Programming Languages

N. WIRTH, Editor

A Language Independent Macro Processor

WILLIAM M. WAITE
University of Colorado, Boulder, Colorado*

A macro processor is described which can be used with almost any source language. It provides all features normally associated with a macro facility, plus the ability to make arbitrary transformations of the argument strings. The program is used at the Basser Computing Department, University of Sydney, Sydney, Australia, to process text for eight different compilers.

1. Introduction

The term "macro" was first used to denote a feature of certain assembly languages which allowed a programmer to refer to a group of instructions as though they were a single instruction. By mentioning the name of the "macro-instruction," the programmer caused all of the component instructions to be inserted at that point in his coding. The possibilities of this sort of redefinition were soon realized, and a classic paper by McIlroy [1] showed how an appropriate set of macro instructions could allow the user to tailor an assembly language to his needs at a quite high level. Until recently little more was said on the subject, and most of the operations proposed by McIlroy were used routinely in a number of assembly languages. The problem of reprogramming has been responsible for a resurgence of interest, and several papers [2–5] have appeared during the past two years.

One of the significant features of the current papers is the concept of a macro processor which is independent of any particular assembly language. Macro processing is viewed as a type of string manipulation—a character stream fed to the input is analyzed according to certain rules, and a character stream is produced as output. The processor described in this paper is designed to operate as such a string manipulator, and its output is to be presented to some compiler or assembler. Because it can deal with almost any input text, it has been named LIMP (*L*anguage-*I*ndependent *M*acro *P*rocessor).

This work was performed at the Basser Computing Department, University of Sydney, Sydney, Australia, and was supported by the U.S. National Science Foundation under Postdoctoral Research Fellowship No. 45070.

* Department of Electrical Engineering, University of Colorado

In classical macro processors such as that described by McIlroy, a macro definition has the form:

```
MACRO NAME (P1 , P2 , ··· , Pn)
Code body
END
```

Here the words "MACRO" and "END" serve to delimit the definition, "NAME" is the macro name, and "P_1" through "P_n" are formal parameters. The code body is a symbol string which may contain instances of the formal parameters. The form of a call on this macro would be "NAME (A_1, A_2, $\cdots$, A_k)" where $k \leq n$. Such a call is replaced by the code body of the macro "NAME", with the actual parameter strings A_1, A_2, $\cdots$, A_k being substituted for the corresponding formal parameters P_1, P_2, $\cdots$, P_k. If $k < n$, then the processor generates actual parameter strings for P_{k+1}, $P_{k+2}, \cdots, P_n$ in some regular way.

LIMP generalizes the notion of a macro definition and macro call. The line which introduces a macro definition may be any arbitrary character string, with parameters indicated by a special parameter symbol. We refer to this line as a *template*. A template essentially allows the name part of a traditional macro to be replaced by a "distributed name" consisting of all portions of the line other than the parameter markers. This approach frees the system from any arbitrary format restrictions of a particular language —templates can be written in a form suited to the language at hand.

Macro calling is accomplished by pattern matching against the set of templates defined by the user. Any line which cannot be matched is copied directly to the output without change. When a match occurs between an input line and a template, the code body corresponding to the template is evaluated with the actual parameters resulting from the template match. The result of this evaluation replaces the matched line in the output text.

Correspondences between actual and formal parameters are set up during template matching. The template is a sequence of fixed strings separated by "holes" (parameter markers). When the matching process is complete, each parameter marker will correspond to some substring of

the input line; the fixed strings of the template will exactly match other substrings of the line. The string matched by a given parameter marker becomes the actual parameter corresponding to that given formal parameter.

The code body of a LIMP macro consists of a series of statements in a language almost identical to SNOBOL [6, 7]. Formal parameters are specified by their relative addresses in the template, up to 9 formal parameters being allowed per macro. (The relative address is a digit between 1 and 9 inclusive, which specifies the number of the parameter marker in the template, counting from the left.) Whereas conventional macro processors permit only one form of substitution of actual for formal parameters, LIMP allows several alternative forms. The type of substitution is specified for a given instance of an actual parameter by a subscript on the relative address.

The structure of LIMP allows it to be used as a preprocessor for almost any compiler or assembler. It was incorporated into the KDF9 operating system developed by the Basser Computing Department in April, 1966 [8] and since then has been used with programs written in eight languages (three variants of ALGOL, two assembly codes, a list processor, a flowcharting language, and an autocode). No change in LIMP is required for any of these languages.

The input to LIMP consists of a series of lines. (On card-oriented machines each card is a line; on paper tape, lines are delimited by carriage-return characters). The first line defines the set of control characters for the run. This *flag line* is necessary to preserve the language independence of the processor; different languages generally require different control characters.

The first character of the flag line is the *end-of-line flag*. If it appears on any subsequent line, its effect is that of a carriage return—the remainder of the line is ignored by LIMP. This allows the use of any text as commentary on any line. If "space" is specified as the end-of-line flag, LIMP assumes that *no* end-of-line character is desired.

The second character of the flag line is the parameter marker. It is only recognized as a control character when a template is being entered into the set of macro definitions; at any other time it is treated as a normal character. In this paper, the parameter flag will be represented by "*", and the end-of-line character by ".".

After the flag line, the user defines his macros. At least one definition must follow the flag line, and further definitions may appear anywhere in the text. Each macro must, of course, be defined before it is called.

The free format of the input line and the large number of templates involved create possible ambiguities in the matching process. Section 2 describes the template-matching algorithm in detail and shows how the scanner resolves these ambiguities. An informal specification of the code body statements can be found in Section 3, where the various types of parameter conversions are defined. This section gives a brief review of the features of SNOBOL which are crucial to the understanding of LIMP. In Section 4 a number of examples of LIMP macros are given, chosen to illustrate the basic principle and some of the unique features of the language. Section 5 outlines the construction of the processor and gives an indication of the direction of future work on this project.

2. Template Matching

Matching a single template against an input line is a special case of the following general problem [9]: "Given a pattern $\{e_1 e_2 \cdots e_n\}$ and a string S, locate consecutive substrings s_i in S such that each s_i is an acceptable value of the pattern element e_i." Pattern elements in LIMP templates are either fixed strings or parameter flags. Each element can be assigned a *weight*, w_i, equal to the length of the shortest acceptable value of e_i. If e_i is a fixed string, w_i is its length; if it is a parameter flag, $w_i = 0$ (because a parameter flag can match the null string).

In general we shall have a whole set of patterns which are candidates for the matching process. These patterns may be grouped into a tree with each element corresponding to a branch; the branches leaving a given node can then be ordered according to the weights of the corresponding elements.

Often there may be several ways for an input line to match a given template, and a given input line might match any one of several templates. For example, the line "X = Y = Z." would match any of the templates "* = *.", "X = *.", "X = Y = *." and many more. Moreover, it could match "* = *." in two ways: With actual parameter 1 equal to "X = Y", or with actual parameter 1 equal to "X". Such possibilities complicate the matching process and require further specifications to eliminate ambiguity.

In order to define uniquely which template will match a given line, and which of the possible matches to this template will be chosen, the following rules are used.

Rule 1. A space which is not adjacent to a parameter flag matches any nonempty string of spaces, and spaces at the end of a line are ignored unless the last character of the matching template is a parameter flag.

Rule 2. The matching process proceeds from left to right with each pattern element matching the shortest possible substring.

Rule 3. At a node, matches are attempted for the branches in order of decreasing element weights.

Rule 4. If no element at a node can match a substring, then a new match is attempted at the previous node. This new match is accomplished by extending the substring formerly matched to the next shortest acceptable value for the same element. If this extension cannot be made, a match for the next element at that node is attempted. If no element remains, rule 4 is applied to that node.

The pattern match succeeds when the last character of the input line has been matched; it fails when no match can be found for the first character.

As an example of the matching process, consider the set of templates

(1) SAM = A.
(2) SAM = *.
(3) * = *.
(4) * = A.
(5) * = * = *.

The tree for this set is shown in Figure 1, where a lower case "b" is used to denote a space.

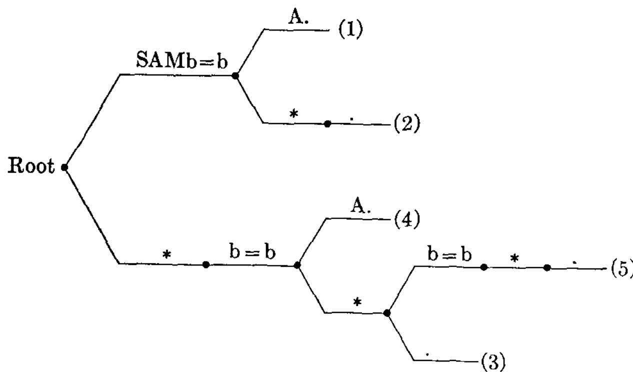

Fig. 1. An example of a template tree

The input line "SAM = A." will match template (1), as will any line which has a nonempty string of spaces between "SAM" and "=" or between "=" and "A.". The effect of Rule 1 is thus to make a string of spaces significant, but the exact number of spaces in the string is immaterial. (When a template is read and merged with the tree any string of spaces is replaced by a single space.)

Suppose that the next two input lines were "SAM = B." and "SAM =   C.", where there are six spaces between "=" and "C" in the second line. Both lines match template (2), and in the first line the value of the actual parameter will be "B". In the second line, however, the value of the actual parameter will be "   C"—five of the six spaces have been included in the actual parameter. This is because the space in template (2) between "=" and "*" is adjacent to a parameter flag: A space adjacent to a parameter flag matches exactly one space, so that the remaining spaces must be absorbed by the parameter. No spaces are actually deleted from the input line during the matching process, so that if no match is found all spaces are preserved when the string is written out.

At first glance one might assume that the input line "B = C = A." would match template (4) with "B = C" as the value of the actual parameter. Consideration of the rules will show, however, that this line will be matched to template (5).

The crucial decision occurs at the second node of the lower trunk of the tree At this point we have matched "B" to the first parameter flag and "b = b" to the fixed portion of the template, as required by Rule 2 ("B" is the

TABLE I. String Names Available in Limp

Name	*Function*
dc $(1 \leq d \leq 9,\ 1 \leq c \leq 3)$	actual parameter d, conversion c
0i $(1 \leq i \leq 9)$	current value of location counter, minus i
IN	effective input line
PR	printing output stream
PU	punching output stream
PT	private tree
ST	symbol tree
TT	template tree

shortest possible matching substring for the first parameter). According to Rule 3 the next match to be attempted must be for "A.".

The match for "A." fails because the next character of the line is "C". So we attempt a match for the parameter flag. Now the only way in which we can extend the match for the first formal parameter is by repeated use of Rule 4. But this requires that we be unable to find *any* match at all nodes further along the branch. Since the line matches template (5), we will not get the chance to apply Rule 4 at all.

3. Code Body

When Limp matches an input line to a template, it creates actual parameter strings and then executes a series of statements making up the code body of the macro. The statements of the code body are written in a language which is almost identical in form to Snobol. There are two reasons for this choice: (1) The operations available in Snobol allow very general manipulations of the actual parameters. (2) The syntax of Snobol effectively separates the metacharacters from the strings being manipulated and thus preserves the language independence of the processor.

The general form of a statement in the code body of a Limp macro is:

⟨label⟩⟨string⟩⟨pattern⟩ = ⟨replacement⟩/⟨go-to⟩

As in Snobol, a statement label must begin in the first character position of the line and may consist of any string of letters or digits. The label terminates at the first space; if the first character of the line is a space, it is assumed that the statement has no label. The next constituent of the statement is a string reference. In Snobol a string may be given almost any symbolic name, but in Limp the programmer may use only a fixed set of names as summarized in Table I. The only strings which may be used as working storage during the execution of a code body are those corresponding to the nine possible actual parameters. System strings (such as IN) have special properties which disqualify them for use as working storage. The private tree (PT) can be used to store intermediate results when more than nine strings are needed.

TABLE II. Pattern Elements Accepted by Limp

Type	*Format*	*Matches*
Literal	'any string'	itself
Constant	dc ($0 \leq d \leq 9$, $0 \leq c \leq 9$)	a substring containing the same characters as the specified constant string.
Arbitrary	dA	any string or substring, including the null string
Balanced	dB	a non-void string which is balanced with respect to ( )
Fixed-length	dFk	any string of length k (k may be any number of digits)
Back-referenced	dR	the string which was previously matched to variable d

Essentially this tree constitutes the "backup store" for the macro processor; it is used only by code bodies and is never altered by any part of the system.

The third constituent of the statement, the pattern, may be absent. If present, it consists of a series of pattern elements separated by blanks. The allowed pattern elements are listed in Table II. When a literal string is read, the procedure for handling quotes is as follows: The first quote marks the beginning of the literal. Any subsequent string of n consecutive quotes is replaced by a string of k consecutive quotes, where k is the greatest integer not greater than $n/2$. If there is a nonzero remainder from this division, the literal terminates. A literal may extend over several lines, in which case a carriage return character is inserted in the literal at the end of each line. This allows the programmer to write a single literal which produces several lines of output.

If a pattern is specified in a statement, the pattern is tested against the string named by the string reference. If the pattern can be matched to some substring of the string reference, strings are created for each variable in the pattern. Also, the matching substring may be altered by specifying a replacement. This replacement may contain the variables defined by the pattern match; see [6, 7] for further details. Note that if no replacement is desired, both the "=" and the replacement string should be omitted. If "=" is present but no replacement string is specified, the null string is assumed.

A statement may stretch over several lines. If the line changes occur within a literal, they result in carriage return characters as noted above. If they fall between the constituents of the statement, they are treated as spaces. Constituents must be separated by at least one space or carriage return, and additional spaces or carriage returns are ignored. The "/" preceding the go-to field *must* be present whether the go-to is used or not—it serves notice that the current line is the last one for the current statement.

The go-to field may be absent, but if present it takes one of the following forms:

(⟨label⟩) —transfer control to ⟨label⟩ unconditionally
S(⟨label⟩) —transfer control to ⟨label⟩ if the pattern match was successful
F(⟨label⟩) —transfer control to ⟨label⟩ if the pattern match was not successful.

Both conditional jumps may be present, in either order. Spaces are ignored everywhere in the go-to field.

A code body may contain any number of statements, the last of which is distinguished by the label "END". This last statement need not contain any other constituents, including the slash. Of course it may also be a normal code body statement, in which both the slash and string reference are required.

The conversion digits, denoted by "c" in Table I, describe the way in which the actual parameter string is to be used in each particular substitution instance. The meanings of these digits are ($1 \leq d \leq 9$):

d0 —use an exact copy of the actual parameter string.
d1 —use an exact copy of the actual parameter string; if this conversion appears in a string reference, any pattern specified must match a substring which begins at the first character of the referenced string. (This is equivalent to "anchored mode" in Snobol.)
d2 —look up the string on the symbol tree and use the value found; if the string is not in the symbol tree, use a null string.
d3 —same as d2, except that if the string is not found, it is added to the symbol tree and given the current value of parameter 0 (the "location counter"). Parameter 0 is incremented by one following this definition.

The symbol tree is established either by the use of type 3 parameter calls or by direct action on the part of the programmer. The symbol tree has the name ST, which can be used as the string reference in any statement. Some caution must be exercised, however, because ST is a tree rather than a string. The pattern must begin with a set of constant elements. This constant portion, consisting of all elements up to the first variable, will be tested against the tree and must match some entry all the way from the root to a leaf. The remainder of the pattern will be tested in the usual way against the string attached to the leaf. Any replacement will only be made to that part of the pattern which is *beyond* the leaf. This constraint allows the programmer to change the definition of a symbol, but not to delete a symbol from the tree. The entire tree may be deleted by "ST = /", and a symbol may be added by "ST = ST ⟨specification of symbol⟩/". If the added symbol is to be defined, a second statement must be used.

The system maintains a location counter which the programmer can use in several ways: (1) Type 3 conversion can be used to assign the current value of the location counter to a symbol. The location counter is automatically incremented by 1 following the assignment. (2) A set of local labels can be obtained by using parameter 0 explicitly. A reference to string 0i is taken as the current value of the location counter minus i. If the programmer

uses such constructions, he is responsible for incrementing the location counter by the maximum value of i, plus 1, *before* the first such reference is encountered. This can be done by the statement "00 = (00 + 'i' + '1')/".

As shown in the statement above, integer arithmetic can be performed on strings in LIMP. Each expression is enclosed in parentheses and has as its value a string containing the numeric value of the expression. The allowed operators are +, −, * and /, with their usual meanings. Note that there is no possibility of confusion between "/" used as an end-of-statement mark and "/" used as a division operator because the latter appears only within parentheses. Parentheses may be nested to arbitrary depth, and arithmetic operations may continue over as many lines as necessary. Numbers are stored as strings of digits; negative numbers are signed, positive are not.

4. Examples of LIMP Macros

The simplest possible macro operation is one in which a single line is expanded into several lines with parameter substitution. For example, suppose we wish to write an arithmetic statement in an assembly language program on a single address machine (the notation for this assembly language is that of [1]):

```
* = * + *.
END PU = 'FETCH, ' 20 '
ADD ' 30 '
STORE, ' 10 /
```

This template would match a line such as "SAM = JOE + BILL.", and the code body would punch:

```
FETCH, JOE
ADD, BILL
STORE, SAM
```

The system string "PU" is the punch unit, and whenever it is assigned a value, that value is punched (followed by a carriage return). "PR" is the printer, and behaves in the same way. The intermediate carriage returns were supplied by including them in literals.

The next step in complexity is to allow a macro call to be used in the code body of another macro. For example, we might define a complex addition by:

```
Z* = Z* + Z*.
  IN = 'R' 10 ' = R' 20 ' + R' 30 /
END IN = 'I' 10 ' = I' 20 ' + I' 30 /
```

If the line which was matched was "ZSAM = ZJOE + SBILL.", then the first statement would assign the value "RSAM = RJOE + RBILL." to the system string IN. (Note that the end-of-line symbol is automatically supplied.) The effect of this assignment is to save the current state of all strings and call the processor recursively to expand the line whose value is assigned to IN. After the expansion is completed, control returns to the next statement. If IN is used in a pattern or replacement, rather than being assigned a value, its value is the next effective input line (an effective input line may have been generated by a macro at a higher level, as "RSAM ⋯" in the present example).

In most conventional macro processors, an expanded line is automatically submitted for a re-scan; in LIMP the re-scan is under the control of the programmer. For most applications this causes no difficulty, but there are certain macros in which the programmer may not know whether a line contains a macro call or not. In such instances he would resubmit the questionable line to LIMP by assigning it to IN. If it matched some template, it would be expanded; otherwise it would merely be copied into the punch output stream.

All of the normal conditional assembly operations of most macro generators are available in LIMP by means of suitable statements. We can easily test the value of any argument, make arbitrary changes in any argument, skip statements, loop, etc. The only feature of conventional macro processors which is not obviously available is that of nested definition. This is provided by explicit reference to the template tree by means of the symbol "TT". Statements using TT obey exactly the same rules as those using ST, the symbol tree. Suppose, for example, that we were writing an algebraic language based on single-address assembly code. Suppose further that we demanded that the programmer declare all variables. We could define a macro to inform him of undeclared variables:

```
FETCH, *.
END PR = 'VARIABLE ' 10 ' IS UNDECLARED.'/
```

The addition macro defined above would be altered to expand the FETCH instruction (using IN), rather than just punching it, and this would cause the "Undeclared" message to be printed:

```
* = * + *.
  IN = 'FETCH,   ' 20 /
END PU = 'ADD, ' 30 '
STORE, ' 10 /
```

Now, whenever a variable is declared, we must define a FETCH macro for that variable. The template matching process ensures that the macros containing variable names will take precedence over the one with a parameter flag.

```
VARIABLE *.
  PU = 10 ':  RESERVE, 1' /
  TT = TT 'FETCH, ' 10 /
END TT 'FETCH, ' 10 = 'PU = ''FETCH, ' 10 ''' /
END' /
```

The first statement punches an assembly code operation to reserve one location for the variable and define the symbol. The second statement inserts the new template on TT, while the third provides the definition. Note how the quotes in the definition are supplied—since they are inside a literal, their number must be doubled. The slash on the third line is a part of the literal, not the termination of the rule. The FETCH macro for the variable A would thus have the form:

```
FETCH, A.
  PU = 'FETCH, A' /
END
```

Notice here that the use of FETCH as both a macro name and a target language operation is possible because of the programmer's freedom to submit a line for re-scan or punch it out immediately.

The main reason for requiring that the programmer declare his variables in the preceding example was to ensure that space was reserved. This can be done using the LIMP symbol tree and type 3 conversion without requiring declarations. Instead of defining the fetch macro to print an "Undeclared" message, we write:

```
FETCH, *.
END PU = 'FETCH, V+' 13 /
```

By the definition of type 3 conversion, this macro will punch out "FETCH, V + n" where "n" is the value found in the symbol tree for the variable. If the variable was not in the symbol tree, it would be added and given the current value of the location counter. The location counter would then be incremented by one.

At the end of the program, we would reserve a block of space to hold all variables. A little thought will show that the size of this block is just the value of the location counter at the end of the program. We can therefore define a macro END by:

```
END.
END PU = 'V:  RESERVE. ' 00 '
END' /
```

This macro will replace the programmer's END statement with:

```
V:  RESERVE, k
END
```

This reserves a block of memory of the current length and gives it the name V. The assembler's address arithmetic feature then references each variable within this block.

Suppose, however, that the assembler being used cannot do address arithmetic. LIMP macros can still be written to avoid the need for variable declaration:

```
FETCH, *.
END PU = 'FETCH, V' 13 /
END.
  10 = 00 '.'/
  20 = '0' /
CHK 11 20 '.' /S(END)
  PU = 'V' 20 ':  RESERVE, 1' /
  20 = (20 + '1') /(CHK)
END PU = 'END' /
```

Here each variable is named "Vi" for successive integers *i*. The "END" macro then creates a seris of space reservations, one for each variable. It is interesting to note that there is no penalty attached to specifying a go-to. Because of the way the statements are stored, one with no go-to field is effectively supplied with two go-to's, both to the succeeding statement.

The use of TT allows us to add code to existing macro definitions or change them temporarily. The statement

```
TT <constant pattern> dA = <string> d0 /
```

will add code to an existing macro. We can change a macro temporarily by something like:

```
TT 'TEMPLATE *' dA = <string> /
PT = PT 'SAVEMAC' /
PT 'SAVEMAC' = d0 /
```

The current code body of macro "TEMPLATE *" is saved on the private tree as the value of the symbol "SAVEMAC", and is replaced on the template tree by ⟨string⟩. The code body can be recovered from the private tree by:

```
PT 'SAVEMAC' dA = /
TT 'TEMPLATE *' = d0 /
```

In the current version of LIMP the code body of a macro is not held as a string because of the consequent slowdown in interpreting it. It is converted into a list structure when first defined, and hence it is not possible to use string operations to make changes *within* the code body. If such operations are necessary, the code body must be entered by means of "IN" and stored on the private tree as a string. Changes can then be made, and the code body redefined from this string. The following macro will read a code body and store it in the private tree as a string:

```
ENTER    CODE BODY * UNTIL *.
         PT = PT 10      /
         30 =            /
LINE     40 = IN /
         41 20           /S(END)
         30 = 30 40      /(LINE)
END PT   10 = 30         /
```

A call on this macro might be:

```
ENTER CODE BODY SAM UNTIL SAM IS FINISHED.
<code body>
SAM IS FINISHED.
```

The code body would be read and stored in actual parameter 3 until the line "SAM IS FINISHED" was recognized by the test "41 20 /S(END)". At this point the string from parameter 3 would be stored as the value of "SAM" on the private tree. This string could then be manipulated in any way and defined as the code body of any template.

5. Implementation and Extension

The entire LIMP processor is written in the list language WISP [10, 11], and hence possesses a certain amount of machine as well as language independence (WISP systems are available on the IBM 7094, 7040, GE 265, English Electric KDF9, Elliot 803, EDSAC 2 and Atlas 2). The processor itself is imbedded in an environment which interacts with the operating system to route LIMP output text to the correct compiler. In the Basser system, the compiler to be used is specified by 3 characters of a 12-character program identifier. The user may provide a program title as well as an identifier, and the LIMP environment uses the first 3 characters of this title to replace the

compiler specification. The output text from LIMP is then resubmitted to the system by writing it on a magnetic tape and resetting the input unit. There is no restriction on the compiler selected; it could easily be LIMP again. It is therefore possible to make multiple passes through LIMP before going to a conventional compiler.

The version of LIMP described in this paper is a reformulation of that which is currently in use in the Basser Computing Department. The two versions are equivalent in power, but the Basser version requires that the code bodies be written out in a form much closer to that in which they are executed. The resulting macro definitions are relatively clumsy to write, and difficult to decipher when debugging.

The processor has two main phases: definition and expansion. When a macro is defined, its template is added to the template tree and its code body is converted into a form which can be handled by an interpretive routine during the expansion phase. This conversion recognizes a number of special cases, and sets up different structures for each. For example, if "PU" is the string reference, then the replacement is formed into a single string with markers for the parameter substitutions. Thus the string need not be re-evaluated each time the macro is expanded. Successive punch statements are combined, with suitable insertion of carriage return characters. Similar simplifications can be made where the string reference is "PR" or "IN". Statements referencing "PT", "ST" and "TT" are likewise singled out and when a symbol or template is defined by means of two such statements, these are combined.

During the expansion phase, input lines are read and matched to the template tree. When a macro call is found, the corresponding code body is examined and one of several processors is called to punch, print, move information to the input, etc. One of these processors is an interpreter for the subset of SNOBOL which makes up the general code body statement. If the statement cannot be recognized as a special case, it is handled by this interpreter. Each such statement has been converted into a list with sublists for the pattern and replacement, each of which is a list of elements. The go-to fields are replaced by links to the lists for the correct statement. If no go-to is specified, both S and F links point to the next sequential rule. Thus every rule effectively has go-to fields, and any rule which is not accessible from some other rule is not attached to the list structure. Such inaccessible rules are eliminated during garbage collection.

LIMP enters the definition phase immediately after processing the flag line, and accepts definitions until a blank line is encountered where a template is expected. The expansion phase is entered with the succeeding line. (Notice that this allows for a blank line to appear anywhere within a code body.) The expansion phase continues until one of the three statements "TT = IN /", "PU = IN /" or "PR = IN /" is executed in a code body. The first statement initiates a re-entry to the definition phase. When another blank line is found, the program returns to the statement following "TT = IN /". The two statements "PU = IN /" and "PR = IN /" initiate block copying from the input unit to the punch and printer, respectively. In each case the template-matching process is bypassed. When a blank line is encountered, the processor returns to the statement following the one which initiated the copying.

The major drawback of the current implementation is that WISP stores a single character per word, which increases both the storage requirements and execution times of LIMP. When it is processing macros with many statements which are not special cases, the name "LIMP" seems particularly apt. This defect of WISP is being remedied by the inclusion of packed character strings as data items.

The major extension contemplated for LIMP is an alteration of the template scan. It is proposed that, instead of a single parameter flag, we allow named parameters of the form *NAME*. Such a parameter would be defined by a Backus Normal Form expression, and the scanner would have the power of a syntax-directed compiler: A substring of the input line would only match a named parameter if it had the specified syntax. Each BNF rule would have an associated code body, which would be called by a function EX(d0). The value of this function would be the string produced by the code body of the rule used to recognize parameter *d*. A parameter with no name (i.e., "**") would be handled in the same way as the current version handles all parameters. The extended LIMP would be closely related to the syntax-directed compiler described by Warshall and Shapiro [12].

Extending LIMP in the manner described above would result in a program with considerable power, but which could be used for simple things. One of the drawbacks of a syntax-directed compiler is that one must specify the entire syntax of a language. This would *not* be the case in extended LIMP. One could make small changes in the apparent behavior of an existing compiler by defining small extensions or changes in its syntax in LIMP—the bulk of the syntax analysis would still be done by the existing compiler.

A macro processor which has this ability to make small changes in an existing compiler was recently described by Leavenworth [13]. His program uses syntactic information during the recognition process and allows fixed strings to be interspersed with the formal parameters. However, the macro name must precede the first parameter and must be unique. Apparently recognition of the macro is performed on the basis of the name alone, and syntactic information is used solely to establish the values of the actual parameters. The code body of the macro allows for conditional generation in a rather restricted manner. By including extra information in the template (*following* the name) one macro definition essentially becomes several. The code body then has ways of generating appropriate code for each alternative.

6. Summary

In this paper we have described an approach to language-independent macro processing. The key principle of Limp is string manipulation with parameter substitution, a technique which has recently been used in two similar processors: Strachey's General Purpose Macrogenerator [3] and Mooers' Trac [5]. These systems are very close to Limp in design goals, but their realization displays a fundamental difference in programming philosophy. Both employ the notion of "nested functional expressions" in which the macro is written as a composite function with strings as arguments. Limp treats the macro as a procedure to be executed rather than a function to be evaluated.

Limp has sufficient capacity to provide the programmer with a powerful tool for modifying the apparent behavior of an existing compiler, without the necessity for a complete redefinition of that compiler's language. This program has been available for general use in the Basser Computing Department since April, 1966. Typical applications have been to reduce the amount of punching required in Algol programs, to eliminate the need for many of the line declarations in a flowchart language, and to avoid the individual specification of large numbers of constants in plotting programs written in assembly code.

Our experience with Limp has led us to consider the possibilities of incorporating most of its features into a general text-editing system, thus combining the tasks of text correction and expansion. Research in this area is being carried out in connection with the development of a multicomputer network by the staff of the Basser Computing Department.

Acknowledgment. The general approach taken by Limp is similar to that of most other macro processors—its relation to Befap and Ibmap can easily be seen. Many features of Limp are a direct outgrowth of the author's work on several Wisp compilers, in collaboration with Prof. M. V. Wilkes (Univ. of Cambridge), H. Schorr (IBM), and R. J. Orgass (Yale Univ.). Special thanks are due to Dr. M. H. Rathgeber of the University of Sydney for his aid in providing test cases and suggestions for both the processor and its documentation, and to the two referees who struggled through the first version of this paper and provided an all-important fresh outlook.

Received May, 1966; revised November, 1967

REFERENCES

1. McIlroy, M. D. Macro instruction extensions of compiler languages. *Comm. ACM 3* (April, 1960), 214.
2. Halpern, M. I. XPOP: a metalanguage without metaphysics. Proc. AFIPS 1964 Fall Joint Comput. Conf., Vol. 26, p. 57.
3. Strachey, C. A general purpose macrogenerator. *Comput. J. 8* (Oct. 1965), 225.
4. Graham, M. L., and Ingerman, P. Z. An assembly language for reprogramming. *Comm. ACM 8* (Dec. 1965), 782.
5. Mooers, C. N. TRAC, a procedure-describing language for the reactive typewriter. *Comm. ACM 9* (March, 1966), 215.
6. Farber, D. J., Griswold, R. E., and Polonsky, I. P. SNOBOL, a string manipulation language. *J. ACM 11* (Jan. 1964), 21.
7. ——. The SNOBOL3 programming language. *BSTJ 65* (July–Aug. 1966), p. 895.
8. Waite, W. M. A language-independent macro processor. Basser Computing Dept. Tech. Report 41, Sydney, Australia, March, 1966.
9. Griswold, R. E., and Polonsky, I. P. String pattern matching in the programming language SNOBOL. Bell Laboratories, July 1, 1963.
10. Wilkes, M. V. An experiment with a self-compiling compiler for a simple list processing language. In Richard Goodman (Ed.), *Annual Review in Automatic Programming, Vol. 4*, Pergamon Press, New York, 1964, p. 1.
11. Orgass, R. J., Schorr, H., Waite, W. M., and Wilkes, M. V. WISP—a self-compiling list processing language. Basser Computing Dept. Tech. Report 36, Sydney, Australia, Oct. 1965.
12. Warshall, S., and Shapiro, R. M. A general-purpose table-driven compiler. Proc. AFIPS 1964 Spring Joint Comput. Conf., Vol. 25, p. 59.
13. Leavenworth, B. M. Syntax macros and extended translation. *Comm. ACM 9* (Nov. 1966), 790.

Proposed USA Standard

COBOL

is now available

- This important Proposed USA Standard COBOL is contained in an issue of the ACM SICPLAN Notices (Volume 2, Number 4, April 1967), which was distributed in June to the regular ACM SICPLAN mailing list. It has also been sent to the COBOL Information Bulletin mailing list.
- To interested persons not on either of the mailing lists for the above publications, this Proposed COBOL Standard is available at $3.00 per copy. Orders must be prepaid and should be addressed: COBOL, Association for Computing Machinery, 211 East 43rd Street, New York, New York 10017. A special price of $2.50 per copy is being granted *by ACM* for bulk orders of 50 or more.
- USA Standards Committee X3 has authorized publication of this document, which contains 538 pages, to elicit comment and criticism from the data processing community prior to voting on its acceptance as a USA Standard. Comments should be addressed to: X3 Secretary, Business Equipment Manufacturers Association, 235 East 42nd Street, New York, New York 10017.