The Synmac Syntax Macroprocessor Introduction and Manual

Version 7

Bruce J. MacLennan*

Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville
web.eecs.utk.edu/~mclennan

December 2, 2020

Abstract

A syntax macroprocessor permits parameterized text substitutions with greater syntactic flexibility than allowed with ordinary macroprocessors. This report describes the *synmac* syntax macroprocessor, which permits arbitrarily delimited macro invocations, and thus allows the definition of new statement and expression forms or even complete languages. Synmac is also a computationally complete programming language. This report defines the synmac macro language, documents a prototype implementation, and gives examples of its use.

1 Introduction

A *syntax macroprocessor* is like an ordinary macroprocessor in that it scans a *source text*, copying it to a *target text*, in the process scanning for macro definitions and macro calls, which may have parameters. Macro calls are *expanded* (replaced by their definitions) when they are encountered, with actual parameters being substituted for formal parameters. Whereas ordinary macroprocessors have a simple macro call syntax (e.g., a macro name followed by a parenthesized list of actual parameters), a *syntax macroprocessor* provides much greater syntactic flexibility in macro calls. This

report describes a syntax macroprocessor provisionally named "synmac," specifies the language, and provides examples of its use.

There are a range of application scenarios for a syntax macroprocessor. Perhaps the most common is as a tool for defining common parameterized abbreviations. In this case, the source text would be largely in the target language, with macro invocations interspersed here and there. Macros might be defined for common code templates or for introducing application-specific constructs into the language. As an extreme case of this, synmac can be used as a parameterized pattern replacement engine, which can recognize specific target-language patterns in the source text and replace them with other target-language text. (It could even be used like a text editor to globally replace incorrect text by correct text.) At the other extreme, synmac can be used to define an entirely different language, independent of the target language. In this case, the source text would be entirely in the defined language, but it would be translated by the macros into the target language. For example, we have used synmac for a prototype implementation of the *morphgen* language that translates into MatLab/Octave code [1].

> *Idea:* The synmac syntax macroprocessor is a work in progress and may continue to evolve. Ideas being considered for future versions, and thus possible incompatibilities, are displayed in this way. This report describes "version 7" of synmac; previous versions are described elsewhere [2].

2 Philosophy

2.1 Limited Syntactic Flexibility

The intent of a *syntax* macroprocessor is to permit more flexibility in the syntax of macro invocations than the relatively rigid syntax permitted by ordinary macroprocessors. But how much flexibility? In the limit, we might allow, for example, any syntax describable by a context-free grammar, but then we have a translator writing system rather than a macroprocessor. The challenge, then, is to find a balance between the syntactic rigidity of an ordinary macroprocessor and the syntactic flexibility of a translator writing system. Our design point in this space is to permit macro invocations in which parameters are separated by arbitrary delimiters and the invocation begins with one or more delimiters (see Sec. 3.2 for details). All of the delimiters are used in determining which macro to invoke, which permits defining families of related macros for different situations. This covers the most common applications of syntax macros. In particular, however, synmac does not include an expression parser, and so the user will be using, for the most part, the expression syntax of the target language.[1]

[1] It is possible, however, to define expression parsing within synmac; see Section 5.2.

2.2 Multiple Syntactic Conventions

One common use of a syntax macroprocessor such as synmac is to implement a prototype translator from some source language into a *target language*. For example, synmac has been used to implement the *morphgen* morphogenetic programming language by translating it to MatLab/Octave code [1]. Therefore, a syntax macroprocessor must negotiate among three sets of lexical and syntactic conventions: (1) the lexics and syntax of the macro language (syntax for definitions, invocations, etc.), (2) the lexics and syntax of the source language, and (3) the lexics and syntax of the target language. In the morphgen example, we are dealing with the lexical and syntactic conventions of synmac, morphgen, and MatLab/Octave. There may be irreconcilable incompatibilities, and sometimes concessions must be made in the source language. Synmac addresses this problem by having a minimal set of lexical and syntactic conventions and by allowing these to be altered to avoid incompatibilities (see Sec. 8.3, p. 28). A syntax macroprocessor cannot, in general, do much syntax checking, since if it doesn't recognize the structure of some text, it has to assume it may be in the target language, about which it's ignorant.

There are three interfaces between the source and target languages. The first is the embedding of macro invocations in target language text, which is how macros are commonly used. Therefore it must be possible to recognize the transition from the lexical structure of the target language to the lexical structure of synmac (and back) so that macro invocations are recognized, and to do this in a transparent and readable way. The second interface is between the defined delimiters of a macro and the actual parameters they surround, which can include both target language text and source text. Finally, the bodies of macro definitions can mix the source and target languages. The lexical conventions of synmac, built on whitespace and special characters, are designed to make these transitions as smooth and transparent as possible.

3 Definition of Syntax Macros

3.1 Tokens and Other Lexical Elements

The macro processor manipulates sequences of tokens, strings, and whitespace. A synmac source file is tokenized (parsed into tokens, strings, and whitespace) before any other macroprocessing.

3.1.1 Tokens

Characters are classified as alphanumeric, special, or whitespace. Each single special character is a token. Each maximal contiguous sequence of alphanumeric characters is a token. Therefore alphanumeric tokens (which we call "words") must be separated from other words by whitespace or special characters. The newline character is considered a special character. For the purpose of determining tokens, the class

of alphanumeric characters can be extended (for example to include "_"); see Sec. 8.4.1 below. These rules can occasionally lead to surprising results. For example, the decimal number "0.25" is considered three tokens: "0", ".", and "25". If this is a potential problem, then the class of alphanumeric characters can be extended to include the decimal point, which is generally safe to do.

3.1.2 Whitespace

Whitespace is generally passed through, which preserves formatting. It is also used to separate words (alphanumeric tokens).

3.1.3 Newlines

In general newlines are passed through like whitespace, but they are considered (more or less invisible) tokens, and so they can be used as delimiters in macros.

3.1.4 Strings

String literals evaluate as themselves, but when they are finally written to the output file, their quotes are removed. They are a means of passing source text into the target text and ensuring it is not altered in any way. See Secs. 8.4.2 and 8.4.3, respectively, for changing the default string quote (") and superquote (by default "\").

3.2 Patterns

3.2.1 Delimiter Structure

A macro definition has the following syntax:[2]

$$\langle \text{definition} \rangle ::= \left\{ \begin{array}{l} \textbf{syntax } \langle \text{pattern} \rangle \textbf{ means } \langle \text{template} \rangle \textbf{ endsyntax} \\ \textbf{pattern } \langle \text{pattern} \rangle \textbf{ endpattern} \end{array} \right\}$$

The syntax of a ⟨pattern⟩ is:

$$\langle \text{pattern} \rangle ::= \langle \text{delim} \rangle^{+} \; [\langle \text{param} \rangle \langle \text{delim} \rangle^{+}]^{*} \; [\langle \text{short param} \rangle] \tag{1}$$

The basic principle is that a pattern has to start with delimiters (to initiate the pattern matching process) and that parameters need to be separated by delimiters, except for the optional final short parameter, which is self-delimiting. The ⟨pattern⟩ is in effect a distributed name for the macro, since the macro will not be called unless the invocation matches the entire pattern. Whitespace is ignored in patterns.

[2]The grammatical notation is defined with the complete grammar in the appendix, p. 34.

3.2.2 Parameters

As indicated in the above syntax, parameters must be separated by one or more delimiters. Macros can have three kinds of parameters, which we call *long parameters*, *short parameters*, and *unevaluated parameters*, but these terms are somewhat misleading. The syntax of formal parameters is as follows:

$$\begin{aligned} \langle\text{param}\rangle &::= \langle\text{short param}\rangle \mid \langle\text{long param}\rangle \mid \langle\text{uneval param}\rangle \\ \langle\text{short param}\rangle &::= \tilde{}\langle\text{word}\rangle \\ \langle\text{long param}\rangle &::= \&\langle\text{word}\rangle \\ \langle\text{uneval param}\rangle &::= '\langle\text{word}\rangle \end{aligned}$$

A short parameter expects to match a single well-defined source expression, which is a single token or a macro invocation, which can, however, have other invocations in its actual parameters (see Sec. 4.1 for syntax). Although it's called a "short" parameter, and that is often its use, the corresponding actual can in fact be many lines long, so long as it's a single macro invocation. Nevertheless, the macroprocessor knows when it has got to the end of a short actual parameter, and that is its most important characteristic.

A long parameter, in contrast, expects to match a sequence of zero or more source expressions, and therefore it can match a sequence comprising a mixture of tokens and macro invocations of any length (see Sec. 4.1 for syntax). The macroprocessor knows it has got to the end of a long parameter by matching the delimiters that are required to terminate it.

An unevaluated parameter expects to match a sequence of zero or more tokens up to a specified delimiter, much like a long parameter, but it does not evaluate any macro invocations that it might contain.

Any token can be used as the name of a formal parameter, but it is normal and good style for it to be a descriptive alphanumeric token, and not a special character (hence, ⟨word⟩ in the above grammar).

3.2.3 Example Patterns

Here are some example patterns from the morphgen language definition [1] (template bodies omitted):

```
syntax del ~var means ... endsyntax // gradient
syntax del^2 ~var means ... endsyntax // Laplacian
syntax substance ~name: &variables behavior: &equations end
    means ... endsyntax
syntax display running ~subst as contours means ... endsyntax
syntax report Courant number for ~vector means ... endsyntax
```

Here "//" marks an end-of-line comment (Sec. 7.1).

3.2.4 Match Order

The macroprocessor attempts to match an expression to the most recently defined macro, and if it does not match, it works backwards, trying previously defined macros. To put it in other words, macro definitions are placed on a stack, and the macroprocessor tries them from top to bottom. Therefore, macros can be redefined, and later definitions supersede earlier ones. This means that later macros with a more specific syntax that handles special cases can partially block earlier macros with a more general syntax that handle the general case (somewhat like methods in subclasses of a class). This also means that intrinsic macros (such as **syntax**) can be redefined, which might be useful in some cases, but the original definitions will be inaccessible.

3.3 Pattern Elements

A pattern is a sequence of one or more pattern elements, which include token delimiters, newline delimiters, commit flags, and formal parameters (described in Sec. 3.2.2). The syntax of ⟨pattern⟩s is given in Eq. 1 for which ⟨delim⟩ is defined:

$$\langle \text{delim} \rangle ::= \{\ \langle \text{token} \rangle \mid \$ \mid \# \mid \textbf{dedent}\ \}\ [\ !\ ] \tag{2}$$

That is, a ⟨delim⟩ is a token delimiter, or a newline delimiter ($), or an endline delimiter (#), optionally followed by a commit flag ("!"). (These special characters can be changed; see Sec. 8.3.)

3.3.1 Token Delimiters

Any token can serve as a delimiter that must be present in a macro invocation. Whitespace around delimiters is ignored for purposes of pattern matching.

3.3.2 Newline and Endline Delimiters

The special pattern element "$" (which can be changed, Sec. 8.3.3) represents a required newline as a delimiter. This is convenient especially for making a newline the final, closing delimiter in a template. For example:

```
syntax for (~x, ~y) within &radius of (&xcent, &ycent):
    ~var = &expr $ means ... endsyntax
```

permits an invocation such as this on a line by itself:

```
for (h, v) within 0.06 of (0.1, -0.225): P = 1 - F(h,v)
```

The usual use of a newline delimiter is as the last delimiter in a pattern, as shown above, but it can also be used as a medial or initial delimiter. Especially in the latter case, the user should be aware that every newline in the source text will cause the

macroprocessor to perform a pattern match, to see if it should invoke the newline-initial macro; this could be slow and lead to unexpected macro substitutions.

As defined, the newline delimiter consumes the newline that it matches; this is often useful . However, there are also cases in which it would be convenient to have several nested macro calls all terminated by a single newline, and for this purpose it would be useful if the newline delimiter did not consume the newline that it matches. This case is represented by the *endline* delimiter, which is represented by "#" in patterns. An example application is an end-of-line comment, which does not consume the endline (cf. Sec. 7.1):

```
syntax // 'comment # means{}endsyntax
```

3.4 Dedent Delimiter

Indenting can be used to delimit syntax macro parameters. The **dedent** pattern element delimits a parameter by "dedenting," that is, by an indent level less than or equal to that at which the macro invocation began.[3] For this purpose, the indent level is determined by looking ahead to the next nonblank line; the end of file is considered unindented. As a simple example, the following macro accepts and discards an unevaluated indented parameter, which is intended as documentation:

```
syntax documentation: 'comment dedent means{}endsyntax
```

This is an example invocation, which expands as just the two assignment statements:

```
documentation:
  This is the first stage of initialization of the matrices.
  The indices are initialized for the beginning of the search.
outer_index = 0
inner_index = 0
```

The delimiting newlines are not consumed, so a particular indent level will match all open dedents for macros invoked at or above that level. For example, this defines an "until loop" for python:

```
syntax // 'comment $ means{}endsyntax
syntax until &condition: &body dedent means//
while not(condition):
    body
endsyntax
```

('//' in the "until" definition absorbs the end of line so that the indenting of "until" is preserved; otherwise "while" begins on a new line.) Here is an example nested invocation:

[3]Therefore a pattern cannot begin with a dedent delimiter.

```
until x<1:
  until y<1:
    x = sqrt(x)
    y = sqrt(y)
    n += 1
print(n)
```

A macro pattern can have several **dedent**s, but they all will be matched by the same indent level, so any actual parameters after the first will be empty.

The body of a macro may contain invocations of macros with dedent-delimited parameters. For this purpose, the body of the macro is evaluated as though it is unindented (i.e., starts on a new line). For example,

```
syntax header means{ documentation:
                        This is some documentation
                        that will be ignored.
                     initialization()
}endsyntax
```

is expanded as though it were written,

```
syntax header means{
 documentation:
    This is some documentation
                        that will be ignored.
                     initialization()
}endsyntax
```

and so "initialization()" becomes part of the comment. Therefore, to avoid confusion, it is good practice to write starting on a new line the template of macro definitions that invoke dedent delimited macros, as in this example:

```
syntax header means{
documentation:
    This is some documentation
    that will be ignored.
initialization()
}endsyntax
```

3.4.1 Commit Flag

When the macroprocessor tries to match the source text to a macro definition, if a sought delimiter is missing, it simply assumes that the source text is not a call of that macro. It backtracks and tries other macro definitions. Therefore, if you make

an error in the syntax of a macro call, it may simply be passed unexpanded into the target text. In many cases, however, as macro designers we know that the first tokens determine the macro to be called, and if later delimiters are missing we want a diagnostic message; we don't want the macroprocessor to ignore it. For this purpose we have a *commit flag* ("!" by default) that can be put after any delimiter. It means that if we have got this far in the pattern matching, then the rest of the delimiters must be present. If any are not, synmac can produce an informative error message. Here is an example of its use:

```
syntax ||! &exp || means L2norm(exp) endsyntax
```

This tells the macroprocessor that whenever it sees "||" it should expect to find a matching "||", and that if it does not, it should generate a useful error message. Of course, you can do this only if this is the only macro beginning with "||" and only if "||" is not used for other purposes in the source text.

3.5 Templates

A template is the body of a macro, which is a (possibly) empty sequence of tokens with optional interspersed whitespace. (As explained later, a ⟨template⟩ is syntactically an ⟨expr seq⟩, Eq. 3, p. 11.) Some of these tokens are formal parameter names. Formal parameters in the template are not marked in any special way, but the macroprocessor recognizes them because they are bound to the values of the corresponding actual parameters. When a macro is invoked, its body is processed in two steps: First, the formal parameters are replaced by the actual parameters to which they are bound. This includes substituting actuals for formals inside metaquotations (Sec. 3.7). Second, the resulting expression sequence is evaluated (expanded), and the resulting expansion replaces the macro invocation. Any whitespace in the template is retained, which preserves formatting.

> *Idea:* This treatment of metaquotations combines actual-formal substitution with delayed evaluation, and provides a capability similar to a lambda expression (so the metaquotes actually denote delayed evaluation). It is unclear whether it is the best choice. Perhaps metaquotes should work similar to string quotes and prevent all processing (but then we might want another mechanism to do substitution without evaluation).

3.6 Constant Macros

In Section 2.2 we mentioned the issues that arise from having to deal with three sets of lexical and syntactic conventions (synmac, source language, target language). One of these is that synmac knows nothing about the syntax of the target language (e.g., that parentheses and brackets should be matched). For example, if we define a syntax macro:

```
syntax succ(&arg) means arg + 1 endsyntax
```

and call it like this:

```
x = succ(A[(N-1)*2])
```

then the result will be

```
x = A[(N-1 + 1 *2])
```

which is not what was intended. This is because the parameter &arg matched the actual up to the first close parenthesis, that is, it matched "A[(N−1", and substituted it for "arg" in the body of "succ". The problem is that synmac does not know that we expect parentheses and brackets to be matched in macro arguments. We can inform it of these requirements by declaring them as *constant macros*:

```
pattern ( &expr ) endpattern
pattern [ &expr ] endpattern
```

These are like standard syntax declarations, but they have no bodies (templates). This means that synmac will recognize and process them like ordinary macros, but treat them like constant functions (i.e., they expand as themselves, but don't result in recursive invocation). If no other macros begin with an open parenthesis and bracket, then we will get better error diagnostics by using the commit flag:

```
pattern (! &expr ) endpattern
pattern [! &expr ] endpattern
```

3.7 Metaquotations

Expansion of a token sequence can be suppressed by surrounding it with metaquotes (by default {}), which can be nested. When a metaquoted sequence is evaluated, the outermost metaquotes are removed. Therefore, when an actual parameter contains metaquoted sequences, the outermost metaquotes are removed in the substituted parameter. Actual-formal parameter substitution *does* take place in metaquotations in macro templates (see Sec. 3.5). The default metaquotes can be changed (see Sec. 8.3.5).

4 Expansion Semantics

4.1 Expansion as Expression Evaluation

The source text is really an expression sequence, which is evaluated ("expanded") to yield the target text. Plain text (that is, text that is not recognized as a macro invocation) is treated as constant text that is passed unchanged to the output. Embedded

macro invocations are evaluated and replaced by their results (expansions), as explained below. (Syntax definitions and pragmatic commands are treated as intrinsic macro invocations with side-effects.)

More precisely, an *expression sequence* is a sequence of zero or more expressions possibly separated and surrounded by whitespace:

$$\langle \text{expr seq} \rangle ::= [\langle \text{whitespace} \rangle] \; [\; \langle \text{expr} \rangle [\langle \text{whitespace} \rangle] \;]^* \tag{3}$$

The elements of the expression sequence are evaluated in order. Whitespace is copied into the result; expressions are replaced by their values.

An *expression* is either a macro invocation, a string literal, a metaquoted expression sequence, or a token (special character or alphanumeric word) that does not begin a valid macro invocation (the latter are called *undefined tokens*). Therefore the syntax for an expression is

$$\langle \text{expr} \rangle ::= \left\{ \begin{array}{l} \langle \text{invocation} \rangle \\ \langle \text{string} \rangle \\ \mathbf{\{} \langle \text{expr seq} \rangle \mathbf{\}} \\ \langle \text{token} \rangle \end{array} \right\} \tag{4}$$

The result of evaluating an undefined token is that token (i.e., they are treated as constants). The result of evaluating a string literal is that string; quotes are not removed during evaluation, but when the string is finally written to the output file, the quotes are removed. The results of evaluating a metaquotation is the enclosed expression sequence with the outer metaquotes deleted but inner nested metaquotes preserved. The evaluation of macro invocations is described below.

> *Idea:* It is not obvious that the preceding is the best way of evaluating strings and metaquotations. Stripping off the quotes allows the enclosed text to be evaluated later; that is, quotation *delays* evaluation, and this is sometimes useful. For example, we generally want to delay the evaluation of the arguments to the **syntax** macro. On the other hand, in other circumstances we want to completely prevent the evaluation of some source text, so that it is preserved into the target text, which argues for leaving the quotes on no matter how many times the quotation is evaluated. In this case, we might want an intrinsic operator to force evaluation by removing the quotes.

4.2 Eager Evaluation

The macro processor uses an eager evaluation order, that is, actual parameters are evaluated (expanded) before they are substituted for the formal parameters in the body of the macro.

4.3 Evaluation Order

4.3.1 Expanding an expression sequence

The macroprocessor proceeds through an expression sequence, expression by expression, concatenating their values to produce the result of evaluating the expression sequence (with intervening whitespace preserved). An expression is evaluated by inspecting its first token. If it is a string literal or a open metaquote, then the quotation is processed as described above (Sec. 4.1). Otherwise the macroprocessor attempts to match a pattern.

4.3.2 Attempting a pattern match

The macroprocessor attempts to match each defined macro pattern to the expression text, starting with the most recently defined macro and working backwards. (Thus later definitions supersede earlier ones.) If at any point the expression does not match the pattern, then the match it abandoned. If the pattern has been *committed* (Sec. 3.4.1) then a diagnostic is generated (showing the pattern and the unmatching text). Whenever a match is abandoned (whether committed or not), the macroprocessor bactracks over any matched text and proceeds to try earlier definitions. (Note, however, that if the evaluation of actual parameters has had side-effects, these side-effects will not be undone.) Therefore you can have alternative macro definitions, or more generic definitions followed by more specific ones. If a word does not begin any successful pattern match, then it is considered an undefined token and is returned as the value of the expression.

Consecutive delimiters in the pattern are matched against successive delimiters in the expression, but intervening whitespace is ignored. Thus, for example, the pattern text "behavior:" (two tokens) will match the expression text "behavior␣:" (two tokens with intervening whitespace), and the pattern ">=" will match ">␣=". Newline, endline, and dedent delimiters attempt to match newlines in the expression text.

4.3.3 Evaluating an actual parameter

When a formal parameter is encountered in the pattern, the macroprocessor attempts to parse and evaluate (expand) a corresponding actual parameter. If the formal parameter is a short (~) parameter, then the macroprocessor skips whitespace in the token sequence and recursively evaluates a single expression. Therefore, the corresponding actual parameter text must conform to ⟨expr⟩ (Eq. 4). Because initial whitespace is trimmed from a short actual parameter, and because parsing of a short actual parameter stops as soon as a complete ⟨expr⟩ is found, effectively whitespace is trimmed from the beginning and end of short actual parameters. The evaluated actual parameter is bound to the formal parameter (eager evaluation).

If the formal parameter is a long (`&`) parameter, then the macroprocessor expects to find an expression sequence (Eq. 3), which must be terminated by the delimiter that follows the long formal parameter in the pattern (Eq. 1, p. 4). The macroprocessor will recursively evaluate the expression sequence, as described in this section (Sec. 4.1), until it finds the terminating delimiter. Any whitespace in the actual parameter will be preserved in the value bound to the formal parameter. If the required delimiter is not found, then the pattern match fails and the macroprocessor backtracks (but any side-effects resulting from evaluation of the actual will remain).

If the formal parameter is an unevaluated (`'`) parameter, then the macroprocessor expects to find a token sequence terminated by the delimiter that follows the formal parameter. If the required delimiter is not found, then the pattern match fails and the macroprocessor backtracks. The unevaluated actual parameter, including any whitespace, is substituted for the corresponding formal wherever it occurs in the body of the macro. Note, however, that although the actual parameter is substituted in unevaluated form, subsequent evaluation of the body might cause it to be evaluated (see Sec. 4.3.5).

4.3.4 Substituting actual parameters for formal parameters

If a token sequence successfully matches a macro's pattern, then the actual parameters are substituted for the formal parameters in the macro's template or body. The tokens (including whitespace) in the template are processed in order, and whenever a word is found to be bound, it is replaced by the token sequence to which it is bound. Formal parameters in the template do not need to be flagged, for example:

```
syntax del ~var    // gradient
  means {grad(var, delta_s)} endsyntax
syntax del^2 ~var // Laplacian
  means {lapl(var, delta_s, delta_s)} endsyntax
syntax space! : &xlwb < x < &xupb, &ylwb < y < &yupb$ means
  x_lwb = xlwb;
  x_upb = xupb;
  y_lwb = ylwb;
  y_upb = yupb;
  x_extent = x_upb - x_lwb;
  y_extent = y_upb - y_lwb;
endsyntax
```

4.3.5 Evaluating the body

After the actuals have been substituted for the formals in the template of the macro, the resulting macro body is evaluated as an expression sequence and becomes the

value of the macro invocation. Evaluation of the body may have side-effects, for example, if it includes **syntax** definitions.

4.4 Preventing Evaluation

Surrounding text with the metaquotes ({} by default) prevents evaluation (expansion) of the enclosed text. Their most common use is to surround the arguments of the syntax definition commands, which are otherwise evaluated like any other actual parameters. If you're doing something tricky (Sec. 7), you might want to compute the pattern or macro template, but this is not the usual case. You can usually get away without metaquoting the arguments of syntax definitions, but it's better practice to metaquote them (unless you're doing something tricky) and thereby avoid hard-to-find errors.

5 Examples

In this section we present several examples of synmac as an interpreter. This is not its most common use, but it illustrates its computational power (it is in fact Turing-complete).

5.1 Peano Arithmetic

By Peano arithmetic, we mean the definition of basic arithmetic operations in terms of integers defined as successors of zero. It illustrates a kind of equational programming. We begin with some preliminary definitions and declarations:

```
syntax || 'text # means{}endsyntax
||
|| Peano Arithmetic
||
#set syntax def as $
#set pattern declare $
```

The first line defines "||" as an end-of-line comment. The last two lines declare alternative syntax for macro definitions, which is more appropriate to recursive arithmetic. The next group of definitions defines several numerals in terms of the successor:

```
declare succ[~arg]
def 1 as succ[0]
def 2 as succ[1]
def 3 as succ[2]
```

The "declare" line ensures that synmac recognizes that square brackets should be matched (on this issue, see Sec. 3.6). Since the templates of the numeral definitions

are not surrounded by metaquotes, they are expanded at definition time. Therefore, for example, "3" is defined to be "succ[succ[succ [0]]] " not "succ[2]" as you might expect. This makes no difference here, but it could make difference with recursive macros (leading, for example, to infinite recursion).

Now we can define addition recursively, both prefix and (fully parenthesized) infix notation:

```
def sum[succ[~M], ~N] as {succ[(M + N)]}
def sum[0, ~N] as {N}
def (~M + ~N) as {sum[M,N]}
```

Notice that we have surrounded the bodies of the macros with metaquotes to delay evaluation. They are not strictly necessary if the definitions are ordered as they are here, but it's good synmac coding and will prevent hard-to-diagnose errors. Subtraction is defined:

```
def dif[succ[~M], succ[~N]] as {(M - N)}
def dif[~M, 0] as {M}
def (~M - ~N) as {dif[M,N]}
```

Using addition we can define multiplication:

```
def prod[succ[~M], ~N] as {(N + (M * N))}
def prod[0, ~N] as {0}
def (~M * ~N) as {prod[M,N]}
```

We can test our definitions by entering:

```
"3 + 2 + 1 =" ((3+2)+1)
"3 * 2 * 1 =" ((3*2)*1)
"3 + 3 - 2 =" ((3 + 3) - 2)
```

This source text expands as the following target text, which is correct:

```
3 + 2 + 1 =   succ[ succ[ succ[ succ[ succ[ succ[0]]]]]]
3 * 2 * 1 =     succ[ succ[succ[succ[succ[succ[0]]]]]]
3 + 3 - 2 =       succ[succ[succ[succ[0]]]]
```

(There are redundant spaces, which could be eliminated at the expense of making the **syntax** definitions less readable.) It is hard to read these long strings of successors, so we can define (limited) decimal decoding:

```
def decimal ~N as {dec[N, "0", "1", "2", "3", "4", "5", ||
  "6", "7", "8", "9", "10"]}
def dec[0, ~first, &rest] as {first}
def dec[ succ[~N], ~first, &rest] as {dec[N, rest]}
```

The "decimal" macro uses an auxiliary macro "dec", which counts off the "succ"s in parallel with the numeral strings. With these definitions, the source text

```
"2 * 3 - 2 =" decimal((2*3)-2)
"(3 * 2) - 2 + (2 * 3) =" decimal (((3*2)-2) + (2*3))
```

expands as:

```
2 * 3 - 2 =          4
(3 * 2) - 2 + (2 * 3) =                  10
```

> *Idea:* Obviously, this is a stupendously inefficient implementation of arithmetic. If in the future it is found useful to have synmac do arithmetic, it will be included by means of intrinsic (built-in) macros.

5.2 Infix Expressions

Although synmac does not include an expression parser, it is possible to define syntax macros that provide limited infix expression parsing. We begin by changing the metaquotes so that we can use curly braces as delimiters, and declare balanced parentheses as a pattern:

```
syntax || 'text # means{}endsyntax   || end-of-line comments
#set metaquotes < >
pattern <(&E)> endpattern
```

We begin by handling "*" and "/" signs:

```
syntax <term {&F}> means <factor{F}> endsyntax
syntax <term {&F * &T}>
  means <prod[factor{F}, term{T}]> endsyntax
syntax <term {&F / &T}>
  means <quot[factor{F}, term{T}]> endsyntax
```

Since patterns are tried in reverse order, the patterns "term {&F * &T}" and "term {&F / &T}" will attempt to match an expression in curly braces up to the first "*" and "/", which will be bound to F (standing for "factor"). The remainder of the text up to the close curly brace will be bound to T (standing for "term"). Since the text bound to T may include additional "*" and "/" signs, it is submitted recursively to "term" for further processing. If, however, the argument of "term" does not contain these operators, then it is submitted to "factor" for further processing, the rules for which are as follows:

```
syntax <factor {&F}> means F endsyntax
syntax <factor {(&E)}> means <expr{E}> endsyntax
```

The second line submits a parenthesized expression to "expr" for expansion; the first, which handles simple unparenthesized factors (such as variable names and numbers), returns its argument without further processing. Here are some simple tests:

```
term{A / B}
term{2*A*B}
```

which expand as:

```
quot[ A , B ]
prod[ 2 , prod[ A , B ] ]
```

The "+" and "−" signs are handled similarly by expr:

```
syntax <expr {&T}> means <term{T}> endsyntax
syntax <expr {&T + &E}>
  means <sum[term{T}, expr{E}]> endsyntax
syntax <expr {&T − &E}>
  means <dif[term{T}, expr{E}]> endsyntax
```

We can test it with these expressions

```
expr {2*B + C}
expr {A−2*C}
expr {1/(C+D)}
```

which expand correctly:

```
sum[ prod[ 2 , B ] , C ]
dif[ A , prod[ 2 , C ] ]
quot[ 1 , sum[ C , D ] ]
```

We can now use infix expressions in appropriate contexts such as assignment statements:

```
syntax let ~D := &E; means <setq[D, expr{E}]> endsyntax
```

We can test it with this text:

```
let X := (A+B)*(C–D);
let Y := A − B − 1;
```

which leads to the expansion (with some blanks deleted for readability):

```
setq[X, prod[ sum[ A , B ] , dif[ C , D ] ] ]
setq[Y, dif[ A , dif[ B , 1 ] ] ]
```

So we can define infix expressions, but if you look closely at the above examples, you will see that some of the translations are surprising. For example, A–B–1 is translated dif[A, dif[B,1]], which means the same as A–(B–1), that is, the operations associate

to the right, which is not the usual convention. It is possible to write synmac syntax macros that implement left-associative operators, but they are more complicated, and do not need to be presented here.[4]

5.3 Simple LISP

LISP-like list processing provides another example of synmac as a computational engine and illustrates some programming techniques. We begin, as before, with some preliminary declarations:

```
syntax || 'text # means{}endsyntax
|| LISP-style Lists
#set syntax def = ;
#set pattern declare ;
```

In this case, we've decided to terminate the declarations with semicolons. Next we declare the essential relations that define a list (represented by the constructor cons(H,T)):

```
declare cons (~H, ~T);
def car(cons(~H,~T)) = H;
def cdr(cons(~H,~T)) = T;
```

The first line tells synmac that "cons(H,T)" is a well-defined structure, and the last two define the "car" and "cdr" functions to select the two halves of this structure. To match these patterns, the argument to car or cdr will have to be an explicit cons structure (since that is part of the pattern). In general, we might want to compute the arguments, so we wrap these in the functions that users are expected to use:

```
def first(~L) = {car(L)};
def rest(~L) = {cdr(L)};
```

We can try these out by defining a list and testing a couple of identities:

```
def Lst = cons(A,cons(B,nil));
"first Lst =" first(Lst)
"rest Lst =" rest(Lst)
"Lst =" cons(first(Lst), rest(Lst))
```

This is the expansion:

[4]Left-associative infix operators can be handled as follows. First define a set of recursive macros that reverse the order of terms connected by + and – and also reverse the order of factors connected by * and / (continuing into parentheses if necessary), replacing the normal operators by specially marked "reversed" operators. For instance, these macros transform "A / (B – 2)" into "(2 ^– B) ^/ A". Next, write recursive macros similar to those in Sec. 5.2 that transform the reversed operators into function calls. For example, "(2 ^– B) ^/ A" becomes "quot[A, dif[B, 2]]".

```
first Lst =   A
rest Lst =   cons(B, nil)
Lst = cons(   A,   cons(B, nil))
```

Instead of writing lists in terms of cons's, it's more convenient to have a list notation, such as $[C, D, E]$, which we can define recursively:

```
def [] = nil;
def [~H] = {cons(H, nil)};
def [~H, &T] = {cons(H, [T])};
```

We can test both simple and nested lists:

```
def Lst2 = [C, D, E];
def Lst3 = [[U, V], [X, Y]];
"Lst2:"Lst2
"Lst3:"Lst3
```

which expands as:

```
Lst2:   cons(C, cons(D, cons(E, nil)))
Lst3:   cons(cons(U, cons(V, nil)), cons(cons(X, cons(Y, nil)), nil))
```

Reading these nested cons's is tedious, so we can define a "pretty print" function. Here is a first attempt, which uses an auxiliary function to deconstruct the list:

```
def prettyprint(~L) = {"["ppaux(L)"]"};
def ppaux(cons(~H,~T)) = {H, ppaux(T)};
def ppaux(cons(~H, nil)) = H;
def ppaux(nil) = ;
```

The square brackets are quoted because we don't want to invoke the square bracket macro that we previously defined! Notice also the order of the ppaux definitions; the second is more specific than the first, and so it must follow the first. Here is a test:

```
"Lst2 =" prettyprint(Lst2)
"Lst3 =" prettyprint(Lst3)
```

which produces:

```
Lst2 =   [ C,   D,   E]
Lst3 =   [ cons(U, cons(V, nil)),   cons(X, cons(Y, nil))]
```

Clearly, this doesn't handle nested lists correctly, but we can fix that by adding:

```
def ppaux(cons(cons(~H,~T),~U))   || handle nested lists
  = {prettyprint(cons(H,T)), ppaux(U)};
def ppaux(cons(cons(~H,~T), nil)) = {prettyprint(cons(H,T))};
```

Then the test

```
"Lst3 =" prettyprint(Lst3)
```

expands correctly:

```
Lst3 =  [  [ U,  V],  [ X,  Y]]
```

In the preceding examples, we have used an equational style of programming, in which patterns are used to test for arguments in various forms. For more general programming it may be more convenient to have conventional boolean values and conditionals. We begin with a way of testing whether a list is empty. As before, we define a low-level pattern that matches a cons structure, and wrap it in a user-callable "empty" function:

```
def null(nil) = true;
def null(cons(~H,~T)) = false;
def empty(~L) = {null(L)};
```

We can try them out:

```
"empty (nil) =" empty(nil)
"empty (Lst) =" empty(Lst)
```

and get the following correct answers:

```
empty (nil) =  true
empty (Lst) =  false
```

But what are "true" and "false"? To make them useful, observe that they are fundamentally choices and therefore define them (using an idea common from λ calculus programming language semantics):

```
def true(&T,&F) = T;
def false(&T,&F) = F;
def if ~bool then ~tbranch else ~fbranch fi
 = {bool({tbranch},{fbranch})};
```

The "if" macro takes the boolean value and applies it to the true and false branches so that the boolean can select one or the other. The true and false branches are metaquoted to delay evaluation. Here's a simple test

```
if empty(nil) then correct else incorrect fi
if empty(Lst) then incorrect else correct fi
```

which returns:

```
    correct
    correct
```

We can now use the conditional and other list operations to define a recursive function to append two lists:

```
def append(~L,~M) = {   || append two lists
 if empty(L) then M else cons(first(L),append(rest(L),M)) fi};
```

We can test it as follows:

```
"Lst^Lst2^Lst3=" prettyprint(append(Lst, append(Lst2, Lst3)))
```

which produces the correct result:

```
Lst^Lst2^Lst3= [A, B, C, D, E, [U, V], [X, Y]]
```

These are, I think, sufficient examples to illustrate the programming capabilities of synmac.

6 Dealing With Problems

Experience has shown that debugging syntax macro definitions and invocations can be very difficult. This is in part because of the flexible syntax permitted in patterns, which can match unexpected regions of the source text. Here we discuss a few common symptoms of problems and how they can be fixed; in hard cases you may have to use the debugging commands (Sec. 8.5).

In general, the user is advised to proceed incrementally. Introduce one new macro at a time and try it out before adding others. Test simple cases first. (You know, the usual good programming practices!) See Section 5 for examples.

6.1 Symptom: Macro is not evaluated

If a macro is not expanded (with the common symptom being that its invocation is copied into the target text), then the most obvious cause is that you have got the form of either the definition or the call incorrect. If one call out of many doesn't expand, then you likely have an error in it; compare it with the calls that did expand. If none of them expand, then you probably have an error in the definition. Inspect it carefully and, if necessary, use **#set definitions on** to see how the macroprocessor read the definition (Sec. 8.5.1).

Sometimes an error in an earlier macro invocation can prevent a later one from expanding. This can occur if an actual parameter of the former extends further than intended (due, for example, to a mistyped terminating delimiter) and absorbs part of the later invocation and prevents it from expanding.

If you have several **syntax** definitions that define different cases of a single conceptual macro, then make sure they are ordered correctly (see Sec. 5 for examples). Remember that macros are tried from most recently defined to earlier, so generally

more specific cases should be defined after more general (which is perhaps the opposite of the natural order).

6.2 Symptom: Incorrect actual parameters

Sometimes the actual parameter substituted for a formal parameter is not what you expect. Sometimes it's more then you expected, which can happen if you mistyped the delimiter that was supposed to terminate an actual, and so the actual continued until the next occurrence of the correct delimiter. Sometimes the actual parameter is less than you inspected, which can happen if it found the terminatig delimiter within the intended actual, but not within a correct embedded invocation. Finally, since actual parameters are evaluated (when not quoted), the substituted parameter might be different from what you expected because it expanded differently than you expected.

6.3 Symptom: Non-termination

The most obvious cause of non-termination is, of course, a recursive macro. If necessary, use the **#set message** and **#set pause** commands to isolate which macro is causing the problem (Sec. 8.5). Look carefully at the offending macro definition, especially whether the recursive invocation is metaquoted as necessary (otherwise the recursion may be expanding at definition time rather than call time). If some invocations of the macro work correctly, look carefully at those that don't, since sometimes an incorrect actual parameter can cause non-termination in a recursive macro (think of taking the factorial of a negative number). If the offending macro begins with common words or symbols, check for accidental invocations of the macro in comments or other text where you don't intend to invoke it.

6.4 Symptom: Repeated side-effects

Sometimes it becomes apparent that side-effects of evaluation (including, for example, **syntax** declarations) are occurring more than once. That means that the source text that produces the side-effect is being evaluated more than once. This can occur during the normal backtracking process that takes place as the macroprocessor attempts to match the source text against the defined patterns, because backtracking backs over the source and target text, but does not undo side-effects. Sometimes this occurs because the macroprocessor is trying to invoke a macro, but the invocation is failing to match any of the patterns. To diagnose the problem, try to find out which invocation is being expanded repeatedly by, for example, using **#set message** (Sec. 8.5).

> *Idea:* Change backtrack so that it undoes **syntax** definitions (the most common and problematic side-effects). This is not too hard to do, since the definitions are stacked.

6.5 Symptom: Syntax errors in unexpected places

This problem can arise if you use common English words as the initial delimiters in a pattern. If you happen to use these words in comments or other source text, the macro processor may try to expand them as macros, generating either syntax error messages or strange target text. The solution is to quote or metaquote text that might be recognized as macro calls.

7 Tricks

7.1 Comments

You can define your own comment conventions by defining a macro that expands as an empty string. The first example defines "//" through and including the next newline to be a comment, and the second defines "||" up to but *not* including the next newline to be a comment:

```
syntax // ! 'comment $ means{}endsyntax
syntax || ! 'comment # means{}endsyntax
```

Notice that the empty metaquotes are required to have a completely empty macro body. The macro's parameter is unevaluated (') in case the comment text happens to contain any macro invocations. This ensures they are not expanded, which could cause problems if they contain errors (i.e., incorrect invocations), infinite recursion, or side-effects (see Sec. 6).

> *Idea:* Treating comments as ordinary macro invocations can have undesirable consequences. For example, a short formal parameter expects to match a single expression, but adding a comment to the corresponding actual makes it two expressions (the original actual plus the comment), which the short formal will not match. This can be avoided by using a long formal instead, but this may be undesirable for other reasons. The solution might be to build into synmac real comments, which are treated as whitespace, not expressions.

7.2 Suppressing Unwanted Whitespace

White space is treated as constant text and passed through the macroprocessor like any other unevaluated text. This tends to preserve formatting from the source text to the target text. However it can also lead to redundant blank lines in the target text. For example, blank lines surrounding macro definitions and newlines after **endsyntax**, which are useful for readability, will be passed through. A small amount

of whitespace can be avoided by defining a comment macro, as described in the previous section, and using it to absorb redundant whitespace and an newline (indicated by "$"). For example:

```
syntax // 'comment $ means{}endsyntax
//
// A macro definition
//
syntax some_pattern
means its_template endsyntax//
//
```

Whitespace is removed automatically from the patterns, so it doesn't need to be commented out between **syntax** and **means**.

Large blocks of definitions can generate a lot of whitespace, but it is tedious and ugly to comment out each redundant newline, as suggested above. Instead, output from any block of text that is executed only for its side-effects (such as a long series of definitions), can be suppressed by embedding it in an invocation of a macro such as this:

```
syntax discard begin! &text discard end means{}endsyntax
```

It is effectively a comment that evaluates its argument for its side-effects. For example, an included file that contains only definitions might be structured like this:

```
syntax definitions begin! &X definitions end means{}endsyntax
definitions begin
  definition 1
  definition 2
    ...
  definition N
definitions end
```

Whitespace and any sort of explanatory text can be included around the definitions (so long as it doesn't include troublesome macro invocations!) and it will be discarded.

7.3 Using Macros as Variables

Since the most recent definition of a macro to match an expression is used, macros can be used as variables. For example, the following uses a macro "_disp_int" as a variable, which holds a quantity that has a default value 0 that can be changed by another macro:

```
syntax _disp_int means 0 endsyntax
syntax display interval = ~N $ means
{syntax {_disp_int} means N endsyntax}
```

```
endsyntax
```

Note that the name "_disp_int" must be metaquoted in the inner syntax declaration, since otherwise it would be replaced by its current value (some number, presumably), and that value would be used as the pattern. On the other hand, the template for "_disp_int" cannot be metaquoted, because we want it to be defined as the value of N, not the word "N". This is an example where the often optional metaquotes are important to control evaluation time!

7.4 Accumulating Lists

Macros can be redefined in terms of their previous values in order to accumulate lists and for similar purposes. For example, suppose we want to accumulate a list of mentioned variables because we need to do some post processing on them. First, initialize an empty list:

```
syntax _variables means endsyntax
```

Then, each time we need to remember another variable name, we add it to the beginning of the list, which we can do with a macro such as this:

```
syntax {remember ~name} means {
syntax {_variables} means name; _variables endsyntax
}endsyntax
```

Notice the use of metaquotes in the inner syntax definition, for they are critical. In the pattern, the name "_variables" is metaquoted because we do not want it evaluated; we want to redefine "_variables", not the value to which it's bound. In the template part of this definition, it is not metaquoted, because we want it to be replaced by its current value (the previous list of variables) before we add the new variable name. If in various places I do

```
remember X
remember Y
remember Z
```

then the final value of "_variables" will be "Z; Y; X; ". (There is nothing special here about the use of ";" as a delimiter.) Notice that the variables are in reverse order, because they are added to the beginning of the _variables macro. If, instead, you need the variables in the order they are mentioned, use these definitions:

```
syntax {remember ~name} means {
syntax {_variables} means _variables name; endsyntax
}endsyntax
```

Then _variables will be "X; Y; Z; ".

7.5 Processing Lists of Items

Once we have a list of things, such as the variable names in the previous example, we may want to apply the same macro to each of them. This can be accomplished with macro definitions such as these:

```
syntax {handle(~name; &rest)} means {
  do something with name
  handle(rest)} endsyntax
syntax {handle()} means endsyntax
syntax {handle everything} means handle(_variables) endsyntax
handle everything
```

The use of metaquotes here is somewhat subtle, so we'll go through it in order. The second-to-last line defines "handle everything" to be the result of evaluating "handle(_variables)", which will be the result of evaluating "handle(X; Y; Z;)" or "handle(Z; Y; X;)" (using our previous examples). This will match the first definition of "handle," with short parameter "name" bound to the first variable (say, "X") and long parameter "rest" bound to, for example, "Y; Z; ". This will expand as:

```
  do something with X
  handle(Y; Z; )
```

This invokes "handle" recursively to do something with Y and Z, and in the final case "handle()" is invoked, which expands as the empty text. The last line, the invocation of "handle everything", then expands as

```
  do something with X
  do something with Y
  do something with Z
```

There are many variations on this structure, but for it to work it is important that the list be substituted before its containing expression ("handle(...)" in this case) is evaluated, which happens when "do everything" is called. This is because the macroprocessor parses the invocation before it evaluates its constituent expressions.

7.6 Computed Invocations

The previous example illustrates another trick. Sometimes we want to compute an entire macro invocation, computing not just the actual parameters, but also the delimiters. In the previous example, we computed an invocation of "handle(;)", where the semicolon came from the evaluation of "handle(_variables)". The way to do this is to define an auxiliary macro whose template is the computed invocation (and hence not metaquoted). Evaluating the auxiliary macro then evaluates the computed invocation:

```
syntax _aux means expression to compute invocation endsyntax
_aux
```

8 Pragmatic Control

All of the following pragmatic controls are processed as intrinsic (built-in) macros; therefore their argument (which extends to the newline) is evaluated in the usual way.

8.1 Including Files

It is often useful to include some or all of the source text from a separate file, for example, the separate file may provide the macro definitions used in the main file. The command

#include ⟨filename⟩

will include the source text from the source file at this point in the current file. Included files can include other files to any depth. The argument to #include can be any expression that returns a token, but string literals are most common. Lexical parameters set in a file (Sec. 8.4) are local to that file.

8.2 Trimming Expression Sequences

Both long and unevaluated parameters match all of the text between their bounding delimiters, including leading and trailing whitespace. (Short parameters do not include leading or trailing whitespace.) Sometimes this leading or trailing whitespace is undesirable, for example when we want to concatenate with other text without any intervening whitespace. To handle such situations, synmac provides an intrinsic *trim* operation that removes leading and trailing whitespece for its (evaluated) argument. The expression

#trim ⟨expr seq⟩ **endtrim**

evaluates its argument ⟨expr seq⟩ in the normal way, and then trims off any leading or trailing whitespace. For example, this macro:

```
syntax define &axis bounds means {
  lower#trim axis endtrim{}bound = 0;
  upper#trim axis endtrim""bound = 100;
}endsyntax

define X /*ordinate*/ bounds
```

```
define Y /*abscissa*/ bounds
```

generates

```
lowerXbound = 0;
upperXbound = 100;
lowerYbound = 0;
upperYbound = 100;
```

The macro also illustrates two different ways to separate alphanumeric words.

8.3 Setting Parameters

8.3.1 Definition Keywords

The keywords used for defining syntax macros can be redefined. The default command for defining a macro has the form:

syntax ⟨pattern⟩ **means** ⟨template⟩ **endsyntax**

Entering, for example, the command:

```
#set syntax def as ;
```

will change the definition syntax to:

def ⟨pattern⟩ as ⟨template⟩;

Similarly, the default syntax for declaring constant macros (patterns) is:

pattern ⟨pattern⟩ **endpattern**

Entering, for example, the command:

```
#set pattern declare $
```

will change the pattern declaration syntax to:

declare ⟨pattern⟩ $

See the examples in Secs. 5.1 and 5.3 to see alternative definition syntax in use.

8.3.2 Parameter Flags

The token that represents the long parameter flag in patterns can be set by

#**set** long ⟨token⟩

The token that represents the short parameter flag in patterns can be set by

#**set** short ⟨token⟩

The token that represents the unevaluated parameter flag in patterns can be set by

```
#set uneval ⟨token⟩
```

In all of these cases the ⟨token⟩ can be a word or a special character. This will not affect macros already defined.

8.3.3 Newline, Endline, and Dedent Pattern Elements

The token that represents a newline delimiter in patterns can be set by

```
#set newline ⟨token⟩
```

The ⟨token⟩ can be a word or a special character. For example, the following changes the newline delimiter from the default "$" to "nl":

```
#set newline nl
```

This will not affect macros already defined. The token that represents a dedent delimiter in patterns can be set by

```
#set dedent ⟨token⟩
```

Similarly the token that represents a endline delimiter — that is, an end-of-line that is not consumed (Sec. 3.3.2) — in patterns (by default "#") can be set by

```
#set endline ⟨token⟩
```

The ⟨token⟩ can be a word or a special character. This will not affect macros already defined.

8.3.4 Commit Pattern Element

The token that represents the commit flag in patterns can be set by

```
#set commit ⟨token⟩
```

The ⟨token⟩ can be a word or a special character. This will not affect macros already defined.

8.3.5 Metaquotes

The metaquotes (by default {}) can be changed by this command:

```
#set metaquotes ⟨token⟩ ⟨token⟩
```

This sets the ⟨token⟩s (words or special characters) as the open and close metaquotes, respectively. For example,

```
#set metaquotes delay enddelay
#set metaquotes < >
```

sets the metaquotes as indicated.

8.3.6 Command flag

The token that represents the command flag (by default "#") can be set by

```
#set command ⟨token⟩
```

The ⟨token⟩ can be a word or a special character.

8.4 Lexical Commands

Lexical commands are indicated by a flag character (by default, the back slash, \) immediately after a newline. (Note that this implies that it cannot be the first character in a file.) These changes stay in effect in a file until and unless they are changed by another lexical command. Whitespace can come between the command and its argument string. The arguments of lexical commands are not evaluated, so they are always string literals.

8.4.1 Alpha characters

Sets the characters that, in addition to digits and upper and lower case letters, will be considered alphanumeric for the purpose of parsing tokens. For example, the following allows underscores in words:

```
\alpha "_"
```

The command resets the set of allowed extra characters, which can be set to none by setting to the empty string.

8.4.2 Quotes

Sets the open and close string quotes (by default ""). If two characters are provided, they become the open and close quotes; of one is provided, it becomes both. This changes the quotes to asymmetric single quotes:

```
\quotes "‘’"
```

This sets the quotes back to the default double quotes:

```
\quotes ‘"’
```

8.4.3 Superquote

Sets the superquote character (backslash by default) for including quotes in strings. This sets the superquote to percent:

```
\superquote "%"
```

8.4.4 Flag

Sets the flag character for lexical commands. For example, this changes it from the default backslash to $:

```
\flag "$"
```

8.5 Debugging

The synmac program provides a number of debugging commands, which may be useful in diagnosing problems with macro definitions.

8.5.1 Definition Dump

To turn on definition dumping enter the text:

```
#set definitions on
```

which will cause all subsequent macro definitions to print all the current definitions (user defined and intrinsic). They are printed in an internal format, but should be clear enough. By looking at these you may discover that a macro is undefined or defined differently than you intended. Definitions are printed using the current tokens representing the newline delimiter, commit flag, and long and short parameter characters. Definition dumping is turned off by setting it to anything but "on".

8.5.2 Pause

Sometimes it is difficult to understand where in the source text a problem has occurred. The command

```
#set pause ⟨string⟩
```

will print a message (the ⟨string⟩) each time the macroprocessor encounters it and wait for you to enter something before continuing.

8.5.3 Message

Sometimes it is difficult to tell where in the source text some problem is occurring. This can happen, for example, in long sequences of macro definitions that generate little if any target text.

```
#set message ⟨string⟩
```

The message text (a ⟨string⟩) is printed each time the macroprocessor encounters it.

8.5.4 Backtrack warning

Sometimes errors in macro definitions or invocations manifest themselves by the macroprocessor scanning and backtracking across large regions of the source text. Backtracking may be constrained by entering the command:

```
#set backtrack ⟨number⟩
```

where ⟨number⟩ is any number. Any attempt to backtrack over more than ⟨number⟩ tokens will trigger an informative diagnostic, which may help in identifying the problem.

8.5.5 Tracing

This command is intended primarily for use by developers.

```
#set trace ⟨word⟩
```

This causes the macroprocessor to print out trace information while it is running, some of which may be voluminous. The ⟨word⟩ may be an number, which sets the trace level to that number. Currently defined trace levels are 1 to 4. Higher levels produce more output and include the trace output at lower levels. The ⟨word⟩ can also be certain specific words to generate trace output about that topic. Currently defined topics (which correspond to internal procedures) are "evalexpr", "tryrule", "tryapply", and "match". Setting the trace level to "0" turns tracing off.

> *Idea:* Most of these tracing options will be eliminated eventually, except those useful for users.

9 Running Synmac

The current implementation of synmac is a python program that reads the source text from standard input and writes the target text on standard output. Diagnostics and debugging information go to the standard error file. It can be run interactively and the source text is terminated by the end-of-file or the word "eof". More commonly, synmac is used non-interactively to generate a target file from a source file, e.g.:

```
% synmac.py <source.smac >target
```

(We are using the extension ".smac" for synmac source text.)

> *Idea:* For the most part, synmac processes utf-8 unicode files correctly. Should it have a utf-16 option?

References

[1] B. J. MacLennan, "Path creation by continuous flocking as an example of a morphogenetic programming language," University of Tennessee Department of Electrical Engineering and Computer Science, Faculty Publications and Other Works — EECS. http://trace.tennessee.edu/utk_elecpubs/24, 2018.

[2] ——, "The Synmac syntax macroprocessor: Introduction and manual, version 5," University of Tennessee Department of Electrical Engineering and Computer Science, Faculty Publications and Other Works — EECS. http://trace.tennessee.edu/utk_elecpubs/23, 2018.

A Synmac Grammar

Because the essence of a syntax macro language is to have a variable syntax, it is difficult to write a grammar that is both informative and formally accurate. This grammar aims at being informative for users, with some sacrifice of accuracy. The grammatical notation is as follows: curly braces surround alternatives and group items; square brackets surround optional items; superscript * means zero or more repetitions; superscript + means one or more repetitions.

⟨expr seq⟩ ::= [⟨whitespace⟩] [⟨expr⟩[⟨whitespace⟩]]*

⟨expr⟩ ::= { ⟨invocation⟩ | ⟨string⟩ | **{**⟨expr seq⟩**}** | ⟨token⟩ }

⟨invocation⟩ ::= ⟨user macro⟩ | ⟨intrinsic macro⟩

⟨user macro⟩ ::= ⟨delim⟩+ [⟨actual⟩⟨delim⟩+]* [⟨expr⟩]

⟨actual⟩ ::= ⟨expr⟩ | ⟨expr seq⟩

⟨intrinsic macro⟩ ::= { ⟨definition⟩ | #**include** ⟨expr seq⟩⟨newline⟩ | #**trim** ⟨expr seq⟩ **endtrim** | #**set** ⟨parameter setting⟩⟨newline⟩ | #**endline** | ⟨newline⟩\⟨lexical command⟩ ⟨string⟩ }

⟨definition⟩ ::= { **syntax** ⟨pattern⟩ **means** ⟨template⟩ **endsyntax** | **pattern** ⟨pattern⟩ **endpattern** }

⟨pattern⟩ ::= ⟨delim⟩+ [⟨param⟩⟨delim⟩+]* [⟨short param⟩]

⟨param⟩ ::= ⟨short param⟩ | ⟨long param⟩ | ⟨uneval param⟩

⟨short param⟩ ::= ~⟨word⟩

⟨long param⟩ ::= &⟨word⟩

⟨uneval param⟩ ::= '⟨word⟩

⟨template⟩ ::= ⟨expr seq⟩

⟨delim⟩ ::= { ⟨token⟩ | $ | # | **dedent** } [!]

⟨text⟩ ::= ⟨token⟩ | ⟨string⟩

⟨token⟩ ::= ⟨word⟩ | ⟨special character⟩

⟨word⟩ ::= ⟨alphanumeric⟩+

⟨string⟩ ::= " { ⟨nonquote char⟩ | \⟨char⟩ }* "

⟨lexical command⟩ ::= **alpha** | **flag** | **quotes** | **superquote**

⟨param setting⟩ ::= {
alpha ⟨string⟩
backtrack ⟨text⟩
command ⟨token⟩
commit ⟨token⟩
dedent ⟨token⟩
definitions ⟨text⟩
endline ⟨token⟩
long ⟨token⟩
message ⟨text⟩
metaquotes ⟨token⟩ ⟨token⟩
newline ⟨token⟩
pattern ⟨delim⟩ ⟨delim⟩
pause ⟨text⟩
short ⟨token⟩
syntax ⟨delim⟩ ⟨delim⟩ ⟨delim⟩
trace ⟨text⟩
uneval ⟨token⟩
}