a warning marker in front of each macro call or of risking unintended calls. Nearly all users take the risk as they would almost certainly make more errors if they worked in "warning mode" due to the mistyping or accidental omission of warning markers.

4 Text Evaluation

The basic action of a macro processor is to scan text and perform certain replacements in the text. This process will be called *evaluating* text. A macro processor will, during any one job, normally scan source text, replacement text and arguments to macros. The order in which this is done, the interrelations between different pieces of text and the effect of new macro definitions on subsequent evaluation are all questions that are fundamental to the design of a macro processor.

McIlroy's philosophy that all text should be treated in a uniform manner seems a desirable criterion to adopt in this area. An implication of this is that recursion should be permitted. Many people feel that recursion is a somewhat academic tool in a high-level language, and the lack of necessity for recursion in this area is illustrated by the success of FORTRAN. However, in macro processing, as in other symbol manipulation applications, recursion is very desirable and arises in quite simple applications.

This Chapter will start by discussing two other considerations which are as important as recursion in determining the implementation method but which normally receive little attention in the design of a macro processor. These considerations may be considered to be in the same status nowadays as recursion was several years ago, say before the advent of ALGOL. In those days it was often impossible from reading the description of a piece of software to ascertain whether recursion was permitted, probably because the designers of the software had not even considered the problem. The two considerations to be discussed below, which will be called respectively "name or value" and "multi-level calls", are in the same state nowadays.

4.1 Name or Value

The difference between "call by name" and "call by value" for subroutine arguments is well understood, but similar considerations for arguments of macro calls have received little attention. The problem arises when it is possible for an argument of a macro call itself to contain a macro call, i.e. when one macro call, which will be called the *nested call*, can be nested within another, which will be called the *outer call*. The following are possible ways of treating this situation:

a. If a nested call is encountered it is immediately evaluated and replaced by its value. This method will be called *call by immediate value.*
b. The outer call is scanned over without performing nested calls. Each argument of the outer call is evaluated and replaced by its value before the replacement text of the outer call is entered. This method will be called *call by delayed value.*
c. The outer call is scanned over without performing nested calls, and its replacement text is evaluated. Arguments are evaluated only when they are inserted into this replacement text. This method will be called *call by name.*

This is not, of course, a complete list of possibilities. However, the above three methods are the methods most likely to be used in practice. To illustrate the difference between the methods, consider the following GPM macro calls:

```
$ OUTER, $ NEST;, C;
```

where the replacement text of the macro NEST is “A, B”. Depending on which method was used — in actual fact GPM uses “call by immediate value” — the arguments passed to the call of OUTER would be:

Method of call	**1st Argument**	**2nd Argument**	**3rd Argument**
Immediate value	A	B	C
Delayed value	A, B	C	(none)
Name	$ NEST;	C	(none)

Due to the implementation overheads of offering the user a choice, most macro processors have a fixed method of treating arguments, which applies to all arguments of all macros. This contrasts with some algebraic programming languages which offer the user the choice for each argument of whether it is to be called by name or value.

In a very large number of cases the choice of method has no effect on the end result. However, there are some important cases where one or other of the methods is definitely superior, and some of these will now be considered.

In applications such as the generation of machine code where macros may be closely interrelated, it is important, when macro calls are nested, to be able to communicate from one macro to another. In this respect, “call by name” is superior to the other methods since nested macro calls are evaluated in the context into which they are to be inserted. Thus in the earlier example assume that the macros OUTER and NEST both generate code for a machine with a single accumulator and assume further that a macro variable X is used to keep track of what is in the accumulator. If the “call by name” method is used then X can be examined within the replacement text of NEST and code can be generated accordingly. If either of the other methods of call is used this technique does not work since X is examined at the wrong time. This advantage of “call by name” is similar to the situation that arises in syntax-directed compiling, where “top down” is better than “bottom up” for code generation.

In addition to this, two other merits can be claimed for “call by name”, namely:

a. In the replacement text of a macro both the text of arguments and the values of arguments may be examined (e.g. ~A1. and ~WA1. in ML/I).
b. It may be possible to create, within the replacement text of a macro, macros to operate on its arguments (e.g. the use of unprotected inserts in ML/I).

On the other hand, the “call by immediate value” method has the following advantages:

a. It makes “descendant calls” more powerful. These are discussed in the next section.
b. Macro names may be generated dynamically from other macro calls.

“Call by delayed value” has the second of these advantages, but apart from this can be simulated using the “call by name” method.

4.2 Multi-Level Calls

A macro call is a piece of text. The question arises as to whether a macro call must be written as a contiguous piece of text or whether it can be built up of separate pieces of text. This is best illustrated by an example. Assume that in ML/I the following definitions are written:

```
MCSKIP MT,<>;
MCDEF NAME AS <MCSET>;
MCDEF RHS AS <P2+1;>;
NAME P1 = RHS
```

The question to be asked is whether the last line represents a call of the macro MCSET. In fact, ML/I does not allow calls such as this so the answer is "no". However, assume that there exists a macro processor, called ML/II, which is similar to ML/I except that it allows calls such as the one above. In ML/II, therefore, the call of MCSET would start in the replacement text of NAME and then ascend to the source text with the text "Pl=" and then descend into the replacement text of RHS for the text "P2 + 1;". Hence, one may speak of *descendant* and/or *ascendant* macro calls, and a macro call which ascends and/or descends will be called *multi-level*. A multi-level call is, therefore, a call built up of separate pieces of text.

Multi-level calls are not simply an academic consideration. They have very important applications in the fields of optimisation and simplification. To illustrate this, an example in the field of optimisation will be considered. One of the well-known problems with using macros to generate machine code is that inefficiencies occur at the boundaries between macros. Thus in the case of PDP-7 assembly language, one macro might generate the instruction

```
DAC XXX
```

(which means "store accumulator at XXX") whereas the next instruction, occurring in the source text or generated by another macro, might be

```
LAC XXX
```

(which means "load accumulator from XXX"). Now this second instruction is redundant, and so it is desirable to write a macro which deletes the second instruction every time the above combination occurs. In ML/II, which like all hypothetical software, is very powerful, this can be achieved simply by designing a macro with name:

```
‘‘DAC XXX
LAC XXX’’
```

and with replacement text "DAC XXX". (This macro works only for one particular XXX. However, it would not be very difficult to generalise it to apply to all XXX.) In ML/I, however, a macro such as this would be useless (unless it was used in a second pass through some previously generated text) since it would only delete the redundant instruction if it had been written physically adjacent to the preceding instruction, a situation that would only arise if the programmer who wrote the instructions was incredibly incompetent.

Now that the existence and potential uses of multi-level calls have been established, the various methods of replacing a macro call will be considered in order to see which methods allow ascendancy and/or descendancy.

Assume that a piece of text has been scanned forward by a macro processor until it has reached the stage that it has found a call and is ready to replace it. (This stage is reached at the end of the first call that is encountered if arguments are called by value, or at the end of the first non-nested call if arguments are called by name.) Let the text be of form XCY where C is the macro call, X is the text preceding it, and Y is the text following it. (As in "XCY" above, the concatenation of string names will be used to represent the corresponding concatenation of the strings they represent.) Furthermore, let C' be the replacement text of the macro called in C, and let the notation V(S), for some string S, represent the result of evaluating that string. Using this notation, five possible methods of evaluating the string XCY can be distinguished. These are:

a. V(XCY) = V(X)V(C')V(Y). This allows no ascendancy or descendancy, since each of X, C', and Y is evaluated in isolation.
b. V(XCY) = V(XC')V(Y). This allows descendancy (since a call can lie partly in X and partly in C') but not ascendancy.
c. V(XCY) = V(X)V(C'Y). This allows ascendancy but not descendancy.
d. V(XCY) = V(XC'Y). This allows both ascendancy and descendancy.
e. V(XCY) = V(X<V(C')>Y), where the symbols "<" and ">" mean that the text designated by the enclosed symbols is to be copied literally and not evaluated. This method is similar to method a) but allows C to occur within another macro call.

If arguments are called by immediate value, then C may be nested within a call lying partly in X and partly in Y. In this case neither X nor Y can be evaluated in isolation and hence only methods d) and e) are applicable. With other methods of argument treatment, method a) is most likely to be employed, though any method is possible.

Note that the above list of methods is not an exhaustive one, and many other possibilities, some reasonable and some absurd, can be imagined. TRAC, for example, has an option whereby replacement text is treated as a literal string. This can be represented as

V(XCY) = V(X<C'>Y)

The action of ML/I on encountering a nested call can be represented as

V(XCY) = V(X<C>Y)

GPM adopts a compromise between methods d) and e). The beginning and end of a call must be at the same level, as in method d), but intermediate delimiters can be at lower levels, as in method e). Any of the above methods can be applied to the replacement of formal parameters by arguments as well as to the replacement of macro calls. It is conceivable that a macro processor could use a different method in the two cases.

4.3 Conclusions on Multi-Level Calls

The advantage of full generality in multi-level calls is that it makes it easier to write macros for simplification and optimisation. Otherwise operations such as these might require several passes through the source text.

However, it is necessary to mention the disadvantages of full generality. Firstly, there are implementation problems, which may be quite severe if macro calls need not be properly nested. Secondly, there is the problem of error recovery if ascendant calls are allowed. The problem is that if the user accidentally omits a terminator of a macro call which occurs within replacement text, then the macro processor will never give up searching for this missing terminator, whereas if ascendancy were forbidden then the error would be detected when the end of the replacement text was reached. Thus the user might have terrible difficulties trying to locate his bugs if ascendant calls were allowed.

4.4 Output

Now that the order and methods of evaluation have been considered in some detail, it is necessary to consider some more mundane questions as to what happens to the output from the evaluation process. This output is normally in the form of text, though in syntactic macro processors it may be in the form of a syntactic tree or some similar representation. The output from a macro processor is not normally an end in itself but rather represents an intermediate stage in the conversion of source text to machine code. However, it is always useful to allow the user to examine this output. A particularly helpful form is to give a listing of the output laid out so that each piece of macro-generated output is preceded by a print-out of the macro call that produced it. This macro call should be represented as a comment.

Few macro processors have explicit output statements for copying individual pieces of text to the output stream as it is normally the rule that text not forming part of a macro call or similar construction is automatically output. However, two examples of macro processors which do require explicit output statements are TRAC and meta-assemblers. It is very useful if a macro processor is capable of producing several separate streams of output with the ability of switching from one to another when desired. This facility would be useful if a macro processor was generating assembly code which consisted of declarative statements, in-line code and subroutines, all produced in a haphazard order. It would be useful, indeed in some cases necessary, to collect together the different types of output. This could be achieved by using a different output stream for each type, so that the assembler could then take these streams in the required order.

Some macro processors have an even better facility whereby the macro processor itself, rather than the processor which follows it, organises the macro-generated output into the desired order. XPOP, for example, has elaborate facilities for storing away pieces of text, either individually or in groups, and then subsequently retrieving these pieces of text. Many other macro-assemblers have a “REM” statement for generating code that is to be inserted at a point remote from the place it was generated. In other macro processors global character variables or even macro definitions can serve the same purpose, though these may not be suitable for storing strings that are so large that they require the use of backing storage. (Section 7.4.7 of Appendix A contains a concrete example of this technique.) In addition to these facilities, a macro processor should have a special output medium for the production of error messages.

4.5 The Scope of Macro Definitions

A primary consideration affecting the evaluation of text is the scope of macro definitions. The term *environment* will be used to mean the collection of macro definitions and similar constructions affecting text evaluation that are in force at any one time. Hence the “meaning” of a piece of text depends on the environment under which it is evaluated. Some simple macro-assemblers evaluate all text under the same environment. However, a considerable amount of power is gained by allowing the environment to vary throughout macro processing. In particular it is useful to have macro definitions with more limited scope than the entire source text, and, furthermore, as McIlroy points out, it is desirable to allow new macro definitions to be generated dynamically during macro processing. Hence it is desirable to allow a dynamic environment.

However, dynamic environments lead to a difficult problem that arises in several areas of macro processing. It will be called the *change-of-meaning problem.* The change-of-meaning problem arises if a piece of text is scanned by a macro processor, and assumptions about the text as scanned during the first scan are carried over to the second scan. As a concrete example of the change-of-meaning problem assume that in ML/I every time a macro is defined it is desired to perform a prescan of its replacement text and search for all occurrences of the macro-time assignment statement, MCSET, and replace each occurrence by compiled code (no doubt preceded by some special marker). In all subsequent calls of the macro this compiled code could be executed, thus saving the overheads of interpretation. However, this whole operation could be invalidated if the meaning of the replacement text and in particular the use of MCSET was redefined during subsequent macro processing. For example, any of the following would invalidate the compiling of MCSET statements:

a. Redefining MCSET.
b. Switching into warning mode or out of warning mode.
c. Defining new skip brackets which cause some occurrences of MCSET to be treated as comments.

The change-of-meaning problem can arise, in one area or another, in all macro processors which allow a dynamic environment. The problem tends to be worst in general-purpose macro processors, where no assumptions can be made about the syntax of the source text. However, even in macro-assemblers it is possible for the problem to arise, for example, if the replacement of a macro call or formal parameter is allowed to insert a symbol that indicates the start of a comment or character string constant.

4.6 Definitions with Restricted Scope

Two questions need to be answered with respect to the scope of macro definitions. These are:

a. Must a macro be declared before it is used?
b. What is the scope of a macro that is defined within the replacement text of another macro? Is it global or local?

If the answer “no” is taken to question a) and a macro definition is to apply to text that precedes it, then it is necessary to perform a prepass to pick up all macro definitions, and a second pass to evaluate the source text under this environment. Because of the change-of-meaning problem and implementation difficulties it would probably be necessary to forbid any dynamic changes in the environment during the second pass. Hence all macro definitions would apply to the entire source text (unless a facility similar to the ACTIVATE and DEACTIVATE statements of PL/I were added) and it would be impossible to redefine a macro. These restrictions, together with the overheads of a prepass, more than offset the marginal gain from not having to define a macro before it is used. (However, if it is necessary to perform a prepass for some other reason, such as lack of storage, then the question is rather more open.)

In answer to question b), it is definitely desirable to have global scope for definitions. This allows the user to write macros of a declarative nature, which, within their replacement text, generate macro definitions that are to apply to the rest of the source text. This technique is described by McIlroy and in Section 7.4.10 of Appendix A.

However, although global definitions are the more necessary, there are also uses for local macro definitions, especially if, as will be discussed in the next Chapter, macro definitions are to serve as macro-time character variables. There is a certain amount of choice in the meaning of "local". It can mean "confined to the replacement text of the macro in which it occurs" or "confined to the replacement text of the macro in which it occurs together with the replacement text of any macros called within that macro and/or the arguments of that macro". Each approach has its advantages and disadvantages, though these are of a rather obscure nature and are rather difficult to evaluate.

4.7 Skips

It is normally desirable to inhibit macro replacement within certain contexts of the base language, such as comments, character strings and literal strings which may look like macro calls. The same thing applies to the replacement of formal parameter names by the corresponding arguments.

Thus many macro processors allow constructions similar to skips in ML/I, which limit the scope of replacement. However, skips usually lead to a small sub-problem. If, for instance, character string constants are "skipped', then it becomes very difficult if, in one particular instance, it *is* desired to perform the replacement of a macro call or formal parameter within a character string constant.

5 Macro-Time Facilities

Every macro processor has a programming language of its own for giving instructions to the macro processor and for defining and manipulating macro-time entities. This Chapter considers some of the facilities that may be offered.

5.1 Macro-Time Variables

Nearly all macro processors have some facility for macro variables, i.e. variables operative at macro-time. In the same way as for macro definitions, it is desirable to have available both local macro variables and global macro variables. Global variables are needed to relay information from one macro call to another, and local variables are desirable for internal working within the evaluation of individual macro calls. If recursion is permitted, local variables are almost obligatory. Macro variables are normally of one of the three following types: character, integer, Boolean.

5.2 Character Variables

If a macro processor has sufficiently powerful facilities for defining macros then there is no need for character variables since macros will serve the same purpose. The conditions that must be satisfied for this to be possible are:

a. Macros can be redefined.
b. Macro calls may be written within macro-time statements.
c. Macros can be either global or local.

Failing these conditions, macro variables of type character are desirable. Since variable-length character strings present an implementation problem unless a macro processor is organised using list processing techniques, character variables are often given a fixed maximum length.

LIMP, which is implemented by list processing techniques, probably offers the best facilities for character variables present in any macro processor. A feature of LIMP is that within the replacement of a macro the arguments act as character variables and the same facilities are used to manipulate both arguments and other character variables. This results in the unusual and useful facility of being able to redefine arguments.

Another macro processor offering unusual features in this respect is IMP, which allows the user to manipulate a macro-time stack.

5.3 Integer Variables

In a macro processor, as in most other symbol manipulation programs, it is unnecessary to have numerical variables of any type but integer. Variables with integer values are useful for counting, for generating constants and as indices for switches and arrays. It is necessary to have facilities for performing macro-time arithmetic with these variables. However, if a macro processor allows character variables, it is not absolutely necessary that it allow integer variables as well, because integers can be stored as a string of characters and arithmetic can be performed on them in this form, though this may involve a certain loss of efficiency.

5.4 Boolean Variables

Boolean variables are rarely implemented since integer variables can serve the same purpose. However, System/360 Macro-Assembler contains Boolean variables and has very powerful facilities for using them.

5.5 Further Facilities

From the above discussion it can be seen that if macros are sufficiently powerful they can be used as character variables which in turn can be used as integer variables which in turn can be used as Boolean variables. Hence no macro variables are absolutely necessary. However, for reasons of efficiency, especially if macro-time statements can be precompiled (see Chapter 6), it may still be desirable to have macro variables. Another reason for having explicit macro variables is that they may easily be organised into arrays, whereas there are not many macro processors where it is easy to subscript a macro name. Most macro processors which have macro variables have a facility to allow them to be combined into vectors, though few macro processors, if any, allow multi-dimensional arrays.

Over and above the facilities described above, it is necessary to have the equivalent of what McIlroy calls *created symbols*. These are unique sequences of symbols created by the macro processor, which can be used when it is desired to generate a set of different names in the output text.

It is often useful for the macro processor to maintain predefined values or predefined initial values for some macro variables. ML/I, for instance, has a scheme for passing information to a macro in the form of initial values of its local variables, and System/360 Macro-Assembler has a special global character variable in which the name of the current "control section" is stored. Another useful feature might be to make the current line number available as the value of a macro variable, since this aids in the production of the user's own error messages.

5.6 Macro-Time Statements

It is necessary for the user to be able to write instructions to the macro processor for such purposes as to make a new macro definition, to transfer control (i.e. to GO TO) or to manipulate macro variables. It will be assumed in this discussion that these instructions are written as statements, though they could, in fact, be represented in other ways. As has been mentioned already, sone macro processors treat macro-time statements as system macros. In these cases the macro-time statements have arguments in the same way as other macros and these arguments are evaluated to replace any macro calls, formal parameters, etc., occurring within them. After this evaluation, arguments are usually treated as literal strings rather than as names of other strings, a point emphasised by Mooers [34] in a discussion on TRAC. However, not all macro processors operate in this way and many perform no macro replacement within macro-time statements.

There is a wide variety of possible macro-time statements. Every macro processor must, of course, contain a statement for setting up macro definitions, but the other kinds of statement are of a more optional nature. It must be possible to place definition-creating statements in the source text but the remaining statements may be restricted to occur only in replacement text. Such a restriction is not, however, very desirable, and it is better to follow McIlroy's dictum that all text should be treated in the same way.

In the discussion that follows arithmetic and control statements are both considered in some detail and then some other possible types of statements are more briefly considered.

5.7 Arithmetic and Control Statements

Nearly all macro processors have macro-time statements to perform arithmetic and to provide a conditional transfer of control. However, Strachey [38] has shown that, with sufficient ingenuity, the user of GPM can construct ordinary macros which perform arithmetic and provide a conditional GO TO facility, and no doubt the same applies to other macro processors. Such schemes as Strachey's, though, are very slow indeed and so in practice it is desirable to have built-in macro-time statements to perform these functions. McIlroy goes to the opposite extreme and says that macro-time statements should be as powerful as the statements to be found in algebraic languages. This view, which is reflected in the PL/I macro processor, has been discussed already. Macro-time arithmetic facilities are often offered in the form of a macro-time assignment statement. There is no need to allow very elaborate arithmetic expressions as it is very rare to perform complicated arithmetic at macro-time.

Macro processors differ considerably in their facilities for macro-time control statements (i.e. DO statements and (conditional) GO TO statements). The PL/I macro processor, which has elaborate IF and DO statements together with a GO TO statement, probably offers the most comprehensive facilities. Other macro processors, especially macro-assemblers, content themselves with a DO statement and a conditional statement that can skip one line. Macro processors which treat macro-time statements as system macros can make do with very primitive macro-time statements, provided that the user can define his own macro-time statements by designing suitable macros which expand into these primitive statements.

It is useful if some kind of multi-way switching facility is offered, either through the GO TO statement or otherwise, since multi-way decisions are very common in the generation of text to replace a macro call.

If there exists a general facility for a GO TO statement, there must, of course, be a corresponding facility for macro-time labels. Most macro processors make labels local to the text containing them but it would be useful, although more difficult to implement, to allow a more general facility for transfer of control from macro to macro, analogous to the GO TO statement in ALGOL.

In implementing macro labels and GO TO statements, it is most natural that on encountering a macro label the macro processor should remember its position in case that label is subsequently referenced in a GO TO statement. Although this is probably the best method of dealing with labels, it has its difficulties because of the change-of-meaning problem mentioned earlier. Examples of possible difficulties are:

a. The label name may be defined as a macro.
b. A new skip may be defined, which causes the label to end up in the middle of a comment.
c. The label may be multiply-defined but only one occurrence of it may have been encountered when a GOTO statement is reached.

If any of these problems can arise it is necessary to define the macro-time GO TO statement very carefully in order to state exactly what happens.

5.8 Further Statements

As well as assignment statements and control statements, there is a very large range of further macro-time statements or functions that may be offered.

String manipulation facilities are probably the most desirable of these. LIMP is very good in this respect in that it includes most of the facilities of the symbol manipulation language SNOBOL. Other macro processors limit themselves to providing a few functions for finding substrings.

Another facility which might be considered under the heading of string manipulation is the ability to manipulate scanning pointers. To achieve this facility, a macro processor is described in terms of hypothetical scanning pointers moving along the source text and the replacement text of macros, and the user is allowed to change these pointers if he wishes. This is a very powerful facility, perhaps too powerful, as it lets the user completely clobber the macro processor.

Another category of desirable macro-time facilities comes under the heading of I/O. Included in this category are facilities for producing the user's own diagnostic messages, for saving text on backing store and for interfacing with a library. Some aspects of this were discussed in Chapter 4.

Lastly the XECUTE mode available in XPOP should be mentioned. This is a facility whereby the user can supply a sequence of FAP-like instructions that is to be executed immediately on being scanned by XPOP. By this means a sufficiently knowledgeable user can patch XPOP itself and add new facilities to it.

5.9 Macro-Time Dictionaries

Shaw [36] in an unpublished critique of XPOP puts the case for a dictionary facility at macro-time. Shaw states that the two main features that make high-level programming languages more powerful than assembly languages are:

a. *One-to-many operators.* One high-level operation may be equivalent to a sequence of several assembly instructions.

b. *One-to-many operands.* In assembly language an operand usually stands simply for a machine address. In high-level languages, on the other hand, the name of a variable also stands for all its attributes. For example, when one writes:

   ```
   A + B
   ```

   the symbol A not only defines a machine address but also conveys the information as to whether the operand it represents is fixed point or floating point, double precision or single precision, local or global, and so on. This information is supplied once and for all in the declaration of the variable A, and is not repeated every time A is used.

Shaw's one-to-many operands are, in fact, a particular case of the context-dependent replacement mentioned earlier. Of Shaw's two characteristics, macro processors are good for one-to-many operators, but generally poor for one-to-many operands, and, in fact, for most forms of context-dependent replacement, in particular for optimisation. In order to alleviate this deficiency some macro processors contain some sort of dictionary facility.

In fact many macro processors that do not explicitly have a dictionary facility can be made to simulate such a facility. If one wants to remember the attributes of a variable one can define a macro with the same name as the variable and store information about the attributes of the variable as the replacement text of the macro. This technique is illustrated in the paper on ML/I [4]. Bennett and Neumann [2] show how the same thing can be accomplished by a cunning method of building up names by concatenation. However, such techniques as these are rather artificial and in many cases slow in execution, and so there is a good case for an explicit dictionary facility in a macro processor.

LIMP offers such a facility. In LIMP the user has absolute control over the building and interrogation of the dictionary, which is represented as a tree. However, LIMP is a one-pass system and, if its dictionary facility is to be useful, variables must be declared before they are used.

The System/360 Macro-Assembler, taking advantage of its base language dependence, offers an even better dictionary facility. This macro-assembler, as has been mentioned already, automatically performs a prepass to build a dictionary of all the assembly language variables declared in the source text. This dictionary may be interrogated when macro calls are replaced. However, if new variable declarations are generated by macro replacement these do not appear in the dictionary.

Unfortunately if an attempt is made to generalise this technique, logical difficulties arise due to the change-of-meaning problem. This is especially true if the technique is applied to a high-level language where variables can have limited scope. In this case the scope of declarations picked up on the prepass can be altered by material generated during the second pass, for example the introduction of overriding declarations or of new BEGINs or ENDs. Hence decisions based on the contents of the dictionary can later be invalidated.

6 Implementation Methods

Three central considerations in the implementation of a macro processor are:

a. How many passes?
b. Are macro-time instructions pre-compiled?
c. Are list processing techniques used?

These three questions are discussed in detail in the material that follows.

6.1 Multi-Pass Macro Processors

Most macro processors are logically one-pass, i.e. the source text need only be scanned once. It has been argued previously that a prepass especially to pick up macro definitions is not generally desirable. The only macro processor that has been considered to gain from being two-pass is the System/360 Macro-Assembler, which builds a dictionary on the first pass. Apart from such special-purpose considerations, the only good reason for a multi-pass macro processor is the lack of enough storage to do everything in one pass. However, although a macro processor should be one-pass, it should still be usable for those jobs that require several passes through the source text. To make this possible, a macro processor should be capable, at the end of one pass, of re-entering itself and performing a second pass using some of the information derived on the first pass. This would make it suitable for multi-pass applications and would allow the actions of the passes to be defined by the user rather than being fixed by the design of the macro processor. Furthermore, the overheads of performing two passes would not be inflicted on the jobs requiring only one. In order to be fully suitable for multi-pass applications, a macro processor should have some further properties over and above the capability of re-entering itself. In particular since each macro-time statement in the source text will usually be executed on one pass and ignored on all the other passes, the user should have the ability to redefine macro-time statements to have null effect.

6.2 Pre-Compiling of Macro-Time Statements

Two kinds of macro-time statement can be distinguished. Firstly those, like macro definitions, which tend to occur in the source text and are executed only once, and secondly those, like arithmetic and control statements, that tend to occur within replacement text and are executed many times. Statements of the second kind will be called *repeat-prone* statements.

There is clearly no case for compiling the first kind. However, in the case of macro definitions that are used by many jobs the overheads of setting up the definitions every time can be avoided if there is a facility for pre-editing definitions into the system. This is specially important in syntactic macro processors, where the setting up of a definition may be a lengthy process.

There is definitely a case for replacing repeat-prone statements by compiled code rather than interpreting them every time they are executed. In jobs where complicated macros are used, a large proportion of the time is spent in executing these statements and the speed of macro processing can be increased very considerably by compiling them.

Moreover, a macro processor which compiles repeat-prone statements is not much more complicated than one that interprets them. After all, the only difference between interpreting a statement and compiling it is that in the former case the statement is translated into some instructions which are executed immediately whereas in the latter case these instructions are stored away to be executed later. It is quite easy to make the same routine capable of either interpreting or compiling. The replacement of macro-time statements by compiled code can be performed in any of the following ways:

a. Repeat-prone statements are compiled just before they are executed for the first time.
b. A prepass of the entire source text compiles all repeat-prone statements, including those within macro definitions.
c. Every time a macro definition is set up, all the repeat-prone statements within its replacement text are compiled.

The main difficulty with method a) is that there may be an implementation problem in replacing a piece of text that is embedded in other text by code that may be a different size to the original text.

Method b) has all the disadvantages of a prepass.

Method c) is probably the best for most implementations. It does require that some repeat-prone statements be interpreted, notably those that are dynamically created and those which occur in the source text, but this is no great disadvantage as these statements will not occur very often and, as has been pointed out, it is quite easy to make the same routine capable of both compiling and interpreting.

Compilation presents only one serious problem, namely the change-of-meaning problem. This problem is worst in macro processors which treat macro-time statements like any other text and allow macro calls within macro-time statements. If compilation is to be possible the user must be forbidden from subsequently changing the meaning of the compiled statements. The restrictions necessary to accomplish this may be quite straightforward and easy to enforce in a special-purpose macro processor, but in a general-purpose macro processor the necessary restrictions may be very difficult to define and almost impossible to enforce.

Nevertheless, in nearly all practical applications the user will not change the meaning of his repeat-prone statements and hence if these statements are interpreted, the price of interpretation is paid without the buying of its advantages. It is an unfortunate fact that programmers have become trained in other areas never to change the meaning of their statements, and when they come across an interpretive system they are reluctant to take advantage of it and are, in any case, completely unfamiliar with some of the powerful techniques that can be used.

6.3 List Processing Techniques

One of the most basic decisions to be taken in the implementation of a macro processor is whether text is to be stored in the form of lists of characters or is to be stored contiguously using one or more stacks. A macro processor implemented by list processing techniques should offer more powerful symbol manipulation facilities and it should be relatively easy to allow macros and other entities to be deleted or redefined. LIMP well illustrates the power that can be gained from using list processing.

However, the disadvantage of list processing techniques is that the macro processor becomes rather slow at doing the simple things and, furthermore, rather wasteful of storage. Since macro processors are usually rather slow at the best of times, the extra overheads of list processing may make a macro processor too slow to be a practical tool for some of its potential applications.

6.4 Further Considerations of Speed

The implementation considerations discussed already have a large impact on the speed of a macro processor. However, the effect on speed of three other considerations should also be mentioned. These considerations are: method of argument treatment, recognition method, and use of backing storage.

The "call by name" method of argument is faster than the other two methods provided that an argument is only inserted once into the replacement text of the macro to which it belongs. This is because in the other methods when an argument has been evaluated it has to be stored away somewhere until it is inserted, whereas in "call by name", no special action need be taken when an argument is first scanned and the argument is evaluated only when it is inserted; at this time its value can be copied directly over to the output text.

A good deal of time may be spent by a macro processor in recognising calls. The overheads of recognition are least if macro calls are introduced by an explicit warning marker, as in GPM, and greatest if calls are only recognised by identifying a macro. In LIMP, for instance, each line of text has to be compared with the pattern associated with each macro definition, though the recognition process is speeded up by the use of trees. In ML/I, if in "free mode", each atom of the source text must be compared with all possible macro names, though this, too, can be speeded up by a special technique, namely the use of "hashing".

The last consideration concerns the output text. The output from a macro processor is normally sent to backing storage so that it can subsequently be fed to some compiler or assembler. In addition to this, a multi-pass macro processor might require backing storage for its intermediate output. As a consequence of this, the effective speed of a macro processor may depend to a large extent on the efficiency of the use of backing storage. Freeman [13] discusses some of the considerations in this area.

If a macro processor is small enough to share core with the piece of software which is to receive its output then the need for intermediate output can be avoided by organising the macro processor and the piece of software as co-routines. (For a description of the concept of a coroutine see Conway [7].) This might represent a considerable saving of time and hence is a point in favour of small and simple macro processors.

6.5 Further Storage Considerations

The design of a macro processor involves the usual, machine-dependent questions as to whether character data should be "packed" and whether backing storage should be used for certain data, for example macro definitions.

One special consideration affecting the use of storage in a macro processor is the design of the macro-time GO TO statement, if there is one. If it is possible to perform a backward

GO TO in the source text then it will not be possible to discard the source text after it has been scanned over. Since this is a big overhead it is advisable to forbid backward GO TO's in the source text, but perhaps to allow a "DO" statement instead.

7 Conclusions

Now that the design of macro processors, both existing and hypothetical, has been considered, the application areas introduced in Chapter 1 will be reviewed to see what has been achieved and what can be achieved. Before this is done, however, three general points should be made.

Firstly it should be remarked that it is still useful if a macro processor can be used in a particular area even if jobs in that area can be done more easily by specially designed software. Thus a macro processor capable of context editing is useful, even though a context editor might do the task better, because:

a. An installation may not possess a special-purpose context editor.
b. Users familiar with the macro processor might prefer to use it to save the trouble of familiarising themselves with a context editor.
c. The macro processor will be useful for jobs on the borderline between context editing and conventional macro processing or jobs involving an element of each.

A second general point concerns the ease of use of macro processors. Most macro processors are in principle quite simple. However, they are completely different from any other type of software and users may require some help and encouragement to start with. Confusion may arise with the concept of macro-time statements and the difference between these and base language statements. In a number of macro processors, replacement text often involves the use of unusual characters (e.g. $, ~) for special purposes. This makes text hard to read, and may act as a disincentive to the intending user. Another problem for the tyro may be the detection of errors within macros, as facilities for detecting errors and recovering from them are not always satisfactory. Overall, however, there is no reason why text macros, at least, should not be a tool that every programmer is capable of using. However, syntactic macros may be more difficult to write and might be considered as a tool only for systems programmers.

Thirdly, there are many applications where it might be thought desirable for the user of a macro processor to be unaware of its existence. For example, a programming language P may be implemented by mapping it into a language B using a macro processor, and then using the compiler for B to map the resultant text into machine language. Ideally the user of P should be unaware that his text was mapped by macro replacement into B, any more than the user of a compiler is aware of the mechanisms used for compilation and the intermediate forms of text. However, this ideal is not usually attainable in practice since error messages produced by the macro processor will be in terms of macros and, unless the macro processor detects all errors, error messages will be produced by the compiler for B in terms of the macro-generated text. This example illustrates the general point that the user usually needs to know something of what is going on behind his back.

Discussions of the individual application areas follow.

7.1 Conclusions on Language Extension

The prime purpose of macro processors is to extend languages, and hence all macro processors have reasonable facilities in this respect. Many macro processors are, however, capable of introducing new statements into the base language but incapable of dealing with any other syntactic class. This restriction is reasonably acceptable if the base language statements have a very simple structure, as in assembly languages, but is less acceptable if the base language is a high-level language. In high-level languages it is desirable for a macro processor to be able to expand syntactic classes that form part of statements, for instance expressions. Even in assembly languages it is useful to be able to macro-generate variables and constants. Hence macro processors with no restrictions on the text to be expanded have advantages over those that only deal with statements, and the higher level the base language the greater these advantages are. However, even the most general macro processors have severe limitations when they are used to expand parts of high-level language statements. The two greatest problems are:

a. *Notation.* All existing text macro processors use bracketed notation for macro calls whereas this notation is foreign to high-level languages. Within arithmetic expressions, for instance, high-level languages use the relative priorities of arithmetic operators to determine the bracketing structure.
b. *Difficulty with replacement.* The uses of macros are restricted because few high-level languages allow statements to be written within other statements.

Problem b) is best illustrated by an example. Assume it is desired to introduce a macro to multiply two complex numbers. Let a call of this macro be written

```
A CMULT B
```

where each argument is the name of a vector with two elements. Then if the macro is to generate in-line code the macro call should be replaced by something such as:

```
(begin TEMP (1) = A(1) * B(1) - A(2) * B(2);
       TEMP (2) = A(1) * B(2) + A(2) * B(1);
end)   TEMP
```

(This means that the two statements should first be executed to assign values to the two elements of the vector TEMP and then TEMP should be used as an operand.)

Unfortunate1y, few high-level languages allow the insertion of text such as the above into the middle of statements. One of the languages that does allow “statements within statements” is GPL, a language specially designed for extendability. *It is suggested that this feature is desirable in all high-level languages.*

The introduction of new operators and data types is one particular case of the extension of high-level languages by macro techniques that has received a good deal of attention recently and is worthy of more detailed consideration. For a start it should be remarked that text macros can only be used to introduce new data types that are expandable in terms of existing data types, and the same will usually apply to syntactic macros as well.

Similarly, text and syntactic macros can only be used to introduce operators describable in terms of the existing base language. However, most high-level languages already contain facilities for introducing new data types and operators expandable in terms of the base language. In PL/I, for instance, new data types formed of combinations of existing data types can be built up by using “structures” and new operators can be introduced by using “generic functions”. Thus a good deal of extension can be done without the aid of macros. This fact, together with the fact that, when it is desired to add a new data type to a language, this is often because that data type *cannot* reasonably be expanded in terms of existing data types, indicates that the importance of macros (other than computation macros) in defining new operators and data types may be over-exaggerated. In fact, the only advantages that may be gained from using macros to introduce new operators and data types to a high-level language already containing reasonable features for defining functions and data aggregates are:

a. Efficiency can be improved in many cases by generating in-line code rather than function calls.
b. Notation can be improved, for example by using infix operators rather than functional notation.
c. Existing polymorphic operators can be extended to cater for new data types.

(In high-level languages with exceptionally good self-extending facilities, it is even possible to achieve b) and c) above without the use of macros.)

Existing text macro processors are not fully capable of achieving any of the three above advantages because of difficulties with notation and replacement. However, this is not to say that text macro processors are completely useless in these respects. To quote one example of the introduction of new operators, text macros could be used to add to PL/I the ALGOL facility for conditional expressions, i.e. expressions of form:

```
    (if ... then expression 1 else expression 2)
```

It would clearly not be feasible to implement this facility by using functions. As further examples of the use of text macros in this way, Hopewell [22] describes an extensive set of macros for enriching Titan Autocode.

If syntactic macros are examined with reference to the three advantages above, they come out well on notation but may have limitations in the other respects due to the difficulties with replacement that have already been mentioned and due to the fact that information about the data type of variables may not be available when syntactic analysis is performed. However, the proposal by Galler and Perlis, which has been designed solely for the introduction of new operators and data types, is satisfactory in all three respects, but this proposal requires special extensions to be made to the base language, which is ALGOL.

In conclusion it can be said that no macro processor is fully suitable for the extension of high-level languages, though syntactic macro processors would represent a step forward from text macro processors for high-level languages with suitable grammars. Perhaps the best answer to the problem is to follow the philosophy of GPL and build better self-extending facilities into the languages themselves.

7.2 Conclusions on Language Translation

In principle any macro processor that is both general-purpose and notation-independent should be capable of translating between arbitrary languages provided that the rules for translation can be specified. However, in practice there are considerable limitations. This is because the rules for translating between languages usually turn out to be very complicated, with numerous special cases, and the mechanisms offered by macro processors tend to be too slow and too inflexible to perform a complete translation. This is, of course, a defect of any “general-purpose” software. Successful translation operations performed by macros tend to be very simple, like Dellert’s [8] translation between IBM 7090 and IBM 7040 assembly languages. The difficulties are well illustrated by considering the use of a macro processor as a compiler, i.e. as a translator from a high-level language to a machine language. The basic operations in compiling are reasonably simple, and can be performed by a macro processor. However, a good deal of the work in writing a compiler is not concerned with performing general operations on data, but is concerned with particular cases that do not fit into any general pattern. The mechanisms of macro processors, though good for general cases, usually turn out to be too inflexible to deal with special situations, in particular with exceptions to general rules. Macro processors, as Shaw has pointed out, are generally weak in performing context-dependent replacement. Furthermore, it is often desirable to organise a compiler in several passes, perhaps producing intermediate text in the form of a tree, and it would be very hard to make a macro processor do this. However, this view of the inadequacy of a macro processor for compiling is not shared by Halpern [19]. He argues that, except for algebraic languages and other languages that can be guaranteed not to change, the best way of compiling is by using macro techniques, and that the main use of macro processors is as general-purpose compilers, capable of compiling any language, and allowing the user to extend any language thus compiled.

One area of application where notation-independent macro processors are very useful is the area in between language translation and language extension. In this type of application the user makes up a programming language of his own design and oriented towards his own field of application and uses the macro processor to translate this language into some base language. This cannot be regarded entirely as an application in language translation since in practice the user, when designing his language, will bear in mind the characteristics of the base language and the translating capabilities of the macro processor, and hence there is an element of “language extension” about it.

In conclusion it can be said that macro processors can certainly be useful in language translation if the translating rules are fairly simple, but if these rules are very complicated the usefulness of a macro processor (or indeed of any general-purpose software) becomes more doubtful. Note that the rules for translation between two languages can be very simple even though the languages may look very different. For example, it is quite trivial to translate between fully parenthesised algebraic notation and Polish Prefix notation.

7.3 Conclusions on Text Generation

To recapitulate on Chapter 1, the five expected capabilities of a macro processor in the field of text generation were: conditional generation, repetitive generation, program parameterisation, simple applications in report generation, and library facilities. The last of these, library facilities, requires special purpose machinery that depends on the operating environment in which the macro processor is embedded. There is a case for having two kinds of library — one for pieces of text and one for pre-compiled macro definitions.

In the remaining four fields any macro processor containing macro-time looping statements and macro-time conditional facilities should be adequate, although a special-purpose macro processor will be limited to generating text in its base language and a macro processor that requires each macro call represent a base language statement will be considerably less useful than a macro processor not having this restriction. Fletcher [12] supplies a very interesting example of the power of a macro processor in the field of text generation, and Magnuson [31] further examples in a paper containing several ingenious and potentially very useful examples of the capabilities of a macro processor. Keese [27] shows how a macro processor can aid in the automatic production of documentation.

7.4 Conclusions on Editing and Searching

A macro processor must be notation-independent if it is to be used to any great extent for systematic editing and searching, but if it is notation-independent it can be a very powerful tool in this field. If the capabilities of macro processors in the field of systematic editing are compared with those of a context editor then the following points arise:

a. Context editors should be easier to use since they are specially designed for editing.
b. Macro processors are not good for making isolated changes in text but are only useful for systematic changes.
c. Macro processors may be superior in applications where they can take advantage of their subsidiary features. As specific examples to give an idea of what is meant by this assertion, a macro processor might be capable of identifying the following entities in a program, whereas a context editor would not: (i) array bounds that are integers greater than some specified value; (ii) declarations of logical variables and all subsequent uses of these variables.

7.5 Final Conclusions

Macro processors have a very wide range of applications, and the first consideration in the design of a macro processor is whether to aim for the largest possible generality or to concentrate on one area and to provide facilities that are especially suitable for that area. Both approaches have been successful. In the case of a macro processor with general capabilities, a great deal of power can be gained using relatively few primitive operations and the range of applicability often turns out to be considerably wider than that envisaged by the original designer. In fact it is often something of a game to try to reduce the number of primitive operations to an absolute minimum. Overall, the ratio of usefulness to implementation effort required is higher for macro processors than for most other software.

The greatest contribution that can be made by macros in the future is probably in the area of standardisation. It is almost impossible to agree on a standard programming language, but it may be possible to agree on a standard base language and a standard macro processor to allow extension of this language. Although a good deal of work has been done on the design of macro processors, very little has been done on the design of base languages that are easy to expand. Indeed, this aspect of the design of a language has often been completely ignored in the past. In future, as the computer attracts more and more lay users in more and more diverse fields, this aspect must receive more attention.