SYMPLE—A general syntax directed macro preprocessor

by JAMES E. VANDER MEY
The Pennsylvania State University
University Park, Pennsylvania

ROBERT C. VARNEY
The Pennsylvania State University
McKeesport, Pennsylvania

and

ROBERT E. PATCHEN
IBM Corporation
Boston, Massachusetts

INTRODUCTION

The subject of this paper is a general syntax directed macro preprocessor system. One of the suggested potential uses of this system is that of evaluating new or extended programming languages by the technique of syntax directed macros. This led to the association of the acronym SYMPLE (SYntax Macro Preprocessor for Language Evaluations) with this system.

A preprocessor is a processor intended to be used prior to another processing stage. In our case, it is assumed that the SYMPLE preprocessor system will generally be used in processing higher level language texts (ones which are user oriented), producing output text in the same or a similar higher level language.

The term “macro” is used in a very general sense in this paper. As in other macro systems, the macro mechanism consists of the recognition of a macro “reference” in the source text being processed, and a macro “definition” defining a translation procedure invoked by some corresponding macro reference.

A SYMPLE macro definition consists of two parts: the “macro semantic portion” or “macro body”; and the “macro templates.”

The macro semantic portion is the translation procedure and consists of the instructions to be executed when the macro is “invoked”. A macro is invoked when a pattern described in one of its macro templates is recognized by the parser in the source input text. This macro reference pattern may have identifiable parts which are then considered as arguments for the semantic portion.

A macro template defines a possible macro reference pattern for this macro and consists of two distinct parts: A specification of a general syntactic substructure of the source input text in which a given macro reference may occur (i.e., context); and any necessary further syntactic qualifications within that general syntactic substructure (e.g., a specific pattern). The actual pattern matching technique for macro reference is thus a two level syntax directed matching procedure. This syntax
directed macro reference technique is the method by which SYMPLE achieves both simplicity and generality.

The SYMPLE system as a macro system is not tied to any particular programming language. The base (source input) language and the object (output) language of the macro facility could in fact be entirely different languages.

The syntax of the languages to be processed and/or extended must be adequately described through the syntax description metalanguage of the SYMPLE system. This syntaxic description is used for determining "context" for macro references and thus the requirements for a minimally "adequate" syntaxic description of a language are proportional to the degree of context required to isolate macro references.

As a very simple example, assume all macro references must occur in only a single specific syntactic unit (syntactic substructure) of the base language (e.g., only labels of Fortran statements). Then to facilitate the recognition of macro references in the source language, the syntax of the base language need only be described via the metalanguage to the extent that it can isolate this syntactic unit type (i.e., Fortran labels.) When recognized, this syntactic unit will then be considered as a candidate for containing a macro reference.

After a candidate syntactic unit is isolated in the source input a check can be made for the existence of specific macro references by testing for further qualifying patterns within that syntactic unit. For instance, a Fortran label of "three blanks followed by two numbers" might be a specific macro reference. A check would thus be made for this reference according to the syntactic pattern defining "three blanks followed by two numbers" whenever a Fortran label is recognized. This process of local syntax investigation is called "template matching" for a macro reference.

It is also through the template matching facility that translation parameters in the source language (e.g., arguments, conditions, etc.) are recognized and passed to the actual macro facility. These translation parameters, which we shall call argument strings, can be manipulated by the instructions contained in the body of the macro (semantic portion).

Since the primary function of the SYMPLE system is that of a preprocessor, the translation process is mainly that of a manipulation of argument strings and the insertion of modified and/or created strings back into the source input. Hence, the actual semantic portion of the macro is implemented in a language oriented to the manipulation of character strings. Thus translation due to macro references and related translation parameters generally results in the insertion of the translation code in the base language into the body of the code being processed. It will be shown that this "in place" translation in the SYMPLE system does not necessarily imply expansion in exactly the same place (i.e., at the lexicographical location of the macro reference).

An attempt will now be made to summarize and interrelate the functions of the SYMPLE system by outlining the system functional flow via a system flow diagram (Figure 1) and the following brief description.

The preprocessor operates as follows:

1. The first items processed contain control information which includes such items as the device(s) from which subsequent information is to be read, the device(s) designed for system output, the names of special edit macros, specific listing options, etc. Control information may occur in the input stream at other logical stages of processing.
2. A description of the base language syntactic structure is read as input and processed to build a data base for the recognition portion. This data base will be used later by a parser.
3. Macros (templates and associated semantic translation routines) are read in, stored, and used to create necessary data bases for later processing.
4. A source deck is read in and parsing of the source input begins. (Probable entry point for most users.)
   a. As a syntactic unit is recognized, a check is made to see if any macros have templates to be matched in this syntactic unit.
Templates of edit macros, if any, are tested last. When there are no templates left to be checked and if the end of the total parse has not been encountered, the parse is continued.

b. If a macro template match is successful, the argument strings are passed to its associated macro semantic portion. There may be any number of macro templates associated with a given macro semantic portion, and identical template patterns can be associated with different macro semantic portions.

c. The instructions in the current macro semantic portion are executed (actually interpreted) and the results of their operations are effected (e.g., storage manipulation, insertion of translation into input source, dynamic creation of new macro templates or semantics for this or other macros). Upon completion of execution control is returned to 4a above.

5. When the source deck has been completely parsed and thus source time translations, including any necessary editing, have been completed, the file is then ready for output in a manner specified by the control information.

6. Processing is now completed, but by appropriate control information another cycle may be initiated on (a) new information or (b) on a previous preprocessor output file. Thus, in the latter case, we have the possibility of a multi-pass preprocessor, if desired.

The remainder of this paper will be devoted in the main to the details of what the SYMPLE system can do and in general how one goes about using the SYMPLE system. The syntax description metalanguage is introduced first followed by an introduction to the macro translation (semantic) and insertion capabilities of SYMPLE.

Syntax description metalanguage

The syntax description metalanguage is used to describe a parsing “grammar” of the base language in which macro references are to be embedded and thereby outline the manner in which the source input is to be parsed. For example, suppose a label field is one syntactic structure to be parsed. The parser should then be told that a label field consists of, say, five characters which are either all digits, all blanks, or a string of blanks followed by a string of digits.

The grammatical metalanguage used to direct SYMPLE's parser is similar to the Backus-Naur Form (BNF) metalanguage. For example, similar grammatical productions are used to define syntactic structures; the nonterminals and terminals of BNF are also used being renamed syntactic units and literal strings, respectively. There are, however, several features in SYMPLE's metalanguage which were incorporated to extend the power and simplicity of grammatical description over that of standard BNF.

Actual productions in SYMPLE's metalanguage to define the parsing desired in the preceding example are:

\[(LABEL-FIELD):5&5(08'08'(DIGIT))\]

\[(DIGIT):'0'1'1'2'3'4'5'6'7'8'9'\]

The first production above is interpreted as: a label field is defined as not less than five nor more than five characters of a string of zero or more blanks immediately followed by zero or more digits.

Productions

The syntactic units of the base language are defined by productions in the metalanguage. These productions are of the form:

\[(LHS): right side\]

where (LHS) represents the syntactic unit being defined on the left side and the right side contains metalinguistic descriptions of other syntactic unit(s) and/or literal string(s) in the left to right order in which they comprise the structure of (LHS). The colon (:) separates the defined syntactic unit on the left side from the defining information on the right side.

The first production of the base language grammar must be the definition of the syntactic unit representing the total syntactic structure of the base language (i.e., the initial or distinguished symbol of BNF). Other productions may be in any order.

(Named) Syntactic units

The metalinguistic representation of a syntactic unit in a production is a string of arbitrary length enclosed in parentheses. The string (called the name of the syntactic unit) may be composed of any characters with the exception of those used as special delimiters in the syntax description metalanguage (i.e., illegal characters are (;;|$:&).
Literal strings

A literal string is represented in the metalanguage by the desired string of characters enclosed in single quotation marks ('). Any character may be used within a literal string, except that a single quotation mark is represented by two adjacent single quotes for each occurrence in the literal string in order to differentiate it from the ending delimiter of the literal string.

Alternatives

If a syntactic unit in the base language may have alternative representations, these alternatives may be represented in the metalanguage as a single production with the alternatives of the syntactic unit each appearing on the right side and separated from each other by the conventional OR symbol (|).

Example: \((\text{DIGIT}):'1'\ | '2'\ | '3'\)(OTHER)

Complex substructures (Unnamed syntactic units)

If one does not wish to break down and label a syntax substructure in detail, but simply label an entire complex substructure as a syntactic unit, pairs of parentheses may be used as grouping indicators. Consider the following equivalent examples of a definition of the syntactic unit \((\text{NUM})\).

Example: \((\text{NUM}):'2'|'3'|'4'\)
\((\text{NUM}2):'3'|'4'|'5'\)
\((\text{NUM}3):'6'|'7'\)
\((\text{NUM}4):'1'\ (\text{NUM})\ (\text{NUM}2)\ |'1'\ (\text{NUM}3)\)

Example: \((\text{NUM}4):'1'\ (('2'|'3'|'4')\ ('3'|'4'|'5'))\ (5'|6'|7'))

Grouping may occur to any depth desired and each quantity within the grouping parentheses must have the form of any legal right side of a production.

Quantity repetition and bounds

Often in the syntax of a base language a (named or unnamed) syntactic unit or literal string may be required to occur several times. Or it may be desirable to specify that a syntactic structure be a function of the length of an input string in addition to other qualifications (e.g., a label field of exactly five characters and consisting of . . . ).

To indicate either the repetition of a string (i.e., the input string defined by a syntactic structure) or the length bound on the number of characters in some string, an operator group must precede the respective quantity in the syntax. The operator group is of the form \(n^m\) or \(n^m\) for the string and character counters respectively, where \(n\) is an integer representing the lower bound and \(m\), an integer representing the upper bound.

Consider the following example.

\((A):\ 3^3\ (\text{SUB-STRUCTURE})\)
\((B):\ 3^3\ (\text{SUB-STRUCTURE})\)
\((C):\ 'C'\)
\((\text{SUB-STRUCTURE}):\ 0^5\ (C)\ 1^3\ 'AB'\)

The first production defines \((A)\) as exactly three strings of \((0^5(C)1^3\ 'AB')\). Thus, acceptable strings for \((A)\) might be ABABAB or ABCABCABCABCABAB or CCA-ABABCABAB, etc. However, \((B)\) is defined as exactly three characters which are otherwise defined as in \((A)\). Thus, \((B)\) can be only CAB; no other combinations will yield exactly three characters. Notice that the string counter differs from the character counter in that it is distributed over all inner strings whereas the character counter represents an absolute bound over a given substructure.

When productions include quantities with repetition counts, the parser which utilizes these productions will attempt to find the largest number of those quantities in the input source consistent with the upper bound of repetitions. If the input contains more than the upper bound of these quantities, the input string corresponding to the upper bound count of quantities will be recognized and succeeding repetitions will be analyzed according to the syntax following. A lower bound count of zero is allowable and simply indicates the optional omission of the quantity.

The absence of an explicit lower bound implies a lower bound of one. The absence of an explicit upper bound implies an upper bound which is the maximum bound allowable in the system. In the present implementation it is \(32767\). It should be noted that

\(1^8\ (\text{SYUN})\) and \((\text{SYUN})\) are equivalent as are \$\ (\text{SYUN})\) and \(1^8\ 32767\ (\text{SYUN})\)

Complement look-ahead

The symbol \(\neg\) preceding a literal string, syntactic unit or grouping indicates that at that point in the syntax the quantity indicated must not occur. This is called a complement look-ahead for the indicated quantity at
parse time. If the quantity is found, the parse being attempted has failed. (Any syntactic units found on the look-ahead will not result in macro template match attempts.) If the quantity is not found, the parse continues as before the complement look-ahead.

(SPLTRSTRG): $( \neg \text{C}(\text{LETTER}))$

The strings recognized as (SPLTRSTRG) will be any string which consists of one or more of A, B, D or E, but not C.

Scan positioning

The production defining a syntactic unit can be made to include, without investigation as to structure, an arbitrary length of input, or it may require that a particular syntactic unit in the input conform to more than one syntactic structure. This is done by explicitly positioning the location at which the parser is “looking.” This location, called the scan position, can be adjusted either relative to its present position or to the beginning reference points in the syntax of the parsed input.

a—X(Space) positioning

The occurrence of the symbol X immediately followed by an unsigned integer number and delimited by bracketing commas at any point in the right side of a production will cause the scan position to be adjusted rightward from its present location the integer number of positions specified. The symbol X and following number must be bracketed on both sides by commas except in the following cases: X is the first (last) symbol of a grouping level or the first (last) symbol of the right side of a production, in which case the left (right) comma is not required.

Example: Define an (END-CARD) to be an 80 character string. The first six characters must be blanks, the next 66 characters must have the word END somewhere with the rest blanks, and the last eight characters may be anything.

(END-CARD): 6 & 6' '66 & 66 (0$ ' ' (END'))

b—T (Tab) positioning

The format is similar to that of X positioning, except a T is used instead of an X.

The T scan positioning results in the scan position being moved the specified number of places to the right of the beginning location at which the parse began at (1) this grouping level, if the T positioning appears within a grouping parenthesis pair, or (2) the right side of the production otherwise.

Example: A syntactic unit (EMPLOYEE-NO.) is defined to be an 80 character string with a syntactic unit (LAST-NAME) beginning in position one, followed by a single blank and then the syntactic unit (FIRST-NAME). Exactly 15 spaces after the beginning of (FIRST-NAME) is to appear the syntactic unit (CODE). Finally (NUMBER) will be 75 spaces from the beginning of (EMPLOYEE-NO.).

EMPLOYEE-NO.: (LAST-NAME) ' '
(FIRST-NAME), T15, (CODE), T75, (NUMBER)

Recursive grammars in the metalanguage

Recursive grammars (i.e., productions with the syntactic unit of the left side occurring as well on the right side, or being in the derivation of a syntactic unit of the right side) are allowed in the metalanguage subject to certain conditions.

For instance, left recursive productions are not allowable, but other recursive productions are allowable.

Further, the character (&) bound counts are cumulative from the initial (top) occurrence in a recursive parse while the repetition bounds ($) are effective at each level of recursion.

Non-specific grammars in the metalanguage

Let a non-specific grammar be one in which the particular alternatives of structure for a syntactic unit may have structurally the same headings (i.e., leading components which are structurally the same). The metalanguage allows the specification of such grammars and at recognition time the parser always picks the first specified (or left most) alternative as its initial guess. Subsequent guesses continue with the next specified alternatives.

The user must be aware of the possible consequences if the apparent ambiguity in a non-specific grammar causes the recognition of syntactic units to be rejected later as a result of an unsuccessful parse. Though the back-up to the next alternative is handled automatically by the parser, the syntactic units recognized may result in macro invocations; the results of which will not automatically be negated. Relevant user aids in this area are provided by the system.

The following example illustrates a parsing grammar
for a language which is context sensitive and not context free and which utilizes recursive productions.

\[ L = \{0^n1^n : n \geq 1\} \]

\((\text{LANG}) : (\text{LSTR}) \rightarrow '1', T1, $'0'(\text{RSTR})\)

\((\text{LSTR}) : '0'(\text{LSTR})'1'\ '01'\)

\((\text{RSTR}) : '1'(\text{RSTR})'0'\ '10'\)

The parser first determines that the input string belongs to the context-free language \(0^n1^n\); checks to make sure \(x\) does not begin with a 1; repositions to the beginning of the parsed substring of 1's and then determines that the remaining substring of the input string belongs to the context-free language \(1^n0^n\). If the above conditions are true, then the input string belongs to the context-sensitive language \(0^n1^n0^n\).

The SYMPLE macro facility

The macro facility of SYMPLE provides the actual translation mechanisms. The macros themselves are read in to the system following the base language grammar and prior to the user’s source deck. The individual macro definitions are described in this section.

MACRO FORMAT

The overall format of an individual macro definitions is as follows:

\[
<\text{macro name}> \ (\text{<syntactic unit> = <template body> / (\text{<syntactic unit> = <template body> . . . ; macro semantic statements END};\)}
\]

The exact format and meaning of the various parts are described in the balance of this section.

Macro name

The first item to appear in the macro is the name of the macro. The name may be any string of characters, excluding those special characters previously mentioned as excluded from a syntactic unit name. The macro name is used exclusively as a "handle" for the user's organization and SYMPLE's internal system and macro referencing. The macro name should not be confused with a macro reference in the source text. A source reference to the macro is completely independent of its name.

Templates

Following the macro name are a series of macro templates which are descriptions of possible macro references that will cause the invocation of the macro. A single macro template is of the form:

\[
(\text{<syntactic unit>}) = \text{<template body>}
\]

where the syntactic unit is any syntactic unit that may occur in the base language, and the template body, if present, consists of a description of a specific structure to be found within that syntactic unit. The syntax and semantics of template body are identical with those of the metalinguage of SYMPLE except for an extension to make it possible to identify and name argument strings for the macro.

The extension added to facilitate the identification and naming of argument strings was simply to allow the enclosing of the desired argument location in the syntactic structure of the template within bracketing parentheses and preceding the left enclosing parenthesis with a name (with the same character restrictions as a macro name) to be associated with the enclosed argument string. These enclosed argument strings may occur anywhere within the template, and in fact may even enclose other argument strings. The names associated with the argument strings must be unique within a single macro template.

A macro template may cause a macro invocation in the following manner. When the syntactic unit designated on the left of the equal sign in a macro template is recognized by the parser, the actual structure of the syntactic unit found is compared with the specific syntax specified in the template body. A successful comparison results in the invocation of the macro and the passing to the macro of identified argument strings in the macro reference, if any. If no template body is specified, then the macro is immediately invoked with no arguments passed.

The syntax structure defined in a template body need not be structurally consistent with that of the object syntactic unit in which it will be compared. However, if the template body contains syntactic units, these units must have been in the productions submitted with the description of the base language. These productions though can be stand-alone productions (not logically in the normal base language structure) included solely for use within templates. The use of these stand-alone syntactic units, literal strings, and alternative arrangements and selection of syntactic units in the base language can result in template structures quite different from those recognized in the process of finding the object syntactic unit. Thus the template comparison is actually an attempted reparsing within
the physical bounds of the object syntactic unit according to the template syntax description.

Any number of macro templates may follow the macro name, with a slash (/) separating each, except that the last template is followed by a semicolon (;).

Example: \texttt{NO1 (LABEL) = A1 ('A2 \langle NUM\rangle)/(STMT) = 'C' A3 (X79); macro semantic statements END;}

Macro NO1 will be invoked when either

1. A \texttt{(LABEL)} is found consisting of four blanks followed by a \texttt{(NUM)}, or else
2. A \texttt{(STMT)} is found beginning with the letter \texttt{e}.

In case 1 two argument strings will be available for manipulation and testing by the macro semantic statements; that associated with the string name \texttt{A1} will be four blanks and the found \texttt{(NUM)}; that associated with the string name \texttt{A2} will be just the found \texttt{(NUM)}. In case 2, the argument string associated with string name \texttt{A3} will be the 79 characters following the initial letter \texttt{e}.

Argument string names which are not in a matched template or which are associated with null argument strings in the matched template are associated with the null string (i.e., have a length attribute of zero).

**Macro semantics**

\textbf{a—General}

The macro semantics facility in SYMPLE is based on a string oriented language which drives an interpretive mechanism. This language closely parallels SNOBOL and has a simple syntax. The basic form of most semantic statements is

\[
<\text{action verb}>, <\text{string name}> = <\text{string reference}> , <\text{string reference}> . . . ;
\]

where the action verb is a key word describing some action to be performed on the referenced strings (literal strings, string names, etc.) with the resultant string generally being associated with the given string name. The details of the semantic language facility are described in another paper.\textsuperscript{18} The use of relatively simple semantic statements in later examples should be intuitively understandable.

This semantic language provides the ability to:

1. manipulate strings of characters
2. reference strings literally, directly, indirectly
3. reference strings with concise notations
4. communicate between macros
5. execute subroutine-like macros
6. manipulate strings of values
7. alter sequential execution (branch)
8. insert strings back into the ground language code
9. loop repetitively
10. perform string comparisons
11. display string-string name associations
12. terminate interpretive action

The last capability mentioned and number 8 listed above are the means by which the macros effect their results in the translation process.

**b—Output string insertion**

Strings which are produced in the macro semantic portion of a macro may be inserted into the source code in any of several ways. The semantic language statement which directs the insertion of a string is of the form:

\[
\text{INSERT, <directives> = <string name(s)> ;}
\]

The directive is a code rather than a string name which specifies the type of insertion to be performed. The directive codes are I, IA, IB, A, B, A, <digit>, B, <digit>, PI, PIA, PIB, PA, PB, PA, <digit>, PB <digit> and MADD.

They are explained below.

I—The string name(s) is an argument string name(s). The associated string is to replace the argument string occurrence in the macro reference.

IA—The string name(s) is an argument string name(s). The associated string is to be inserted immediately after the referenced argument string in the macro reference. In this, and for all remaining insertion directives, the macro reference itself remains unchanged.

IB—Same as IA except read “before” instead of “after”.

From the collection of the Computer History Museum (www.computerhistory.org)
A—The string(s) associated with the string name(s) is to be inserted immediately after the syntactic unit in which the current macro reference occurred.

B—Same as A except read “before” instead of “after.”

A, <digit>—The string(s) associated with the string name(s) is to be inserted after a particular syntactic unit or grouping level of the parsed tree, called the referenced syntactic unit (RFSYUN). The RFSYUN is the first syntactic unit (at the same or higher level) to the immediate left of the syntactic unit or grouping level on the parsed tree, whose derivation includes, and is the value of <digit> levels above, the present macro reference. If a RFSYUN does not exist by the above definition then the directive A, <digit> references the beginning of the input stream.

B, <digit>—Same as A, digit except read “before” instead of “after”.

P prefix directives—(e.g., PI, PIA, etc.) Each P prefix directive results in the same type of insertion as the non-prefixed directives. However, the string inserted is transparent to all future attempts at parsing or template matching (i.e., “protected”). The only exception to this is that a P prefix inserted string will be visible to the template matching of a specially designated macro, called the “edit” macro, whose name is specified at submission time via the processor control language. All P prefix inserted strings, if unaltered by the edit macro, will appear in their inserted locations in the final output.

c—Dynamic macro modification

In addition to inserting strings in the source submission, strings may be treated as new/changed macros via the following directive.

MADD—The string associated with the string name is a macro and includes macro templates and/or macro semantics. If the macro is new (no other macro with the same name) it will be added to the present library of macros for this submission. If the macro name is that of a current macro, macro templates, if present will be added to those presently associated with the macro and macro semantics, if present, will replace those of the present macro.

CONCLUSION

The purpose of the SYMPE system is to provide a general language-independent macro preprocessor. The syntax directed approach was used to allow both general and flexible macro referencing techniques.

The SYMPE syntax description metalanguage was designed from the premise that the metalanguage should be a practical tool for real programming languages with their many syntactic idiosyncrasies (e.g., imbedded blanks, fields of specified length, continuation columns, etc.). As far as possible and practical these real problems should be easily describable in the SYMPE syntax description metalanguage. In a standard BNF metalanguage, such problems are at best very awkward to describe. This led to such concepts as length and repetition binding, and explicit scan positioning.

Explicit scan positioning added the ability to perform successive analyses, even within a local template match, by repositioning the analyzer for rescan of already parsed information. This rescanned information may of course, contain different information as the result of insertions from macro invocations.

The insertion of information in the “protected” mode (P-prefix directive insertions) further extends the power of the scan and rescan mechanism of the syntax analyzer. It allows the user the option to insert code which either may possibly affect the future syntax analysis (normal mode), or be completely “transparent” and thus not possibly affect subsequent syntactic analyses.

Systems such as TMG, C3 COGENT14 and similar syntax directed compilers or compiler-compilers have their semantic actions hooked to the parsed syntactic units of a source submission, much the way SYMPE would do without the local syntax parsing of a macro template. In the context of macro processors, however, the application of global syntactic analysis followed by local syntax analysis for the macro references ap-
pears to be a new application. The obvious advantage of this technique is that it provides a means of specifying a contextual dependence for macro references. Patterns in the source input which would qualify as macro references on a local syntax basis will qualify only if they are in the correct global context.

Several previous macro systems [notably XPOP, ML/I, LIMP] use some sort of a generalized macro reference technique. Most used a template matching technique based on pseudo-syntax methods (e.g., noise word structuring of XPOP, specific literal template structures of LIMP). In each case, however, the scope of applicability of these macro references was not controlled on a global syntactic basis. ML/I, for instance, depends on the occurrence of a name of a macro in a statement for the recognition of a macro reference. XPOP looks for a macro reference in each statement based on word structures, with non-“noise” words in these structures being the arguments. Macro references in LIMP are perhaps the most general of the above mentioned systems. However, the templates of LIMP are (1) literal templates (i.e., character strings—not defined syntax structures) with “holes” in them, the “holes” being filled by the required arguments; and (2) each template is eligible in any given “line” of input. Thus there is no discrimination in regard to the applicability of a template on a global basis in any of the above mentioned systems; nor is there structuring of the templates themselves on a general syntactic basis; nor can the arguments be identified in a really general manner. It would take little to show that, at least from a macro reference point of view, these systems would be relatively simple special instances in SYMPLE.

The general applicability of the SYMPLE system has been alluded to, and a few mostly simple examples are illustrated in the appendix. These examples illustrate the use of SYMPLE as a language extension facility, in handling “sifting” problems, and text editing. There are certain to be many other areas of applicability not mentioned.

APPENDIX I

SYMPLE processing examples

Example 1

The first example of this appendix is designed to take OS/360 Fortran IV input and condense all non-comment statements into single condensed strings by eliminating unnecessary blanks, sequence number fields, and continuation fields. Each condensed statement will be separated by a record mark (!). Processor control information is included for completeness.

SYNTAX;

(PROG):$(STMT) (END-CARD)
(STMT) : (LABEL-FIELD) ('0' | ' ')
              (UNLAB-STMT) | (COMMENT)
(COMMENT) : 'C', 'TSI', '
(LABEL,FIELD) : 5 & 5((BLKSTRG) (NUM))
(UNLAB-STMT) : ¬(END-BODY) (BLKSCN)
             (SEQFIELD) 0819 ((CONT-FIELD)
             (BLKSON) (SEQFIELD))
(END-CARD) : 066 ' ' (END-BODY) (SEQ-
FIELD)
(END-BODY) : 66 & 66 (0$1 (BLKSTRG)
                   'END' $81 (BLKSTRG))
(BLKSCN) : 66 & 66 (0$1 (BLKSTRG) $ (NONBLK))
(BLKSTRG) : $'
(NONBLK) : ¬(' | "", X1 | "", X1)""
(NUM) : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
(SEQFIELD) : 'X7', '!
(CONT-FIELD) : ¬('C', 'T6', ¬('0' | '')) X1
SYNEND;
MACROS;
CONDENSE (BLKSTRG) = A1 ((BLKSTRG))/
             (SEQFIELD) = A1 ((SEQFIELD))/
             (CONT-FIELD) = A1 ((CONT-FIELD));
REPLACE, A1 = ;
INSERT, I = A1; STOP; END;
END-STMT (UNLAB-STMT) = ; INSERT,
             A='!'; STOP; END;
MACEND;
SOURCE, REMARK;

Output from SYMPLE after processing above input

INTEGER*2AA(4)/'A B C D'/, BLK/!/,
VAL(100) = VAL(K) + 85 * K
AA(K/100) = BLK! 1000 STOP! END!

Notes on example:

1. The grammar of Fortran IV is detailed here...
only to a level which will distinguish major substructures. If one wished to further detail the syntax structure, the syntax of Fortran statements in the condensed form would be relatively simply since all extraneous clutter has been removed. The P-prefix insert capability could be used to ignore clutter for possible reparsing without actually removing it from the input (and thus output).

2. The grammar is non-specific with at least one point of apparent ambiguity. The beginning characters of an \(\text{(END-CARD)}\) will qualify as the beginning characters of a \(\text{STMT}\) (i.e., \(66\#^*\) \(= (\text{LABEL-FIELD})^*\)). Thus upon encountering an \(\text{(END-CARD)}\) there will be a back-up, since an attempt is first made to parse it as a \(\text{STMT}\). In this case, of course, the back-up will not have any bearing on the total processing result.

3. \(\text{(LABEL-FIELD)}\) will accept a label, say \(\text{bb}1\text{b5}\) and the compressed result would be \(15\). The structure of this particular \(\text{(LABEL-FIELD)}\) would be

\[
\text{bb} \quad 1 \quad \text{b} \quad 5
\]

\(\text{(BLKSTRG)} \quad \text{(NUM)} \quad \text{(BLKSTRG)} \quad \text{(NUM)}\).

4. Note how \(\text{(NONBLK)}\) includes all non-blank characters and literal strings (including blanks).

5. The macro template \(\text{(UNLAB-STMT)}\) \(=\) in the second macro results in the macro \(\text{END-STMT}\) being invoked with no arguments.

6. The use of the processor control \(\text{RECMK}\) parameter results in a ! being added to the end of each logical record on input. The syntax grammar used assumes this, though an equivalent grammar without the \(\text{RECMK}\) could easily be used in this case.

Example 2

This example is designed to remove all redundant parentheses in a language which uses pairs of left and right parentheses for grouping. A redundant parentheses pair is any pair of parentheses which enclose a string which is also totally enclosed in parentheses.

\[
\text{SYNTAX;}
\text{SYNEND;}
\]

\[
\text{MACROS;}
\]

\[
\text{REDUN(PAREN)} = '\text{('}\text{AA(}\text{(INARDS)})')\text{'};
\text{SEPAR,T,AA} = '\text{'}, \text{AA, '}/\text{F, L1; INSERT, I} = \text{AA; L1:STOP; END;}
\text{MACEND;}
\text{SOURCE, LIST;}
\text{(((A(B)))C)(((XYZ)((Q)(A))(F)))}
\]

Output from SYMPLE after processing

\[
\text{((A(B)))C((XZ((Q)(A))))F)}
\]

Note: In a recursive parse, inner-most (lowest) recursive syntactic units \(\text{[e.g., (PAREN)]}\) are recognized first and subject to macro expansion first.

Example 3

A final example shows a simple extension of OS/360 Fortran IV obtained by adding a different statement type to the grammar. This different statement type will contain a macro reference. The format of, and argument location in, the macro references will be strictly dependent on the local syntax specified in the templates of the macros.

A different statement type could be designated simply as starting with a non-numeric non-blank character after column 1 and before column 6. The grammar defining this basic extension could appear in a submission as follows.

\[
\text{SYNTAX,PUT;}
\text{(PROG)} : \text{*}(\text{STMT})(\text{END-CARD})
\text{(STMT):(NEW-STMT)} \quad \text{END-CARD}, \text{T80}
\text{(NEW-STMT):} 666 \text{;} 666 \quad \text{T80}
\text{(END-CARD)} : 666 \quad 666 \quad \text{;} 08 \quad \text{;} \text{'END'}
\text{; 08} \quad \text{;}, \quad \text{X8}
\text{(NUM):} 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9
\text{(NONNUM-BLK):} \quad \text{;}(\text{NUM}) \quad \text{;}, \quad \text{X1}
\text{SYNEND;}
\]

At this point the syntax description differentiating this new statement type is defined and any user could take advantage of the description via the processor control \(\text{PUT}\) parameter has been saved. Using the appropriate processor control and job control statements to retrieve the above syntactic specification, a user could make submissions similar to the following.

\[
\text{SYNTAX,GET;}
\text{(NOISE)} : 80 \quad |'\text{STORE}'| \quad \text{IN} |'\text{TO}'| \quad \text{INTO} |'\text{THE}'
\quad \text{; 'PUT'|'OF'|'AND'}
\]
SYMPLE

$ (\neg (\text{NOISE}), X1) $
SYNEND;
MACROS;
SUM (NEW-STMT) = A1 (\text{NOISE}) (\text{ADD'})
\text{SUM'}) A2 (\text{NON-NOISE})) \text{SUM A1} A3 (\text{NON-NOISE}) \text{SUM A2 A3} A4 (\text{NON-NOISE}), T80);
CONCAT, A1 = ' ', A4, ' = ', A2, '+', A3;
INSERT, I = A1;
STOP; END
MACEND;
SOURCE;
C THIS IS A FORTRAN COMMENT
ADD A TO B AND STORE IN C
SUM A AND B AND PUT INTO C
STORE THE SUM OF A AND B IN C
END

The macro used above is a simple macro using a keyword and non-noise positional parameters. The illustrated new type of statement if imbedded in any Fortran source deck, would, when processed, be converted to the Fortran type statements listed, and replace the new statements.

REFERENCES
1 M J BAILEY M P BARNETT P B BURLESON
Symbol manipulation in Fortran - SASP I subroutines
CACM Vol 7 No 6 June 1964 339-346
2 P J BROWN
The ML/I macro preprocessor
CACM Vol 10 No 10 Oct 1967 618-623
3 J A FELDMAN
A formal semantics for computer languages and its application in a compiler-compiler
CACM Vol 9 No 1 Jan 1966 3-9
4 J A FELDMAN D GRIES
Translator writing systems
CACM Vol 11 No 2 Feb 1968 77-113
5 D E FERGUSON
Evolution of the meta-assembly program
CACM Vol 9 No 3 March 1966 190-196
6 M L GRAHAM P Z INGERMAN
A universal assembly mapping language
Proc ACM Aug 1965 409-421
7 M I HALPERN
XPOP: A metalanguage without metaphysics
Proc FJCC Vol 26 1964 57-68
8 M I HALPERN
Toward a general processor for programming languages
CACM Vol 11 No 1 Jan 1968 15-26
9 B M LEAVENWORTH
Syntax macros and extended translation
CACM Vol 9 No 11 Nov 1966 790-793
10 R M McCLURE
TMG-A syntax directed compiler
Proc ACM Aug 1965 262-274
11 M D MCELROY
Macro instruction extensions of compiler languages
CACM Vol 3 No 4 April 1960 214-220
12 C N MOOERS L P DEUTSCH
TRAC-A text handling language
Proc ACM Aug 1965 229-246
13 R E PATCHEN
String oriented macro language and interpreter
Penn State Univ Dec 1968 Thesis in Computer Science
14 J E REYNOLDS
An introduction to the COGENT programming system
Proc ACM Aug 1965 422-437
15 S ROSEN
A compiler building system developed by Brooker and Morris
CACM Vol 7 No 7 July 1964 403-414
16 E F STORM
CHAMP - Character manipulation procedures
CACM Vol 11 No 8 Aug 1968 561-566
17 J E VANDER MEY
A general syntax directed macro preprocessor
Penn State Univ March 1969 Thesis in Computer Science
18 R C VARNEY
The central portion of the SYMPLE system—Tree construction and parsing
Penn State Univ June 1969 Thesis in Computer Science
19 W M WAITE
A language independent macro preprocessor
CACM Vol 10 No 7 July 1967 433-441

From the collection of the Computer History Museum (www.computerhistory.org)