MetaC

David McAllester

May 29, 2018

1 Motivation

MetaC is a C extension supporting a read-eval-print loop (REPL) capable of incremental procedure redefinition (dynamic linking) and the general look-and-feel of python programming. MetaC also extends C with symbolic programming features for writing sophisticated preprocessors (compilers) as packages of computed macros. Computed macros can do sophisticated type inference or data-flow analysis as part of macro expansion. MetaC supports computed macros with the high level symbolic progrmming features of pattern matching and pattern instantiation. Pattern instantiation is done with backquote — a language feature of Lisp. Generating C code using backquote (pattern instantiation) is facilitated by the use of a universal algebraic concrete syntax — a universal algebraic mapping between character strings and expression trees.

2 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.

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

```
MC> int x;

done

MC> x = 2;

done

MC> x += 1;

done

MC> int_exp(x)

3

MC>
```

The current implementation requires that expressions given to the REPL are either C statements of C expressions that evaluate to Universal Algebraic Syntax (UAS) expressions. UAS expressions are abstract syntax trees which are often viewed as representations of strings. The procedure `int_exp` converts an integer to a UAS expression. Backquote can be used in a manner analogous to string formatting.

```
MC> expptr friend = `{Bob Givan};

done

MC> int height = 6;

done
```

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

My friend Bob Givan is 6 feet tall.

MC>
```

While it is often appropriate to think of UAS expressions as representations of strings, UAS 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 = ‘{a+b};

done

MC> ‘{bar(${x}}

bar(a+b)

MC>
```

The pattern `foo(!x)` will match the expression `foo(a+b)` with `x` bound to the expression (tree) `a+b`. Pattern matching is described in more detail below.

It is sometimes convenient to use a Gnu C extension in which a basic block defines an expression. For example, `({int x = 2; 5*x;})` is an expression whose value is 10. We also have the following.

```
MC>({expptr x = ‘{a+b}; ‘{bar(${x}};})

bar(a+b)

MC>
```

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

```
MC> typedef struct ptrliststruc{
  void * car
  struct ptrliststruct * cdr;
} *ptrlist;
```

```
done

MC> typedef  (void *) ptr_to_ptr(void *);

done

MC> ptrlist mapcar(ptr_to_ptr f, ptrlist l){
     if(l == NULL)return NULL;
     return cons(f(l->car),mapcar(f,l->cdr);
}

done

MC>
```

The function `cons` involves memory allocation and the definition has been omitted. MetaC supports a couple approaches to memory management described below. The type ptrlist exposes a weakness in the C type system — we would like cons to be a polymorphic function supporting many different types of elements of lists. A language with a richer type system can be implemented as a MetaC macro package.

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 tags **!** and **?**. Variables tagged with **!** can bind to any UAS expression while variables tagged with **?** can only bind to atomic symbols. For example we might write the following.

```
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)]);})
}
```

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. The pattern **!x*!y** does match **(a+b)*c**.

In the current implementation 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.

As another example we consider a code fragment used in the expansion of an iteration macro.

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

4 Nested Backquotes

The general form of a backquote expression is ‘{<exp>} where the value of this is the given expression <exp> with subexpressions of the form ${<exp2>} replaced with the computed C value of <exp2>. This simple rule is unfortunately incomplete. One of the most confusing situations is where the expansion of a macro contains a backquote. Writing such a macro involves nested backquotes. While nested backquotes are confusing, and should be avoided when possible, they are none the less supported. 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 unary operator **$** can be included in an expression constant 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). Quotation expression can be included in the expression by adding another layer 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).

5 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 also defines a semantics on UAS expression 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.

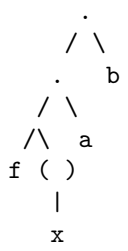
The MetaC reader converts character strings to semantics-free syntax trees (expressions). The MetaC reader first preprocesses the input string to remove C-style comments. The preprocessor also divides file character strings (as opposed to REPL inputs) into top level segments. A segment starts at the 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 begining 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).

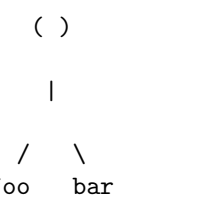
- A parenthesis expression. This is an expression of the form (e), $\{e\}$, or $[e]$ where e is an expression. In MetaC (e) is always a different expression from e.
- An application expression. This is a tagged symbol followed by a parenthesis expression or an application expression followed by a parenthesis 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 conectives 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-chracter connectives are left-associative and the others are right-associative. For example **f(x).a.b** reads as

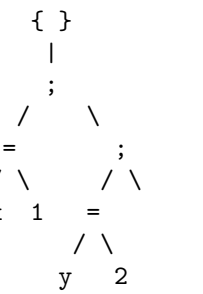


All multi-character binary connectives have the same precedence which is higher than (innermost to) all single character connectives other than **:**. Multi-character connectives are left associative. The null connective, described below, is right associative and is outermost to (lower precedence) than 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.

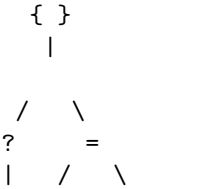
The null expression and the null connective. We can draw expressions as trees where the null expression and the null connective are drawn as empty nodes. For example **(foo bar)** is the expression.



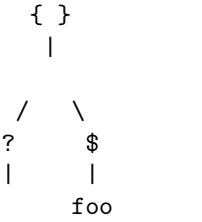
{x=1;y=2;} is the expression



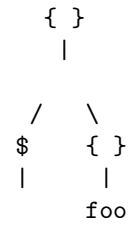
{?=} is the expression



{?$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.

6 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.

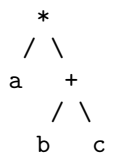
7 Reader-Printer Inverse Properties

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 numberals 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.

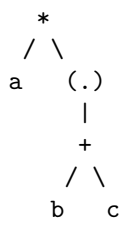
For any character string we define the white space normal form of s to be the result of removing all c-style comments and then removing all white space characters except for a single space character between two alpha-numeric characters and where segment terminator newlines are replaced by null bytes.

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 prining the result to get string s' has the property that s abnd s' have the same white space normal form. Conversely, the reader inversts 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 precidence than * and hence the tree



cannot be represneted 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 aften 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)`}.