

MetaC

David McAllester

May 27, 2018

1 Motivation

MetaC is a C extension supporting notebook-style programming based on a read-eval-print loop with incremental procedure redefinition (dynamic linking) and the general look-and-feel of python programming environments. MetaC also extends C with the symbolic programming features of pattern matching, backquote, and computed macros. Pattern matching and backquote together allow one to directly express simple rewrite macros. However, rewriting is a trivial special case of the much more powerful notion of a computed macro where computation is supported by high level symbolic programming tools. Computed macros allow arbitrary high-level symbolic computation in macro expansion. Computed macros should be viewed as compilers. A computed macro can do sophisticated type inference or data-flow analysis as part of macro expansion. Compilers for sophisticated high-level languages can be implemented as packages of computed macros. When writing a compiler it is desirable to target a low level language offering considerable control over the hardware. In MetaC the macros expand to C. MetaC's computed macros are written in the MetaC extension of C. MetaC itself is written using macros defined in MetaC. But all such macros ultimately expand to C.

2 Universal Algebraic Syntax (UAS)

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 syntax (UAS). UAS is a compromise between the extreme syntactic simplicity of Lisp and the desire for a syntax that is more similar to C. While not as minimal as Lisp, UAS is still simple — there are seven syntactic categories including a category for the

null expression. UAS is universal in that the syntactic trees are considered to be semantics-free and, more importantly, under very weak assumptions a semantics defined by a compiler on strings can be transferred to UAS trees. This later property is made possible by ensuring that the UAS reader will accept any parenthesis-balanced string and that the process of reading a string into a UAS expression can be inverted — printing the expression recovers the original string. A semantics defined by a compiler on strings can be imposed on UAS expressions simply by printing the expressions and applying the compiler to the resulting strings.

The MetaC reader converts character strings to semantics-free syntax trees (expressions). The MetaC reader first preprocesses the input string to remove C-style comments. The preprocessor also divides file character strings (as opposed to REPL inputs) into top level segments. A segment starts at the beginning of any nonempty line whose first character is not a space or tab and where the preceding line is empty. A segment also starts at the beginning of any line beginning with the character `#` (indicating a C preprocessor line). The file must be parenthesis-balanced within each segment. This means that every open parenthesis, brace or bracket has a matching closing character and that string quotations are properly closed. The reader also has the property that white space is ignored other than in the determining the top level segmentation and in separating symbols. The reader converts each segment to a single expression. An expression is defined recursively to one of the following.

- A symbol. A symbol represents a string of alpha-numeric characters (upper and lower case letters of the alphabet, plus the decimal numerals, plus underbar). For example `foo_bar1`.
- A tagged symbol. A tagged symbol consists of a symbol-tag followed by a symbol. The symbol tag characters are `$`, `\`, `'`, `!`, `?`, `#` and `%`. For example we have the tagged symbols `?foo` and `!foo`.
- A quoted string. For example `"!:&?;"` or `'a'`. In UAS there is no distinction between `"foo"` and `'foo'` other than the choice of the quotation character (which must match). To avoid confusion newline characters are not allowed in strings (the two character sequence `/n` is ok).
- A parenthesis expression. This is an expression of the form (e) , $\{e\}$, or $[e]$ where e is an expression. In MetaC (e) is always a different expression from e .
- An application expression. This is a tagged symbol followed by a parenthesis expression or an application expression followed by a parenthesis expression. For example `foo(x)` or `foo(x)(y)`. The formation of application expressions is tighter than (higher precedence than) the formation of binary connective expressions.

- A binary connective expression. For example **x=3** is the binary connective expression with connective **=** and symbol arguments **x** and **3**. Binary connectives are described in more detail below.
- The null expression and the null connective. In order to ensure that the UAS reader can read any parenthesis-balanced string we can use the null expression as arguments and the null connective to combine pairs of adjacent expressions. This is described in more detail below.

Binary connectives. The binary connectives consist of all non-empty strings over the characters **;**, comma, **{=, ~, <, >}**, **{|, &}**, **{+, -}**, **{*, /}**, **., @**, and **^**. Single-character connectives are outermost to innermost in the order given (low precedence to high precedence) where set brackets indicate same-precedence. All binary connectives are infix. The last four single-character connectives are left-associative and the others are right-associative. For example **f(x).a.b** reads as

```

      .
     / \
    .   b
   / \
  /\  a
 f ( )
  |
  x

```

All multi-character binary connectives have the same precedence which is higher than (innermost to) all single character connectives other than **:**. Multi-character connectives are left associative. The null connective, described below, is right associative and is outermost to (lower precedence) than all other binary connectives except **;** and comma. This precedence of the null connective causes C argument lists and C statement sequences to have friendly tree structure.

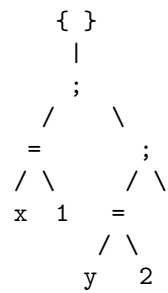
The null expression and the null connective. We can draw expressions as trees where the null expression and the null connective are drawn as empty nodes. For example **(foo bar)** is the expression.

```

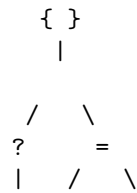
  ( )
   |
  /  \
foo   bar

```

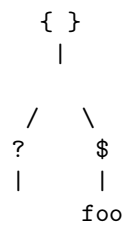
{x=1;y=2;} is the expression



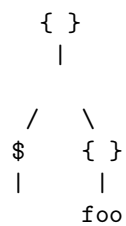
$\{?\} =$ is the expression



$\{?\$foo\}$ is the expression



$\{\$\{foo\}\}$ is the expression



Since any parenthesis-balanced string reads as an expression, we can think of expressions as structured representations of strings.

3 Pattern Matching

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

ucase(**e**; {< **pattern1** >} : {< **statement1** >}; ...; {< **patternn** >} : {< **statementn** >})

Variables are marked in patterns by the symbol tag **!**. For example we might write the following.

```
int value(exp 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)];};)
}
```

Here pattern symbols tagged with **!** are bound to expressions. 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 string **a+b*c** because **+** is above (outer to) ***** in the parse tree constructed by MetaC. Note that the pattern **!x*!y** does match **(a+b)*c**.

Repeated variables, such as the pattern **{!x + !x}** are not supported (and not checked for). If variables are repeated the variable will be matched at its last occurrences under depth-first traversal.

Pattern symbols can also be tagged with **?** in which case they must be bound to symbols. For example, in the following **?x** must bind to a symbol.

```
ucase{e;
  {dolist{?x !L}!body}:{
    rest = gensym('{rest}');
    return '{for(list ${rest} = ${L};
      ${rest} != NULL;
      ${rest} = cdr(${rest});)
    ${body}}}
```

4 Backquote

MetaC provides Lisp-style backquote. In the simplest case $\{e\}$ evaluates to a data structure which is the UAS syntax tree for the string e . For example, the MetaC expression $\{a+b+c\}$ evaluates to the expression $a+b+c$.

Backquote can also be used for pattern instantiation. When computing the value of $\{e\}$ subexpressions of e of the form $\$w$ are replaced by the value of the expression w . For example consider the following expression.

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

If the value of variable z is the expression c then the value of expression (1) is the expression $a+b+c$.

Structured symbols are expressions and structured symbols can be constructed by substitution into parts of the symbol. For example, if x is bound to **foo** then $\{\$x:bar\}$ evaluates to the structured symbol **foo:bar**.

We allow for expression evaluation inside backquote by writing applications of $\$$. For example if the procedure call $f(x)$ evaluates to the symbol **world** then $\{\text{hello } \{f(x)\}\}$ evaluates to **hello world**.

The unary operator $\$$ can be included in an expression constant by quoting it as in the following.

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

If the value of variable y is the expression b then the value expression of (2) is expression (1). Quotation expression can be included in the expression by adding another layer 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).

5 The MetaC Pretty Printer

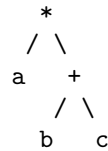
The printer converts an abstract syntax tree back into a byte string which includes newline bytes and space characters so that the expression is well formatted. The printer inverts the reader up to string equivalence (as defined by the reader). This emphasizes that we are thinking of the abstract syntax tree as a representation of a character string.

6 Reader-Printer Inverse Properties

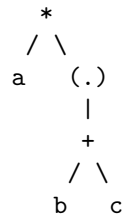
The white space characters are space, tab and newline. White space characters are generally ignored. There are two exceptions to this. First, white space between two alpha-numeric characters (upper and lower case letters and the numerals 0 to 9) causes two symbols to be read rather than one. Second, a newline that is not preceded by the quotation character `'/'` and is not followed by a white space character is treated as a file segment terminator. Parentheses must be balanced within each file segments.

For any character string we define the white space normal form of s to be the result of removing all c-style comments and then removing all white space characters except for a single space character between two alpha-numeric characters and where segment terminator newlines are replaced by null bytes.

The reader has the property that if string s and s' have the same white space normal form then the result of reading s is the same as the result of reading s' . The printer inverts the reader in the sense that reading a string s and then printing the result to get string s' has the property that s and s' have the same white space normal form. Conversely, the reader inverts the printer in the sense that, if syntax tree x can be represented by a string, then printing x and reading the result gives back x . However, there are syntax trees which cannot be represented by a string. For example we have that `"+"` is lower precedence than `*` and hence the tree



cannot be represented by a string. It can, however, be created with `{a * $x}` where x is the result of reading `b + c`. Note that parsing `a * (b + c)` results in the tree



which is different.

It is often more natural to think in terms of strings where tree structure exists only to disambiguate pattern matching in case expressions. When constructing expressions with backquote it is often advisable to insert parentheses around inserted values to ensure that the resulting expression is string-representable. For example, rather than write `{a * $x}` one should write `{a * ($x)}`.