McC: Lisp-like C Programming

David McAllester

April 27, 2018

1 Motivation

McC is a C extension supporting the symbolic programming features of Lisp and the interactive programming environment of scripting languages such as Python.

The fundamental language features of interest are pattern matching[1], backquote and computed macros. Pattern matching is familiar in many languages. Backquote is a generalization of Lisp quotation supporting the insertion of computed values into the quoted expression. Backquote will be familiar to those versed in writing Lisp computed macros. Pattern matching and backquote together allow one to express rewriting. To rewrite an expression ones used pattern matching to bind variables to parts of the expression and uses backquote to construct an expression using the values of the bound variables. Computed macros allow arbitrary source code (Lisp code in the case of Lisp) to be used in computing the expansion of a macro. Computed macros should be viewed as compilers. A computed macro can do sophisticated type inference or data-flow analysis as part of macro expansion.

Packages of computed macros are often used as compilers for sophisticated languages. However, given that one is writing a compiler, C is a preferable target language. Experience with Lisp indicates that for computed macros to work smoothly it is important that the source language be the same as the target language. The target language should be C.

[1]Pattern matching is not a feature of Lisp but is easily implemented given backquote and computed macros

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 nine categories of expression 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 defined instead on the 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 McC reader converts character strings to semantics-free syntax trees which we call expressions. The McC reader first preprocessing the input string to remove C-style comments. The preprocessor also divides the character string (or file) into top level segments. We adopt the convention that each top level segment starts at a nonempty line whose first character is not whitespace. The preprocessor inserts a null byte at the end of each top level segment (just before the beginning of the next segment). The preprocessed character string must be parenthesis-balanced within each top level 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 top level segment of an input string (or input file) to an 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 structored symbol. A structured symbol is built from symbols, prefix symbol modifier characters, and the symbol connective **:**. For example we have the structured symbols **?foo**, **foo:bar** and **?foo:bar** where modifier characters bind more tightly than the connecive **:**. We can pattern-match into the structure of structured symbols. The symbol prefix modifiers are the characters **!**, **$**, **#**, **‘**, \, **%**, and **?**.
- A quoted string. For example `"!:&?;"` or `'a'`. In UAS there is no dis-

tinction between **"foo"** and **'foo'**. 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 McC (e) is always a different expression from e.

- An application expression. This is a structured symbol followed by a parenthesis expression. For example **foo(x)**. The binding of structured symbols to parenthesis expressions to form application expressions is tighter than (higher precedence than) the formation of binary connective expressions.

- A terminated expression. This is an expression followed by the terminator by the character **;**. For example **x=3;**.[2] The termination symbol **;** binds more weakly (has lower precedence) than all binary connectives.

- 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. We also allow a null binary connective. For example **(foo bar)** is a parenthesis expression containing the two symbols **foo** and **bar** connected by the null connective. We have that **{x=1;y=2;}** is a parenthesis expression containing the two terminated expressions **x=1;** and **x=2;** connected by the null connective. We have that **{?=}** is a parenthesis expression containing the symbol modifier **?** applied to the null expression and the connective expression = applied to two null expressions and where these two expressions are connected with the null connective.

The binary connectives consist of the null connective and all non-empty strings over the characters comma, {=, ∼, <, >}, {|, &}, {+, -}, {*, /}, ., @, and ^. All binary connectives are infix. Single-character connectives are outermost to innermost in the order given (low precedence to high precedence) where set brackets indicate same-precedence. The last three single-chracter connectives are left-associative and the others are right-associative. For example **f(x).a.b** reads as **f(x).a** and **b** connected with dot. Multi-character connectives are innermost to (higher precedence than) all single character binary connectives except **:** which is the innermost binary connective. All multi-character binary

[2]There is near-universal consensus that it is more reasonable to treat **;** as a connective rather than a terminator. However, the primary application of UAS is a representation of C and making **;** a terminator simplifies the manipulation of C code.

connectives have the same precedence. Multi-character connectives are left associative. The null connective is right associative and is outermost to (lower precedence) than all other binary connectives except comma — the null connective binds more tightly than comma (this causes argument lists to have a friendly tree structure).

3 Pattern Matching Case Statement

McC 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 `"!"`. 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)]);})
}
```

In many cases the choice of the particular tree structure imposed by the McC 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 McC. Note that the pattern `!x*!y` does match `(a+b)*c`.

Structured symbols are expressions with tree structure. The pattern **!x:bar** matches the structured symbol **foo:bar:gritch** with x binding to **foo:bar**.

4 Backquote

McC provides Lisp-style backquote. In the simplest case the expression **‘{**e**}** (now an McC code expression) evaluates to a data structure which is the UAS syntax tree for the string e. For example, the McC 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} suexpression 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`. The unary operator `$` can be included in an expression constant by quoting it as in the following.

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

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 The McC 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 numberals 0 to 9) causes two symbols to be parsed rather than one. Second, a newline that is followed by a character other than a white space character or a parenthesis character (one of "(", ")", "{", "}", "[" or "]"), but is not preceded by "\\", is treated as a "hard stop" and forces the reader to finish on whatever character string precedes the newline. 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 hard stop newlines are not removed.

The reader has the property that if string s and s' have the same white space normal form then the result of parsing 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. These two properties imply that 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

```
  *
 / \
a   +
   / \
  b   c
```

cannot be represneted by a string but can be created with `‘{a * $x}` where x is the result of parsing `b` + c. Note that parsing `a * (b + c)` results in the tree

```
  *
 / \
a   (.)
     |
     +
    / \
   b   c
```

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)}`.