# McC: A Lisp-like extension of, and Environment for, C Programming

David McAllester

April 19, 2018

## 1 Universal Concrete Syntax

McC is a C extension intended to support language features, and the symbolic programming philosophy, of Lisp. A fundamental aspect this philosophy is a notion of expression (symbolic expressions or s-expression) where the concrete syntax of expressions is orthogonal to (independent of) their semantics. Lisp expressions are "universal" in the sense that they can be assigned the semantics of any desired language with any desired language features. This is often done with macros built with the backquote feature of Lisp. McC similarly supports a universal notion of expression with a universal concrete syntax defined by a reader mapping character strings to syntax trees. McC also supports backquote and Lisp-like computed macros. Unlike Lisp, however, the universal expressions of McC are largely compatible with the algebraic syntax of C. Although no attempt is made to to make the McC reader replicate the C parser, the syntax is sufficiently similar to C that writing C preprocessors as Lip-like macros is easy. Perhaps most importantly, the mismatch between the simple universal reader of McC and the C parser is overcome by the fact that the reader preserves the string. Rather than having the reader invert the printer, in McC the printer inverts the reader. This means that the reader does loose information and, in particular, a valid C expression is preserved when read and then printed. This is a much easier property to achieve than designing a reader that accurately parses all of C (or C++). The McC reader is also universal in the sense that it places minimal requirements on the character strings that can be parsed into expressions. The parser operates on strings of ASCII printable characters plus space, tab and newline, and requires only that quotation characters, parentheses, brackets and braces can be matched into open and close pairs.

## 2 Universal Concrete Syntax

For file inputs the UCS parser first passes the input character stream through a "stream cleaner" which removes C-style comments and inserts null bytes at certain termination points. A terminating point occurs at the first character of a line that starts with a character other than whitespace, another newline, a parenthesis, bracket or brace. Such positions are assumed to be the start of a new top level expression in the file. The stream cleaner inserts the null character '\0' at the beginning of any such line.

The UCS parser operates on the cleaned input stream and is specified by the following properties.

- The UCS parser converts each maximal substring of alpha-numeric characters (upper and lower case letters of the alphabet, plus the decimal numerals, plus underbar into a lexical token.
- The UCS Parser treats strings of the form $(e)$, $\{e\}$, or $[e]$ has having both a syntax node for the expression $e$ as well as a seperate syntax node for the matched pair of parenthesis, brackets, or braces.
- The UCS parser recognizes the characters comma, space, {`=`, $\sim$, `<`, `>`}, {`|`, `&`}, {`+`, `-`}, {`*`, `/`}, `.`, `@`, `^` and `:` as binary connectors (binary operation symbols). These connectors are outermost to innermost in the order given (low precedence to high precedence) where set brackets indicate same-precedence.

The last four are left-associative and the others are right-associative. Any consecutive sequence of more than one of these binary connective characters (with no intervening white space) is taken to be a left-associative binary connective. Multi-character binary connectives are taken to be innermost to (higher precedence than) and single-character binary connectives.

- The UCS parser recognizes the characters !, $, #, ‘, \, %, and ? as prefix unary connectives. These unary connectives are innermost to (higher precedence than) all binary connectives.
- Following C, the UCS parser recognizes the character ’;’ as a unary postfix connective. This connective is outermost to (lower precedence than) all binary connectives.
- The null character ’\0’ inserted by the cleaner at the beginning of new expressions (as defined above) terminates an expression. An error is generated if ’\0’ occurs inside parentheses, brackets, or braces. The null character does not appear in parsed expressions but serves to separate expressions in character streams containing multiple expressions.
- The UCS parser recognizes an alphanumeric constant followed by a parenthesis, bracket or brace as an application node. The formation of application nodes is innermost to (higher precedence than) all connectives.
- The UCS parser inserts null arguments and space connectives as needed so as to be able to parse any well-balanced string.

Examples:

`!foo(x)` This parses as the unary operator `!` applied to the application `foo(x)`.

`A[x++]` This parses as an application node with operator `A` and argument `x++`. The node for `x++` is labeled with the connective `++` and has first child `x` and a null second child.

`{[a](b)c}` This parses as a tree with a root node labeled with {} with one child representing `[a](b)c`. The node representing `[a](b)c` is labeled with the space connective and has first child `[a]` and second child `(b)c`. The node representing `[a]` is labeled with `[]` and has one child labeled with `a`. The node representing `(b)c` is labeled with the space connective and has children `(b)` and `c`. The space connective is right associative.

`f(x).a.b` This parses as a node labeled with the dot connective and with first child `f(x).a` and with second child `b`. The node for `f(x).a` is labeled with a dot connective with first child `f(x)` and second child `a`. The node for `f(a)` is an application node. The dot connective is left associative.

# 3 Pattern Matching Case Statement

UCC 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(expptr e){
  ucase(e;
   {!x+!y}:{return value(x)+value(y);};
   {!x*!y}:{return value(x)*value(y);};
   {!z}:{return atoi(string_table[constructor(z)]);})
}
```

In many cases the choice of the particular tree structure imposed by the UCS parser 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 UCS. Note that the pattern `!x*!y` does match `(a+b)*c`. One can generally use parentheses, brackets or braces to force a desired tree structure. To evaluate an expression with parentheses, such as `(a+b)`, we can add the following clause to the evaluator.

```
int value(expptr e){ucase(e; ... {(!x)}:{return value(x);}; ...)}
```

# 4 Backquote

The UCS C extension provides Lisp-style backquote. In the simplest case the expression ‘$\{e\}$ appearing as a C expression evaluates to a data structure which is the syntax tree for $e$. For example, the UCC expression ‘{`a+b+c`} Evaluates to a data structure representing the expression (syntax tree) for `a+b+c`. This is somewhat analogous to the more familiar notion of string constant such as `"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 UCS Pretty Printer

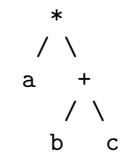
The printer converts an abstract syntax tree back into a byte string which includes newline bytes and space characters so that the expression is well formatted. The printer inverts the parser up to string equivalence (as defined by the parser). This emphasizes that we are thinking of the abstract syntax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

# 6 White Space Removal and Parser-Printer Inverse Properties

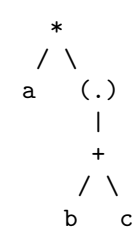
The white space characters are space, tab and newline. White space characters are generally ignored. There are two exceptions to this. First, white space between two alpha-numeric characters (upper and lower case letters and the numberals 0 to 9) causes two symbols to be parsed rather than one. Second, 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 parser 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 parser 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 parser 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 parser 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 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




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)}`.