MetaC

David McAllester

June 6, 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. Ar-

bitrary types can be given single symbol names using `typedef`.

2 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`.

2.1 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 `?`. Variables tagged with `!` can bind to any UACS 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);}}
return 0;
}
```

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 abstract syntax tree. 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.

2.2 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 macros until no more 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 general a macro patterns must either be a 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.

3 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). It should be noted up front that this section is a specification of behavior which bears little resemblance to the actual implementation.

UACS is "universal" in that it has the following three properties.

1. UACS syntax trees are semantics-free. They are simply trees and can be viewed as representations of character strings.

2. Any parenthesis-balanced character string can be read by the UACS reader producing a UACS expression tree. Here parenthesis-balanced means that every open parenthesis, brace or bracket has a matching closing character and that string quotations are properly closed.

3. UACS Expression trees represent the strings from which they are constructed — 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 — they lexicalize to the same sequence of whitespace-free tokens as described at the beginning of section 3.2. This implies that s and s' both read to the same expression e. It also implies that the strings s and s' will be treated equivalently by the C compiler.

The 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 the expression-terminating semicolon are particularly helpful in aligning UACS tree with C syntax.

3.1 UACS Expressions

A UACS expression is simply a tree structure imposed on a character string. More formally, a UACS expression is either a whitespace-free character string or a finite sequence of UACS expressions. The UACS reader produces UACS expressions of a particular form described below. A UACS expression can be represented by "phantom brackets" which are not printed when printing the expression. For example, the reader takes the string {`int x = 1; y=2;`} and imposes the phantom bracketing

$$\langle\{\ \langle\langle\texttt{int}\rangle\langle\texttt{x} = \texttt{1}\rangle;\rangle\ \langle\rangle\ \langle\langle\texttt{y} = \texttt{2}\rangle;\rangle\ \}\rangle.$$

Here the empty expression $\langle\rangle$ is acting as a binary connective. Whitespace separates whitespace-free character stings so that $\langle\texttt{x} = \texttt{1}\rangle$ is a sequence of three strings. The printer simply removes the phantom brackets but leaves a space

character after each whitespace-free character string to ensure proper lexicalization when the printed string is read. In practice a pretty printer inserts newline characters and indentation to make large expressions more readable.

All UACS expressions returned by the reader are one of the following.

- A symbol. 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 the string `foo_bar1`.
- A connective. A connective is a character string consisting of only "connective characters" — the characters comma, `=`, $\sim$, $<$, $>$, `|`, `&`, `+`, `-`, `*`, `/`, `.`, `@`, `^`. For example the strings `->` and `==`.
- The null expression $\langle\rangle$ represents the empty string and is both a symbol and a connective.
- A quoted string. A character string starting and ending with the same string quotation character. For example `"!:&?;"` or `'a'`.
- A parenthesis expression. This is an expression of the form $\langle(e)\rangle$, $\langle\{e\}\rangle$ or $\langle[e]\rangle$ where e is a UACS expression.
- A semicolon expression. This is an expression of the form $\langle e$ `;`$\rangle$ where e is an UACS expression. As described in section 3.2, the reader treats a semicolon very similarly to a parenthesis close character (`)`, } or `]`). A semicolon expression is like a parenthesis expression but formed with a single character rather than a pair of characters.
- An orphan character. This is a string consisting of a single non-whitespace character not mentioned above. In particular, one of the characters `$`, \, `‘`, `!`, `?`, `#` and `%`.
- A tagged symbol. A tagged symbol is a UACS expression of the form $\langle c\ s\rangle$ where c is an orphan character and s is a symbol.
- A generalized application expression. This is a UACS expression consisting of an orphaned character, a symbol or a tagged symbol followed by a sequence of parenthesis expressions. For example `foo(x)`, `foo(x)(y)`, `?foo(a)` or `?(a)`.
- A connective expression. This is a UACS expression of the form $\langle e_1\ R\ e_2\rangle$ where e_1 and e_2 are UACS expressions and R is a connective. For example, the reader converts `x=3` to the connective expression $\langle$`x` $=$ `3`$\rangle$.

Note that with the exception of generalized application expressions, the expressions returned by the UACS reader never have more than three subexpressions. Long sequences of arbitrary expressions are represented by nested null-connective expressions. Examples of the bracketings imposed by the UACS reader are shown in figure 1.

$$
\begin{array}{rcl}
\texttt{Hello World} & \Rightarrow & \langle \texttt{Hello} \ \langle\rangle \ \texttt{World} \rangle \\
\texttt{one two three} & \Rightarrow & \langle \texttt{one} \ \langle\rangle \ \langle \texttt{two} \ \langle\rangle \ \texttt{three} \rangle \rangle \\
\texttt{x} + \texttt{y} * \texttt{z} & \Rightarrow & \langle \texttt{x} + \langle \texttt{y} \ * \ \texttt{z} \rangle \rangle \\
(\texttt{x} + \texttt{y}) * \texttt{z} & \Rightarrow & \langle \ \langle (\ \langle \texttt{x} \ + \ \texttt{y} \rangle \ ) \rangle * \texttt{z} \rangle \\
\texttt{foo(int x)} & \Rightarrow & \langle \ \texttt{foo} \ \langle (\ \langle \texttt{int} \ \langle\rangle \ \texttt{x} \rangle \ ) \rangle \ \rangle \\
\texttt{foo(int x, float y)} & \Rightarrow & \langle \texttt{foo} \ \langle (\ \langle \langle \texttt{int} \langle\rangle \texttt{x} \rangle \ , \ \langle \texttt{float} \langle\rangle \texttt{y} \rangle \rangle \ ) \rangle \ \rangle \\
\texttt{?f(!args)} & \Rightarrow & \langle \ \langle \texttt{?} \ \texttt{f} \rangle \ \langle (\ \langle \texttt{!} \ \texttt{args} \rangle \ ) \rangle \ \rangle \\
\texttt{`\{x\}} & \Rightarrow & \langle \ \texttt{`} \ \langle \texttt{\{} \ \texttt{x} \ \texttt{\}} \rangle \ \rangle \\
\texttt{\{int x = 1; y = 2;\}} & \Rightarrow & \langle \texttt{\{} \ \langle \langle \texttt{int} \langle\rangle \langle \texttt{x = 1} \rangle \rangle \texttt{;} \rangle \ \langle\rangle \ \langle \langle \texttt{y = 2} \rangle \texttt{;} \rangle \ \texttt{\}} \rangle
\end{array}
$$

Figure 1: **Examples of UACS reader bracketing.** The first two expressions are null connective expressions. The next two are connective expressions. The next four are all generalized application expressions. The last one is a parenthesis expression containing a null connective expression connecting two semicolon expressions.

3.2 Reader Specification

The above discussion ignores the presence of C comments. The MetaC reader first preprocessing the input string to remove comments. The Meta preprocessor also divides long character strings, such as entire code files, into segments where each segment is read as a separate expression. The partitioning into segments is different in file inputs than in REPL inputs. For file inputs 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 quoted return — the character '\\' followed by '\n' — does not start a new line. This is important for "multi-line" preprocessor definitions. A file must be parenthesis-balanced within each segment. For REPL inputs the segment ends at the first return character outside of parenthesization independent of further input characters. This allows the REPL to respond immediately to the top level return character.

We allow the reader to be applied to the empty string (or a string consisting only of white space) in which case the reader returns the null expression. For example, the strings `()` and `( )` read as $\langle(\langle\rangle)\rangle$.

Given a nonempty string with comments removed the reader first "lexicalizes" the whitespace-free strings into symbols, connectives, quoted strings (which are considered whitespace-free), and single character strings for the remaining non-white characters. Two strings are whitespace-equivalent if they result in the same lexical sequence.

To specify the reader further we first consider semicolon expressions. UACS

treates a semicolon as a kind of close parenthesis. Like a pair of balanced parentheses, the subexpression of a semicolon expression has a trivially determined span in the text. We define the "begin position" for a semicolon to be the nearest position to the left of the semicolon that is either an open parenthesis not closed in the interveaning span, a semicolon not contained in the span of an intervening close parenthesis, or the beginning of the read. The span of the argument to a semicolon is simply the subsequence between the begin position (but not including the begin position) and the semicolon. The argument expression for a semicolon is simply the reader applied recursively to the subexpression span. A sequence of semicolon expressions is effectively a sequence of parenthesis expressions but with one character instead of two characters to define each span.

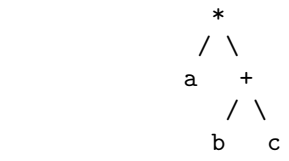
To specify the reader it now suffices to consider a sequence of connectives, symbols, orphan characters, parenthesis expressions, and semicolon expressions. The first step is to group each pair of an orphan character followed by a symbol into a tagged symbol. Next we group any sequence of an orphan character, a symbol or tagged symbol followed by parenthesis expressions into a generalized application expression. This leaves a sequence of connectives, symbols, tagged symbols, generalized application expressions, parenthesis expressions and semicolon expressions. Anything other than a connective in this sequence will now be called an argument. The null expression can be used as a connective and we now insert the NULL expression between each pair of adjacent arguments. The null expression can also be used as an argument and we also insert the null expression between each pair of adjacent connectives. If the sequence starts with a connective we insert the null expression at the beginning of the sequence. Similarly, if the sequence ends with a connective we add the null expression at the end. We now have an alternating sequence of arguments and connectives starting and ending with arguments. It now suffices to specify operator precedence and left/right associativity for precedence ties.

The connective characters are comma, {`=`, $\sim$, $<$, $>$}, {`|`, `&`}, {`+`, `-`}, {`*`, `/`}, `.`, `@`, `^` Single-character connectives other than space have precedence that is outermost to innermost in the order given above (low precedence to high precedence) where grouped characters have the same-precedence. The null connective is outer to (lower precedence than) all other connectives except comma. Having comma be outer to the null connective causes C procedure argument lists to have an appropriate tree structure.

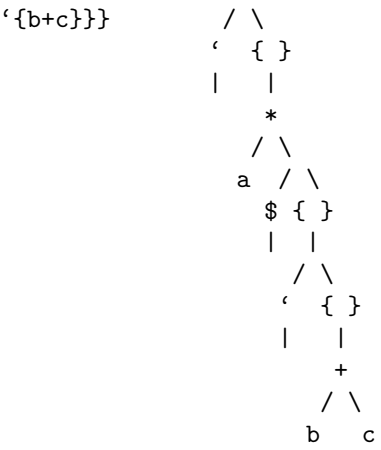
The last four single-character connectives in the above list are left-associative and the others are right-associative. All multi-character binary connectives have the same precedence which is higher than (innermost to) all single character connectives other than `:`. Multi-character connectives are left associative.

3.3 Backquote Specification

There are UACS trees that can be constructed by backquote but which cannot be represented by a string. For example we have that `+` is lower precedence than `*` and hence the expression



cannot be represented by a string. It can, however, be created with backquote. Backquote and pattern matching both operate on trees rather than character strings. The above tree can be created as the value of `‘{a * ${‘{b+c}}}`. In general a subexpression of the form `${e}` inserts the value of e at the node of the syntax tree for the backquote expression where the `${e}` subexpression occurs. The backquote expression has the following tree.



The general form of a backquote expression is `‘{<exp1>}` where the value of this is the given expression `<exp1>` with the nodes in the tree for `<exp1>` at which subexpressions of the form `${<exp2>}` appear replaced with the computed C value of `<exp2>`. Under this evaluation rule the value of above backquote expression is the preceding tree which has no string representation.

Unfortunately the above simple 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, nested backquotes are 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 constructed with backquote 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). 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).

4 Interning, Property Lists, and Memory Management

4.1 Expression Interning

4.2 Expression Properties

4.3 Undo Stacks and Undo Memory Management

4.4 The MetaC Execution Stack for Debugging

4.5 Allocating Memory from the MetaC Execution Stack

5 System Features

5.1 The Emacs Interactive Development Environment

5.2 MetaC Source Code File Management

MetaC macro-expander executables macro-expand MetaC source code files .mc into a C source file .c. The MetaC macro-expander applications are written

in MetaC — MetaC is bootstrapped on itself. The expanders are defined in layers where each layer has a set of active macros. Each layer has both an expander executable and a REPL with the same set of macros pre-installed. A new layer with more installed macros can be constructed using the previous layer to expand files with additional code including additional macro definitions.