MetaC

David McAllester

June 12, 2018

Abstract

MetaC is a Lisp-inspired programming environment for C. MetaC provides a read-eval-print loop (REPL) capable of evaluating (compiling and running) C statements and expressions. The REPL is capable of incremental procedure definition and redefinition (dynamic linking). MetaC also provides a macro feature similar to that of Lisp and powerful symbolic programming features for writing computed macros. Most of the symbolic programming features of MetaC are implemented as bootstrapped MetaC macros. MetaC is intended to be a development environment for frameworks implemented as macro packages. Frameworks written in MetaC (and MetaC itself) use a universal algebraic concrete syntax (UACSS) — a simplification of the C concrete syntax but based on the Lisp philosophy that a single concrete syntax suffices for a wide variety of languages with a wide variety of semantics. MetaC macro packages should be viewed as compilers from framework expressions to C code. MetaC gives the framework developer complete control over the C code generated for framework expressions. Frameworks written in MetaC inherit the REPL and general programming environment of MetaC.

1 Introduction and Overview

We start with hello world.

1.1 Hello World

Once one has cloned the MetaC repository (git@github.com:mcallester/MetaC.git) one can cd to the repository directory and type

```
bash$ make MC
...
```

```
basth$ MC

MC>
```

We can then evaluate a hello world expression.

```
MC> `{hello world}

hello world

MC>
```

In the above example the backquote expression given to the REPL macro-expands to a C expression which evaluates to the expression “hello world”. Backquote is described in more detail below.

1.2 C Statements, C expressions, and UACS Expressions

We can also declare global variables and execute statements from the REPL.

```
MC> int x[10];

done

MC> for(int i = 0; i < 10; i++)x[i] = i;

done


MC> for(int i = 0; i < 10; i++)fprintf(stdout,"%d",x[i]);

0123456789done

MC>int_exp(x[5])

5

MC>
```

In C syntax one distinguishes C expressions from C statements. A C statement is executed for effect while a C expression is evaluated for value. The REPL can be given either a statement or an expression. When given a C statement, the REPL simply prints "done" after executing the statement. When given a C expression the REPL computes and prints the value. The REPL assumes that the value of a C expression is a UACS tree (also called a UACS expression giving a second, and confusing, sense of the term "expression"). The procedure `int_exp` above converts an integer to a UACS expression. UACS expressions are abstract syntax trees but can viewed as representations of strings.

If a C statement executes a `return` outside of any procedure then that value is returned to the REPL and printed. For example, the above session can be extended with the following.

```
MC>{int sum = 0; for(int i = 0; i < 10; i++)sum += x[i]; return int_exp(sum);}

45

MC>
```

1.3 Definitions of Types and Procedures

One can also define new data types and procedures from the REPL.

```
MC> typedef struct expliststruct{
      expptr car;
      struct expliststruct * cdr;
    } expliststruct, *explist;

done

MC> explist mycons(expptr x, explist l){
      explist cell = malloc(sizeof(expliststruct));
      cell->car = x;
      cell->cdr = l;
      return cell;}

done

MC> expptr list_exp(explist l){
     if(l == NULL) return NULL;
     return `{ ${l->car} ${list_exp(l->cdr)} };
}

done
```

```
MC> list_exp(mycons(‘{foo},mycons(‘{bar},NULL)))

foo bar

MC>
```

Procedures can also be redefined at the REPL provided that the signature (argument types and return type) remains the same. In the current implementation (June 5, 2018) changing a procedure signature requires restarting the MetaC REPL. This restricting ensures meaningful C type checking in the presence of dynamic linking.

1.4 The Global Variable Array Restriction

To simplify dynamaic linking, all global data variables must be arrays. One can always use single element arrays to represent non-array variables. To make this more convenient we allow single element arrays to be declared with an initial value in a manner similar to non-array data variables. For example, we can declare and assign a variable `y` as follows.

```
MC> int y[0] = 2;

done

MC> y[0] += 1;

done

MC> int_exp(y[0])

3

MC>
```

Here assignments to $y[0]$ are allowed but assignments to y are not — assignment to array variables are not allowed in C. As noted above, this restriction greatly simplifies dynamic linking of data variables.

1.5 The Single Symbol Type Restriction

To simplify the implementation of MetaC all type expressions appearing in procedure signatures and global array declarations must be single symbols. Arbitrary types can be given single symbol names using `typedef`.

1.6 Backquote and UACS Expression Trees

We now consider backquote and UACS expressions in more detail. Backquote can be used in a manner analogous to string formatting.

```
MC> expptr friend[0] = ‘{Bob Givan};

done

MC> int height[0] = 6;

done

MC> ‘{My friend ${friend[0]} is ${int_exp(height[0])} feet tall.}

My friend Bob Givan is 6 feet tall.

MC>
```

While UACS expressions can be viewed as representations of strings, UACS expressions are actually abstract syntax trees. The tree structure plays an important role in pattern matching. The tree structure is more apparent in the following example.

```
MC> expptr x[0] = ‘{a+b};

done

MC> ‘{bar(${x[0]})}

bar(a+b)

MC>
```

As described in the next section, the pattern `foo(!x)` will match the expression `foo(a+b)` with `x` bound to the expression (tree) `a+b`.

1.7 Pattern Matching

MetaC provides a pattern matching case statement. The general form of the case construct is the following.

```
ucase{e;
  {<pattern1>}:{<body1>}
   ...
  {<patternn>}:{<bodyn>}}
```

Variables are marked in patterns by the symbol tags `!` and `?`. The variable `!x` can match any expression while `?x` can only match symbols (or other lexicalized tokens without internal structure as described in section 2.1).

```
MC> int value(expptr e){
      ucase{e;
        {!x+!y}:{return value(x)+value(y);}
        {!x*!y}:{return value(x)*value(y);}
        {(!x)}:{return value(x);}
        {!z}:{return symbol_int(z);}}
     return 0;
   }

done

MC>value(`{3+(2*5)})

30

MC>
```

In many cases the choice of the particular tree structure imposed by the MetaC reader is unimportant. For example, the expression `a+b+c` will match the pattern `!x+!y+!z` independent of whether + is left associative or right associative. But the tree structure does matter in other cases. The pattern `!x*!y` will not match the expression `a+b*c` because the reader brackets `a+b*c` as $\langle \mathtt{a} + \langle \mathtt{b} * \mathtt{c} \rangle\rangle$. The pattern `!x*!y` does match `(a+b)*c`.

As of June 2018 repeated variables, such as in the pattern {`!x + !x`}, are not supported (and not checked for). If a variable is repeated it will be matched at its last occurrences under depth-first left to right traversal.

1.8 Gensym and Macro Definitions

MetaC supports computed macros which can exploit pattern matching, backquote pattern instantiation, and other (bootstrapped) high level language features. As a very simple example we can define a `dolist` macro at the REPL.

```
MC> umacro{mydolist(!x, !L){!body}}{
     expptr rest = gensym(`{rest});
     return `{for(explist ${rest} = ${L};
                  ${rest} != NULL;
                  ${rest} = ${rest}->cdr;)
               {expptr ${x} = ${rest}->car; ${body}}}}

done

MC>macroexpand(`{dolist(item,list){f(item);}})

for(explist _mcgen_rest1=list; _mcgen_rest1 !=NULL; _mcgen_rest1=_mcgen_rest1->cdr;)
    {expptr item=_mcgen_rest1->car;
     f(item);}

MC>
```

The general form of a macro definition is

```
umacro{<pattern>}{<body>}
```

where instances of `<pattern>` in MetaC code are to be replaced by the value returned by `<body>` under the variable bindings determined by the match. The procedure `macroexpand` takes a UACS expression and returns the result of repeatedly expanding outermost macros until no further macro expansion is possible. Macro expansion can generate effects as well as return an expansion. The REPL performs macro expansion on the given expression and also performs the effects of that expansion. `umacro` is itself a MetaC macro and we can feed the above macro definition to the REPL.

The macro expansion of the above macro definition defines a procedure to compute the macro expansion and installs that procedure on a macro property of the symbol `mydolist`. In a typical use a macro patterns is a symbol followed by a sequence of parenthesis expressions., in which case the expansion procedure is attached to the given symbol, or generalized application expression or a binary connection expressions. UACS expression types are discussed in the next section. The procedure for macro expansion is attached to either the head symbol of the application or the binary operator of the binary connection expression.

2 Universal Algebraic Concrete Syntax (UACS)

It is not obvious how to implement light weight expression quotation and expression pattern matching for C expressions. The syntax of C is complex. We bypass this complexity by introducing universal algebraic concrete syntax (UACS). UACS is "universal" in that it has the following three properties.

1. UACS syntax trees are semantics-free. They are simply trees viewed as representations of character strings.
2. The UACS reader can read any parenthesis-balanced character string. Here parenthesis-balanced means that every open parenthesis, brace or bracket has a matching closing character and that string quotations are properly closed.
3. The printer inverts the reader. If we read a string s into a UACS expression e and then print e back into a string s' we have that s and s' are "whitespace equivalent" as defined in section 2.1. This implies that s and s' will be treated equivalently by the C compiler.

The emphasis here is on the representation of strings. The reader does not always invert the printer — the expression ⟨⟨`one two`⟩ `three`⟩ prints as `one two three` which reads as ⟨`one` ⟨`two three`⟩⟩. But the represented string is preserved. This is fundamentally different from most programming languages supporting symbolic computation (such as Lisp) where it is assumed that the reader should invert the printer.

The above universality properties allow one to assign semantics to UACS expressions based on the strings that they represent. We can assign C semantics to an expression by printing the expression and passing the resulting string to a C compiler. This string-based semantics is not always compositional with respect to the UACS bracketing. However, the tree structure imposed by the reader is designed to approximate the compositional structure of C syntax. In most cases pattern matching on UACS expressions recovers substructure that is semantically meaningful under C semantics. Parentheses and semicolons are particularly helpful in aligning UACS trees with C syntax.

2.1 Comment Removal, Segmentation and Lexicalization

The MetaC preprocessor removes C-style comments and divides files into segments where each segment is to be read as a separate expression. A new segment starts at the beginning of any line whose first character is not a space, tab or close character (right parenthesis, brace or bracket). A file must be parenthesis-balanced within each segment. For the REPL a segment ends at the first return character not occurring inside parentheses.

Lexicalization segments a pre-processed into to a sequence of character strings. The UACS lexer preserves all non-white characters. For UACS lexer the strings in lexicalized sequence are classified into the following types.

- Symbols, A symbol is a character string consisting of only alpha-numeric characters — upper and lower case letters of the alphabet, plus the decimal numerals, plus underbar. For example `foo_bar1`.
- Connectives. A connective is a character string consisting of only "connective characters" — the characters comma, colon, semicolon, `=`, ∼, <, >, `|`, `&`, `+`, `-`, `*`, `/`, `.`, `@`, ^. For example the strings `->` and `==`.
- Quoted strings. A character string starting and ending with the same string quotation character. For example `"foo bar"`, `"!:&?;"` or `'a'`.
- Misc characters. These are non-white single characters other than those mentioned above and other than parenthesis characters. In particular, one of the characters `$`, \\, `` ` ``, `!`, `?`, `#` and `%`.

The UACS lexicalization preserves all quoted character strings and all non-white characters outside of string quotation. Two strings are called whitespace-equivalent if they lexicalize to the same sequence. For many strings s we have that s' is whitespace equivalent to s if and only if s and s' have the same sequence of non-white characters. Two strings that lexicalize equivalently under the UACS lexer will also lexicalize equivalently under the C lexer although the C lexicalization may be different from that of the UACS lexicalization. Ultimately an unlexicalized character string is passed from MetaC to the C compiler. The MetaC programmer should keep in mind the character strings that UACS expressions represent.

2.2 UACS Expressions: cons-car-cdr

A UACS expression is either an atom (a lexical item character string), a pair $\langle e_1 e_2 \rangle$ where e_1 and e_2 are expressions, or a "parenthesis expression" the form (e), $\{e\}$, or $[e]$ where e is an expression. A cons-car-dcr interface to expressions can be implemented as follows.

```
exptr cons(expptr x, expptr y){return `{${x} ${y}};}

expptr car(expptr cell){
  ucase{cell;
         {!x !y}:{return x;}}}

expptr cdr(expptr cell){
  ucase{cell;
         {!x !y}:{return y;}}}

expptr add_parens(expptr e){return `{(${e})};}

expptr inside(expptr parens){
  ucase(parens;
        {(!e)}:{return e;}}}

int atomp(expptr s){
    ucase{s;
          {?s}:{return 1;}
          {!x}:{return 0;}}}
```

As part of this interface we can convert an atom (or any expression) to a C string with the MetaC procedure `exp_string`. The procedure `exp_atring` is a version of the MetaC printer.

2.3 UACS Expressions: Phantom Brackets

A UACS expression can be represented by "phantom brackets" where there is a pair of brackets for each pair expression (cons cell). For example the expression {`one two three`} reads as {⟨`one` ⟨`two three`⟩⟩}. To reduce the clutter of brackets we will adopt the Lisp convention of not showing all the cell brackets for right-associative sequences (lists). For example ⟨`zero` ⟨`one` ⟨`two three`⟩⟩⟩. will be written as ⟨`zero one two three`⟩. Note that these "lists" are atom-terminated rather than the Lisp convention of nil termination. We will also often write $\langle\langle a_1\ o\rangle\ a_2\rangle$ where o is a connective atom as $\langle a_1\ o\ a_2\rangle$. For example, the expression {`a + b`} reads as {⟨⟨`a +`⟩ `b`⟩} which is then abbreviated as {⟨`a + b`⟩}. Left-association for binary connectives gives a C-consistent treatment of semicolon as a binary connective while supporting expressions of the form $\langle e\ ;\rangle$. It will generally be clear from context which abbreviation is being used. A set of examples of strings and the phantom brackets generated by the reader is given in figure 1. The printer simply removes the phantom brackets and prints the string that the expression represents.

To emphasize the significance of left-association for binary connectives we note that the UACS reader bracketing of

$$\{e_1\ ;\ e_2\ ;\ e_3\ ;\}$$

can be written using either the binary connective convention as

$$\{\langle e_1\ ;\ \langle e_2\ ;\ \langle e_3\ ;\rangle\rangle\rangle\}$$

$$\begin{aligned}
\texttt{Hello World} &\Rightarrow \langle\texttt{Hello World}\rangle\\
\texttt{one two three} &\Rightarrow \langle\texttt{one}\ \langle\texttt{two three}\rangle\rangle\\
&= \langle\texttt{one two three}\rangle\\
\texttt{x} + \texttt{y} &\Rightarrow \langle\langle\texttt{x} +\rangle\ \texttt{y}\rangle\\
&= \langle\texttt{x} + \texttt{y}\rangle\\
\texttt{x} + \texttt{y} * \texttt{z} &\Rightarrow \langle\texttt{x} + \langle\texttt{y} * \texttt{z}\rangle\rangle\\
(\texttt{x} + \texttt{y}) * \texttt{z} &\Rightarrow \langle(\langle\texttt{x} + \texttt{y}\rangle) * \texttt{z}\rangle\\
\texttt{foo(int x)} &\Rightarrow \langle\texttt{foo}\ (\langle\texttt{int x}\rangle)\rangle\\
\texttt{foo(int x, float y)} &\Rightarrow \langle\texttt{foo}\ (\langle\langle\langle\texttt{int x}\rangle\ ,\rangle\ \langle\texttt{float y}\rangle\rangle)\ \rangle\\
?\texttt{f}(!\texttt{args}) &\Rightarrow \langle\ \langle?\ \texttt{f}\rangle\ (\langle!\ \texttt{args}\rangle)\ \rangle\\
\backslash\$\{\texttt{x}\} &\Rightarrow \langle\backslash\ \langle\$\ \{\texttt{x}\}\rangle\rangle
\end{aligned}$$

Figure 1: **Examples of Reader Bracketings.** Bracketings are shown for the expression that results form reading the given strings. A complete bracketing shows a pair of brackets for every expression pair (cons cell). The second and third example show two somewhat informal conventions for dropping some of the brackets — general sequences are assumed to be right-associative and binary connective applications are assumed to be left-associative. These conventions are used in other examples. Expressions are printed without the brackets — the brackets are "phantoms" that show the tree structure. The bracketing (parsing) done by the reader is specified formally in section 2.5. Since the bracketing does not affect the represented string, the precise bracketing is often unimportant.

or the right-associative sequence convention as

$$\{\langle\langle e_1\ ;\rangle\ \langle e_2\ ;\rangle\ \langle e_3\ ;\rangle\rangle\},$$

both of which abbreviate the same full bracketing

$$\{\langle\langle e_1\ ;\rangle\ \langle\langle e_2\ ;\rangle\ \langle e_3\ ;\rangle\rangle\rangle\}.$$

Hence the string `(a ; b ;)` matches both the pattern the pattern $\{(!x\ !y)\}$ with x bound to $\langle a\ ;\rangle$ and y bound to $\langle \texttt{b}\ ;\rangle$, and also matched $\{(!x\ ;\ !y)\}$ with x bound to `a` and y bound to $\langle\texttt{b}\ ;\rangle$.

2.4 Backquote and Pattern Matching Revisited

The semantics of backquote is defined in terms of cons-car-cdr view of UACS expressions independent of the UACS reader. In the typical case, the C value of a backquote expression $\langle`\ \{e\}\rangle$ is the expression e with subexpressions of the form $\langle\$\ \{w\}\rangle$ replaced by the expression which is the C value of the string represented by the expression w.

Unfortunately this evaluation rule for backquote expressions is incomplete. One of the most confusing situations is where the expansion of a macro contains a backquote. Writing such a macro typically involves nested backquotes. While nested backquotes are confusing, and should be avoided when possible, MetaC supports nested backquotes. We consider a series of backquote expressions each of which evaluates to the previous one.

First we have

$$`\{a + b + \$\{z\}\} \tag{1}$$

If the value of variable `z` is the expression `c` then the value of expression (1) is the expression `a+b+c`. The symbol `$` can be included in the value of a backquote expression by quoting it. This gives our second expression.

$$`\{`\{a + \$\{y\} + \backslash\$\{z\}\}\} \tag{2}$$

If the value of variable `y` is the expression b then the value expression of (2) is expression (1). We can even have multiple layers of quotation as in the following.

$$`\{`\{`\{\$\{x\} + \backslash\$\{y\} + \backslash\backslash\$\{z\}\}\}\} \tag{3}$$

If the value of variable `x` is the expression `a` then the value of expression (3) is expression (2).

As with backquote, pattern matching is defined in terms of the cons-car-cdr view of expressions independent of the UACS reader. We define a substitution to be a mapping from symbol atoms to expressions. For a substitution σ and a pattern $\{e\}$ we define the expressions $\sigma(e)$ to be the result of replacing each subexpression of e of the form $\langle !\ x\rangle$ where x is a symbol atom with $\sigma(x)$ and each subexpression of the form $\langle ?\ x\rangle$ where x is a symbol atom by $\sigma(x)$ provided that $\sigma(x)$ is an atom. A pattern $\{e\}$ matches an expression w with substitution σ if $\sigma(e) = w$. An exception to this is the case where a variable occurs multiple times in the pattern. As of June 2018 multiple occurrences of a variable in a pattern is not supported.

2.5 The Reader Specification

We now give a grammar specifying the UACS reader. We also attempt to describe the reader in an intelligible way independent of the formal grammar. The grammar will define a linear-time deterministic shift-reduce parsing process.

To read a string s it will be convenient to enclose s in parenthesis and read the string (s). This gives the reader access to "endpoint tokens" marking the beginning and end of the string. Reading (s) results in a parenthesis expression (e) where the string s is read as the expression e.

The grammar uses nonterminals SYM, CONN, QUOTE and MISC to range over symbol atoms (alphanumeric strings), binary connective atoms (strings over the connective symbols), quoted string atoms, and the miscellaneous character atoms respectively. There is also a nonterminal PAREN to range over parenthesis expressions.

The UACS reader has the property that a string of the form `${...}` always reads to a single expression of the form $\langle\$\ \{e\}\rangle$ independent of the context in which the string appears. The general rule is that any sequence of the miscellaneous characters followed by either a symbol atom, a string quotation atom, or a parenthesis expression reads as a single expression independent of context. A sequence of miscellaneous characters followed a close character or a connective atom also reads as a single expression. The reader grammar has a nonterminal GATOM, for generalized atom, ranging over this class of expressions. This definition of the range of the nonterminal GATOM can be expressed by the following context-sensitive grammar rules where the close character can be interpreted as any of the three parenthesis expression closing characters.

$$\begin{array}{rcl} \text{GATOM} & \rightarrow & \text{SYM} \mid \text{QUOTE} \mid \text{PAREN} \mid \text{MISC GATOM} \\ \text{GATOM)} & \rightarrow & \text{MISC)} \\ \text{GATOM CONN} & \rightarrow & \text{MISC CONN} \end{array}$$

Note that any occurrence of a miscellaneous character is contained in some GATOM expression.

There is also has a nonterminal ARG that ranges over sequences of GATOM's that are followed by a binary connective or a closed parenthesis character. This is defined by the following grammar.

$$\begin{array}{rcl} \text{ARG)} & \rightarrow & \text{GATOM)} \\ \text{ARG CONN} & \rightarrow & \text{GATOM CONN} \\ \text{ARG} & \rightarrow & \text{GATOM ARG} \end{array}$$

The string `(!x !y)` is now deterministically parsed as (GATOM GATOM) and then as (ARG). The string $(f(x)+!y)$ is now parsed as (GATOM GATOM + GATOM) and then as (ARG + ARG).

We now consider the problem of parsing a sequence of arguments and connectives. We first consider the normal case where the sequence alternates arguments and connectives and starts and ends with arguments. The more general case is considered below (we are requiring that we be able to parse any string).

Each connective atom is associated with a precedence and each precedence level is associated with a convention for being left-associative or right associative.

Grouped by precedence, the binary connective characters are semicolon, comma, {=, ~, <, >}, {|, &}, {+, -}, {*, /}, ., @, ˆ and colon. The precedence of a binary connective is determined by its first character. The characters have low to high (outer to inner) precedence in the order given with symbols in the same group having the same precedence. Note that semicolon has the lowest precedence giving it syntactic power similar to that of parentheses in separating text. Also note that colon has the highest precedence which allows it to function as a "package connective" between symbols.

To be able to parse any string we intuitively add a phantom argument between each pair of adjacent binary connectives and also add a phantom argument at the beginning if the string starts with a connective and a phantom argument at the end if the string ends with a connective. This determines a parse tree which may contain phantom arguments. The phantom argument are then removed by replacing any cell with a phantom argument by the non-phantom sibling. This gives the following readings.

$$\begin{array}{rcl}
(\mathtt{x}\ +\ \ *\ \mathtt{y}) & \Rightarrow & (\langle \mathtt{x}\ \mathtt{+}\ \langle *\ \mathtt{y}\rangle\rangle) \\
(\mathtt{x}\ *\ \ +\ \mathtt{y}) & \Rightarrow & (\langle\langle \mathtt{x}\ \mathtt{*}\rangle\ +\ \mathtt{y}\rangle) \\
(+\ \mathtt{a}\ +\ \mathtt{b}) & \Rightarrow & (\langle +\ \langle \mathtt{a}\ +\mathtt{b}\rangle\rangle) \\
(\mathtt{a}\ +\ \mathtt{b}\ +) & \Rightarrow & (\langle \mathtt{a} +\langle \mathtt{b}\ +\rangle\rangle) \\
(\mathtt{a}\ ;\ \mathtt{b}\ ;) & \Rightarrow & (\langle \mathtt{a}\ ;\ \langle \mathtt{b}\ ;\rangle\rangle) \\
 & = & (\langle\langle \mathtt{a}\ ;\rangle\ \langle \mathtt{b}\ ;\rangle\rangle)
\end{array}$$

Parsing sequences of arguments and connectives with precedence and associativity conventions should be familiar. However, for completeness, and to handle subtleties of phantom arguments, We give grammar rules. We will let CONN_p be a nonterminal ranging over connectives with precedence p and let E_p be a nonterminal ranging over expressions containing connectives where p is the lowest precedence of any connective in the expression. We let $\tilde{\mathrm{E}}_p$ be a nonterminal ranging over expressions of the form $\langle e\ o\rangle$ with $\mathrm{E}_p \rightarrow^* e$ and o is a connective with precedence q where $q > p$ or $q = p$ with p being a left associative precedence. The grammar rules for the case where all arguments are explicit can be written as

$$\begin{array}{rcll}
(\ \tilde{\mathrm{E}}_p & \rightarrow & (\ \mathrm{ARG}\ \mathrm{CONN}_p & \\
\tilde{\mathrm{E}}_p\ \tilde{\mathrm{E}}_q & \rightarrow & \tilde{\mathrm{E}}_p\ \mathrm{ARG}\ \mathrm{CONN}_q & \text{for } q > p \text{ or } q = p \text{ and } p \text{ right associative} \\
\mathrm{E}_p\ \mathrm{CONN}_q & \rightarrow & \tilde{\mathrm{E}}_p\ \mathrm{ARG}\ \mathrm{CONN}_q & \text{for } q < p \text{ or } q = p \text{ and } p \text{ left associative} \\
\mathrm{E}_p\ ) & \rightarrow & \tilde{\mathrm{E}}_p\ \mathrm{ARG}) &
\end{array}$$

Phantom arguments are handled by the following rules which also define a deterministic parsing algorithm.

$$\begin{array}{rcll} (\ \tilde{\mathrm{E}}_p & \rightarrow & (\ \mathrm{CONN}_p & \\ \tilde{\mathrm{E}}_p\ \tilde{\mathrm{E}}_q & \rightarrow & \tilde{\mathrm{E}}_p\ \mathrm{CONN}_q & \text{for } q > p \text{ or } q = p \text{ and } p \text{ right associative} \\ \mathrm{E}_p\ \mathrm{CONN}_q & \rightarrow & \tilde{\mathrm{E}}_p\ \mathrm{CONN}_q & \text{for } q < p \text{ or } q = p \text{ and } p \text{ left associative} \\ \mathrm{E}_p\ ) & \rightarrow & \tilde{\mathrm{E}}_p) & \end{array}$$

Finally we have

$$\mathrm{PAREN} \Rightarrow (\mathrm{E}_p)\ \mid\ (\mathrm{ARG})\ \mid\ ()$$

where analogous rules hold for the other parethesis characters as well. The empty parenthesis expression contains the atom for the empty string. In the implementation the empty string is denoted by `nil` which is also used as a general list terminator.

The system of rules presented in this section define a linear time deterministic shift-reduce parsing process.