MetaC

David McAllester

June 20, 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 syntax" — 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
...

bash$ 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 Universal Syntax

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 universal syntax tree (also called an expression, giving a second, and confusing, sense of the term "expression"). The procedure `int_exp` above converts an integer to a universal syntax expression. Universal syntax 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 myexpstruct{
   char * label;
   struct myexpstruct * car;
   struct myexpstruct * cdr;} myexpstruct, *myexp;

done

MC> myexp mycons(char*s,myexp x,myexp y){
    myexp cell=malloc(sizeof(myexpstruct));
    cell->label=s;
    cell->car=x;
    cell->cdr=y;
    return cell;}

done

MC> expptr myexp_exp(myexp x){
   if(x==NULL)return string_atom("nil");
```

```
   return
    `{${string_atom(x->label)}
      ${myexp_exp(x->car)}
      ${myexp_exp(x->cdr)}};
}

done

MC> myexp_exp(mycons("foo",mycons("bar",NULL,NULL),NULL))


foo bar nil nil nil
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 Universal Syntax Expressions

We now consider backquote and universal syntax 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 universal syntax expressions can be viewed as representations of strings, universal syntax 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 e[0] = ‘{a+b};

done

MC> ‘{bar(${e[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 construct `ucase` (universal case). The general form of the case construct is

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

Variables are marked in patterns by the prefix $.

```
MC>int numeralp(expptr x)
    {if(!atomp(x))return 0;
    char*s=atom_string(x);
    for(int i=0;s[i]!='\0';i++){if(s[i]<'0'||s[i]>'9')return 0;}
    return 1;
}

done

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}.(numeralp(z)):{return atoi(atom_string(z));}
    }
  return 0;
  }

done

MC>int_exp(value(`{5+2*10}))

25

MC>int_exp(value(`{foo}))
```

```
 match error: the value


foo
does not match any of


{$z}.(numeralp(z)){($x)}{$x*$y}{$x+$y}

MC>
```

In many situations the choice of the particular tree structure imposed by the MetaC reader is not important as long as the same conventions are used in both the patterns and the expressions they are matching. 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 $\langle$`a` +$\langle$`b` $*$ `c`$\rangle\rangle$. The pattern `$x*$y` does match `(a+b)*c`.

When a `ucase` is executed the patterns are tries sequentially and stops after the first match. If no pattern matches an error is generated. One can always add a final pattern of $\{\$x\}$ to provide a default catch-all case if that is desired.

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, back-quote 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=$L; cellp($rest);$rest=cdr($rest);){
      expptr$x=car($rest);
      $body}};
}
```

```
done

MC>macroexpand(‘{mydolist(item,list){f(item);}})

for
  (explist _genrest33=list;
   cellp(_genrest33);
  _genrest33=cdr(_genrest33);
  ){expptr item=car(_genrest33);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 universal syntax 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. Universal syntax 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 Syntax

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 a syntax with the following three “universal” properties.

1. Universal syntax trees are semantics-free. They are simply trees viewed

as representations of character strings.

2. The universal syntax 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 universal syntax expression e and then print e back into a string s' we have that s and s' are equivalent in a strong sense. Here we want s and s' to be "whitespace equivalent" so as to be treated equivalently by the C lexer.

The emphasis here is on the representation of strings. The reader does not always invert the printer — the expression $\langle\langle$`one two`$\rangle$ `three`$\rangle$ prints as `one two three` which reads as $\langle$`one` $\langle$`two three`$\rangle\rangle$. 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 tree structure, rather than the represented string, is fundamental and hence that the reader should invert the printer.

The above universality properties allow one to assign semantics to universal syntax 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 universal syntax 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 universal syntax expressions recovers substructure that is semantically meaningful under C semantics. Parentheses and semicolons are particularly helpful in aligning universal syntax trees with C syntax.

2.1 Universal Expressions: cons-car-cdr

A universal syntax expression is either an atom (a wrapper around a 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. All three of these datatypes have the same C type `expptr` (Expression Pointer). Each has a constructor procedure, a predicate, and accessor functions. The constructor, predicate and accessors for atoms, pairs and parenthesis expressions are the following. The atom data type has the constructor, test and accessor function for atoms are the following.

```
expptr string_atom(char *);
int atomp(expptr);
```

```
char * atom_string(expptr); //the argument must be an atom.

expptr cons(expptr,expptr);
int cellp(expptr);
expptr car(expptr); //the argument must be a cell. returns the first component.
expptr cdr(expptr); //the argument must be a cell. returns the second component.

expptr intern_paren(char,expptr); // the char must be one of '(', '{' or '['
int parenp(expptr);
expptr paren_inside(expptr);
```

2.2 Universal Expressions: Phantom Brackets

The MetaC reader maps a character string to a universal syntax expression. A specification of the reader is given in section 2.3. The expression produced by the reader can be represented by "phantom brackets" around the given string where there is a pair of brackets for each cons cell. For example the expression {`one two three`} reads as {⟨`one` ⟨`two three`⟩⟩}. Examples of strings and the phantom bracking imposed by the reading those strings is given in figure ref-fig:reader. The printer simply removes the phantom brackets and prints the string that the expression represents.

To reduce the clutter of brackets we will adopt two conventions for supressing brackets when exhibiting phantom brackets. The first convention is the Lisp right-branching convention of not showing all the cell brackets for right-associative (right-branching) sequences. 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.

The second convention is that we will sometimes write ⟨⟨a_1 o⟩ a_2⟩ where o is a connective atom as the left-branching structure ⟨a_1 o a_2⟩. For example, the expression {`a + b`} reads as {⟨⟨`a` +⟩ `b`⟩} which is then abbreviated as {⟨`a + b`⟩}. Left-branching for binary connectives gives a C-consistent treatment of semicolon as a binary connective while also supporting the interpretation of semicolon as a statement terminator. This is discussed in more detail below.

It will generally be clear from context whether we using the Lisp right-branching list convention or the left-branching connective convention. A set of examples of strings and the phantom brackets generated by the reader is given in figure 1.

To emphasize the significance of left-branching binary connectives we note that the MetaC reader bracketing of

$$\{e_1 \ ; \ e_2 \ ; \ e_3 \ ; \ e_4\}$$

$$
\begin{array}{rcl}
\texttt{Hello World} & \Rightarrow & \langle \texttt{Hello World} \rangle \\
\texttt{one two three} & \Rightarrow & \langle \texttt{one } \langle \texttt{two three} \rangle \rangle \\
 & = & \langle \texttt{one two three} \rangle \\
\texttt{x + y} & \Rightarrow & \langle \langle \texttt{x +} \rangle \texttt{ y} \rangle \\
 & = & \langle \texttt{x + y} \rangle \\
\texttt{x+y*z} & \Rightarrow & \langle \texttt{x +} \langle \texttt{y * z} \rangle \rangle \\
\texttt{(x+y)*z} & \Rightarrow & \langle (\langle \texttt{x + y} \rangle) \texttt{ * z} \rangle \\
\texttt{foo(int x)} & \Rightarrow & \langle \texttt{foo } (\langle \texttt{int x} \rangle) \rangle \\
\texttt{foo(int x, float y)} & \Rightarrow & \langle \texttt{foo } (\langle \langle \texttt{int x} \rangle \texttt{ , } \langle \texttt{float y} \rangle \rangle) \rangle
\end{array}
$$

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 2.3. Since the bracketing does not affect the represented string, the precise bracketing is often unimportant.

can be written using either the left-braching binary connective convention as

$$\{\langle e_1 \ ; \ \langle e_2 \ ; \ \langle e_3 \ ; \ e_4 \rangle \rangle \rangle\}$$

or the right-braching sequence convention as

$$\{\langle \langle e_1 \ ; \rangle \ \langle e_2 \ ; \rangle \ \langle e_3 \ ; \rangle \ e_4 \rangle\},$$

both of which abbreviate the same full bracketing

$$\{\langle \langle e_1 \ ; \rangle \ \langle \langle e_2 \ ; \rangle \ \langle \langle e_3 \ ; \rangle \ e_4 \rangle \rangle \rangle\}.$$

2.3 The Reader Specification

The MetaC reader can be described in three phases — preprocessing, lexicalization, and parsing.

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 character sting into to a sequence of atoms. The MetaC lexer preserves all non-white characters. For the MetaC lexer each atom is one of the following.

- 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 connective characters consist of all non-alphanumeric, non-white, non-parenthesis, characters other than the three special characters `$`, ‘ and \.
- Quoted strings. A character string starting and ending with the same string quotation character. For example `"foo bar"`, `"!:&?;"` or `'a'`.
- Single character atoms for parenthesis characters and the three special characters ‘, `$`, and \.

Two strings will be called whitespace-equivalent if they lexicalize to the same sequence of atoms. Two strings that lexicalize equivalently under the MetaC lexer will also lexicalize equivalently under the C lexer.

The reader is specified by a the grammar shown in figure 2. To simplify the specification of the reader we enclose the given string in parentheses so that, without loss of generality, the input is assumed to be of the form $\{s\}$ where s is the lexical sequence to be read. The grammar has a nonterminal P for parenthesis expression which we take as the top level nonterminal. The nonterminals SYM and QUOTE range over alpha-numeric string atoms and quoted string atoms respectively. For each positive integer p we have a nonterminal CONN_p ranging over connective atoms of precedence p. We have a nonterminal E_p for expressions formed with connectives of precedence p. We also has a nonterminal E_∞ for expressions that are formed at higher precedence than any connective expression. The precedence levels are divided into a set $\mathcal{L}$ of left-associative precedence levels and a set $\mathcal{R}$ of right associative precedence levels. There is a special precedence level 3 for combining expressions with the "null connective". The null connective is left-associative. The null connective is higher precedence that semicolon and comma but lower precedence than all other connectives. The nonterminal E_∞ includes an epsilon production allowing null arguments.

The nonterminal S ranges over symbols or variables where V ranges over variables. Variables are handles specially in pattern matching (`ucase`) and pattern instantiation (backquote). Expression of the form $\langle S\ P^* \rangle$ include `foo`, `foo(x)`, `foo(int x){return x;}`, `$x`, `$f(x)`, and `${f(x)}`.

$$
\begin{array}{rcl}
P & ::= & (E_p) \mid \{E_p\} \mid [E_p] \\
E_p & ::= & \langle\langle E_\ell \ \mathrm{CONN}_p\rangle \ E_r\rangle \quad p \in \mathcal{R},\ \ell > p,\ r \geq p \\
E_p & ::= & \langle\langle E_\ell \ \mathrm{CONN}_p\rangle \ E_r\rangle \quad p \in \mathcal{L},\ \ell \geq p,\ r > p \\
E_3 & ::= & \langle E_\ell \ E_r\rangle \quad \ell > 3,\ r \geq 3 \\
E_\infty & ::= & \mathrm{QUOTE} \mid \langle S \ \ P^*\rangle \mid \langle \text{`} \ P\rangle \mid P \mid \epsilon \mid \mathrm{JUNK} \\
S & ::= & \mathrm{SYM} \mid V \\
V & ::= & \langle \$ \ \mathrm{SYM}\rangle \mid \langle \$ \ P\rangle \mid \langle \backslash \ V\rangle \\
P^* & ::= & \epsilon \mid \langle P \ P^*\rangle \\
\mathrm{JUNK} & ::= & \text{`} \mid \mathrm{SJUNK} \\
\mathrm{SJUNK} & ::= & \$ \mid \backslash \mid \langle \backslash \ \mathrm{JUNK}\rangle
\end{array}
$$

Figure 2: **The grammar defining the MetaC reader.** The input string is assumed to be of the form (s) so that endpoints are explicitly labeled. The non-terminal SYM generates non-empty alpha-numeric strings; CONN_p generates non-empty connective strings of precedence p; and QUOTE generates string quotations. $\mathcal{R}$ is a set of right-associative precedence levels and $\mathcal{L}$ is a set of left-associative levels. Only the highest precedence level is left-associative. MetaC classifies all printable ASCII characters as being either alpha-numeric, connective, string quotations, parenthesis characters or one of the three special characters `, $ or \. The reader will accept any parenthesis-balanced string. ϵ denotes the empty string. Generated cells of the form $\langle w \ \epsilon\rangle$ or $\langle \epsilon \ w\rangle$ are replaced by w. Although the grammar is ambiguous, the reader is deterministic as specified by the constraints that E_∞ expressions must be maximal and that the use of the ϵ production for E_∞ must be minimized. See the text for details.

The above grammar is ambiguous. For example the string {`foo (a) (b)`} can be parsed as either the left-branching structure {⟨⟨SYM P^*⟩ P⟩} or the right-branching structure {⟨SYM P^*⟩}. The reader is of course deterministic — {`foo (a) (b)`} is read as right-branching. While a deterministic grammar for the reader can be given, it is simpler to specify a deterministic parsing process for the above ambiguous grammar. The implementation runs a deterministic left-to-right shift-reduce process. However, the parser is easier to specify as operating in global stages. In the first stage one identifies all maximal E_∞ substrings. The maximal E_∞ requirement specifies that {`foo (a) (b)`} is parsed as a single (right-branching) E_∞ expression. However the string {$ `foo (a) (b)`} is read as the longer E_∞ expression {⟨⟨! `foo`⟩ ⟨(`a`) (`b`)⟩⟩. The grammar does have the property that any given string has a unique segmentation into maximal E_∞ expressions each of which has a unique structure.

After identifying maximal E_∞ expressions we are left with a sequence of arguments (the E_∞ expressions) and connectives. However, it is possible that this sequence contains multiple consecutive connectives or multiple consecutive arguments. We then place an argument between any two consecutive connectives using the ϵ production for E_∞ and also add an ϵ term at the beginning of the sequence if the sequence starts with a connective and an epsilon argument at the end if the sequence ends in a connective. We now have a sequence of arguments and connectives starting and ending with arguments and where there are no two consecutive connectives. Next set k to be the largest precedence of any connective and form all the E_k substrings. We then repeatedly decrement k and identify all the E_k substrings until we have identifies all E_p substrings for all p. At this point the entire string must be parsed as E_p where p is the smallest precedence of the connectives where we think of adjacent arguments as having an implicit precedence 3 connetive between them.

This process can parse any parenthesis-balanced string.

The above specification of the reader can result in cells of the form ⟨e ϵ⟩ or ⟨ϵ e⟩. The implementation avoids producing such cells and the reader can be viewed as replacing ⟨e ϵ⟩ or ⟨ϵ e⟩ by e so that all cells are of the form ⟨e_1 e_2⟩ where e_1 and e_2 are expressions representing non-empty strings. An empty parenthesis string `()` is read as a parenthesis expression containing an atom for the empty string.

Grouped by precedence, the binary connective characters are

{`;`} {`,`} {ϵ} {`|`} {`&`, `?`, `!`} {`=` $\sim$, $<$, $>$} {`+`, `-`} {`*`, `/`} {`^`, `%`, `.`, `:`, `@`, `#`}.

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. Here ϵ represents the null connective at precedence level 3. The precedence can be summarized in outer to inner order as semicolon, comma, the null connective, Two levels of Boolean connectives, binary predicates, two levels of binary functions, and

innermost connectives which can be viewed as providing a way of building structured atoms. Below the level of parentheses, semicolon and comma the universal syntax precedence conventions have various divergences with C syntax. However, when writing a macro package in MetaC one can simply adopt the MetaC precedence conventions for the source code and use parentheses where necessary in the generated C code.

2.4 Backquote and Pattern Matching Revisited

The semantics of backquote is defined in terms of cons-car-cdr view of expressions independent of the MetaC reader. In the typical case, the C value of a backquote expression ⟨‘ {e}⟩ is the expression (tree) e with subexpressions (subtrees) of the form ⟨\$ x⟩, where x is a symbol, replaced by the C value of x and subexpressions (subtrees) of the form ⟨\$ {w}⟩ 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.

$$‘\{‘\{a + \${y} + \\\${z}\}\} \tag{2}$$

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.

$$‘\{‘\{‘\{\${x} + \\\${y} + \\\\\${z}\}\}\} \tag{3}$$

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 MetaC 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 (subtree) of e of the form ⟨\$ x⟩ where x is a symbol atom with $\sigma(x)$. A pattern expression (tree) {e} matches an expression (tree) 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.