# MetaC 1.0

David McAllester

September 10, 2018

## Abstract

MetaC provides a read-eval-print loop (a REPL) and notebook interactive development environment (a NIDE) for C programming. MetaC can also be described as a Lisp-inspired programming environment for C. The REPL and the NIDE are capable of incrementally evaluating (compiling and running) C statements and incremental procedure definitions in a persistent C process. This is implemented by incrementally compiling dynamic load libraries which are then linked into the persistent process.

MetaC also extends C with Lisp-like features. More specifically, MetaC provides a C-like universal syntax, computed macros, backquote [1], and pattern-matching. The implementation is bootstrapped — it is implemented in itself. These C extensions are intended to support the development of high level frameworks with notebook interfaces as direct extensions of C. This is in contrast to scripting-C hybrids such as Matlab, Numpy, TensorFlow or PyTorch.

The NIDE is implemented in Emacs piped to a MetaC REPL. Presumably notebook IDEs for C or C++ could also be implemented as extensions of Atom or Visual Studio Code piped to a persistent process.

Dynamic linking is difficult to implement robustly. MetaC release 1.0 runs under both Ubuntu 18.04.1 / gcc 7.3.0 and MAC OSX 10.11.4 / gcc Apple LLVM version 7.3.0 (clang-703.0.31).

# Contents

# 1 Introduction and Overview

## 1.1 Installation

To install the REPL:

1. Clone the MetaC repository in a local MetaC directory.

2. From a shell running in your MetaC directory type make MC.

3. From the same shell type MC to get the command line prompt MC>.

To install the NIDE:

1. Clone the repository into a repository directory.

2. From a shell running in your MetaC directory type make NIDE.

3. Add (`load "<MetaC directory>/mc.el"`) to your .emacs file.

4. Edit the first line of mc.el to give the path to your gdb executable (the Gnu debugger).

5. Edit the second line of mc.el to give the path to your MetaC directory.

6. Start or restart Emacs. The file mc.el adds hooks such that it is safest to restart Emacs rather than load em.el into a running Emacs.

The NIDE can be run on any file with extension .mc in any directory. To experiment with the NIDE use Emacs to visit the file NIDE-examples.mc from the MetaC repository. Type C-M-s (control-meta-s) to start (or restart) the C kernel. Typing C-c C-c will run the cell containing the cursor and print the result in a C comment at the bottom of the cell. This is substantially equivalent to typing the cell contents into the REPL. More Emacs key bindings and other features of the NIDE are described in section 1.10.

The following examples use the REPL. However, all of these examples can be run as cells in the NIDE.

## 1.2 Hello World

```
bash$ make MC
...
```

```
bash$ MC

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.

## 1.3   C Statements and C 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 universal syntax tree (also called an expression, giving a second, and confusing, sense of the term "expression"). The procedure int_exp above converts an integer to a universal syntax expression. Universal

4

syntax 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>
```

## 1.4 Definitions of Types and Procedures

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

```
MC>typedef struct myexpstruct{
   char * label;
   struct myexpstruct * car;
   struct myexpstruct * cdr;} myexpstruct, *myexp;

done

MC> myexp mycons(char*s,myexp x,myexp y){
    myexp cell=malloc(sizeof(myexpstruct));
    cell->label=s;
    cell->car=x;
    cell->cdr=y;
    return cell;}

done

MC> expptr myexp_exp(myexp x){
   if(x==NULL)return string_atom("nil");
   return
    '{${string_atom(x->label)}
      ${myexp_exp(x->car)}
      ${myexp_exp(x->cdr)}};
}

done
```

```
MC> myexp_exp(mycons("foo",mycons("bar",NULL,NULL),NULL))


foo bar nil nil nil
MC>
```

Procedures can also be redefined at the REPL provided that the signature (argument types and return type) remains the same. Changing a procedure signature requires restarting the MetaC REPL. This restricting ensures meaningful C type checking in the presence of dynamic linking.

## 1.5   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.

## 1.6   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. Arbitrary types can be given single symbol names using typedef.

## 1.7 Backquote and Universal Syntax

We now consider backquote and universal syntax 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 universal syntax expressions can be viewed as representations of strings, universal syntax 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 e[0] = '{a+b};

done

MC> '{bar(${e[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`.

## 1.8 Pattern Matching

MetaC provides a pattern matching case construct `ucase` (universal case). The general form of the case construct is

```
ucase{e;
   {<pattern1>}:{<body1>}
    ...
   {<patternn>}:{<bodyn>}}
```

Variables are marked in patterns by the prefix $.

```
MC>int numeralp(expptr x)
    {if(!atomp(x))return 0;
    char*s=atom_string(x);
    for(int i=0;s[i]!='\0';i++){if(s[i]<'0'||s[i]>'9')return 0;}
    return 1;
}

done

MC> 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}.(numeralp(z)):{return atoi(atom_string(z));}
    }
  return 0; //this dead code convinces the compiler there is a return value.
  }

done

MC>int_exp(value('{5+2*10}))

25

MC>int_exp(value('{foo}))


 match error: the value


foo
does not match any of


{$z}.(numeralp(z)){($x)}{$x*$y}{$x+$y}
```

```
MC>
```

In many situations the choice of the particular tree structure imposed by the MetaC reader is not important as long as the same conventions are used in both the patterns and the expressions they are matching. 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 expression a+b*c because the reader brackets a+b*c as $\langle$a $+\langle$b $* $c$\rangle\rangle$. The pattern $x*$y does match (a+b)*c. The variable $any is special. It acts as a wild card and is not bound to a value.

When a ucase is executed the patterns are tried sequentially and stops after the first match. If no pattern matches an error is generated. One can always add a final pattern of {$any} to provide a default catch-all case if that is desired.

In release 1.0 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.

## 1.9   Gensym and Macro Definitions

MetaC supports computed macros. Pattern matching and backquote greatly facilitate the writing of computed macros. Backquote and ucase are both implemented as computed macros in the bootstrapped implementation. 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=$L; cellp($rest);$rest=cdr($rest);){
      expptr$x=car($rest);
      $body}};
}

done

MC>macroexpand('{mydolist(item,list){f(item);}})

for
  (explist _genrest33=list;
   cellp(_genrest33);
  _genrest33=cdr(_genrest33);
```

```
  ){expptr item=car(_genrest33);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 pattern is restricted so as to have an identifiable head symbol to which the macro is attached. The procedure `macroexpand(exptr e)` takes a universal syntax expression and returns the result of repeatedly expanding outermost macros until no further macro expansion is possible.

The macro expansion of the above definition of `mydolist` defines a procedure to compute the macro expansion and installs that procedure on a macro property of the symbol `mydolist`. A typical macro pattern is a symbol followed by one or more parenthesis expressions in which case the macro is attached to the initial symbol. The pattern can also be a binary connective expressions such as $x. >$`y` in which case the macro is attached to the connective — `.>` in this case.

## 1.10  The Notebook IDE

Starting the kernel with the key C-M-s creates a running C process. When a cell is executed with the command C-c C-c the text of the cell is (invisibly) passed to $C$ process by the process is placed inside a comment below the cell. If a comment already occurs directly below the cell this comment is replaced by a new comment containing the response of the REPL. As in a Jupyter notebook, the values returned from the REPL are numbered so that one can see when the cell was executed relative to other cell execution in the notebook. If C-M-s is run when a kernel is already running the kernel process is killed and a new one started (as in a Jupyter notebook). When the kernel is restarted the execution numbering is reset to start with 1. Changing a procedure signature (argument and return types) requires restarting the kernel.

**Compile-Time Errors.** If an error occurs while attempting to read or compile the cell expression an error statement is displayed — typically the output of the C compiler — and an error message is printed in the code buffer as the value of cell.

**Run Time Errors and Breakpoints.** The procedure `berror(char *)` can be called from C code when a dynamic safety check fails. MetaC primitives call `berror` when safety checks fail. When a `berror` is called during execution gdb is

entered in a interactive debugging REPL. Giving the continue command to gdb will abort the cell execution and return to the code buffer with the persistent C process intact.

The MetaC procedure `breakpt(char *)` can be used for breakpoints from which execution can be continued. When this procedure is called gdb will be entered. Giving the continue command to gdb will continue from the breakpoint.

Segmentation faults are trapped by gdb directly. In this case the NIDE will move to the gdb REPL for investigating the error. But the normal gdb continue command will not return to the code buffer. In this case typing `print return_to_NIDE()` will in most cases return to the code buffer with the persistent C process intact. When memory corruption is a possibility it is safest to restart the C process.

**Printing.** The procedure `mcprint` is like `fprintf` but omitting the file argument. The expression `mcprint("x = %s\n",s)` will print to the *Messages* buffer in Emacs. The expression `mcpprint(e)` will pretty-print the expression $e$ to the *Messages* buffer.

**Key Bindings.** mc.el provides the following key bindings.

C-M-s  starts or restarts the persistent C process.

C-c C-c  executes the cell containing the cursor.

C-c C-r  executes all cells in the current region sequentially with a different execution number for each cell execution.

C-M-a  moves to the begging of the current cell.

C-M-p  moves to the beginning of the preceding cell.

C-M-n  moves to the beginning of the next cell.

Executing the eLisp expression `(MC:strip-cell-values)` in a code buffer will remove the cell values. This is useful before a git commit as it can reduce or eliminate differences in files between commits.

**Include Statements.** A cell in a code file containing **#include(¡filename¿)** (with no semicolon) will load the cells of the named file into the persistent C process. The extension ".mc" will be automatically added and should not be explicitly given.

# 2  Universal Syntax

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 a syntax with the following three "universal" properties.

1. Universal syntax trees are semantics-free. They are simply trees viewed as representations of character strings.

2. The universal syntax reader can read any parenthesis-balanced character string. Here parenthesis-balanced means that every open parenthesis, brace or bracket has a matching closing character and that string quotations are properly closed.

3. The printer inverts the reader. If we read a string $s$ into a universal syntax expression $e$ and then print $e$ back into a string $s'$ we have that $s$ and $s'$ are equivalent in a strong sense. Here we want $s$ and $s'$ to be "whitespace equivalent" so as to be treated equivalently by the C lexer.

The emphasis here is on the representation of strings. The reader does not always invert the printer — the expression ⟨⟨one two⟩ three⟩ prints as one two three which reads as ⟨one ⟨two three⟩⟩. But the represented string is preserved. This is fundamentally different from most programming languages supporting symbolic computation (such as Lisp) where it is assumed that the tree structure, rather than the represented string, is fundamental and hence that the reader should invert the printer.

The above universality properties allow one to assign semantics to universal syntax 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 universal syntax 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 universal syntax expressions recovers substructure that is semantically compositional under C semantics. Parentheses and semicolons can be helpful in aligning universal syntax trees with C semantics.

## 2.1  Universal Expressions: cons-car-cdr

A universal syntax expression is either an atom (a wrapper around a string), a pair ⟨$e_1 e_2$⟩ where $e_1$ and $e_2$ are expressions, or a "parenthesis expression" the form $(e)$, $\{e\}$, or $[e]$ where $e$ is an expression. All three of these datatypes

have the same C type `expptr` (Expression Pointer). Each has a constructor procedure, a predicate, and accessor functions. The constructor, predicate and accessors for atoms, pairs and parenthesis expressions are the following. The atom data type has the constructor, test and accessor function for atoms are the following.

```
expptr string_atom(char *);
int atomp(expptr);
char * atom_string(expptr); //the argument must be an atom.

expptr cons(expptr,expptr);
int cellp(expptr);
expptr car(expptr); //the argument must be a cell. returns the first component.
expptr cdr(expptr); //the argument must be a cell. returns the second component.

expptr intern_paren(char,expptr); // the char must be one of '(', '{' or '['
int parenp(expptr);
expptr paren_inside(expptr);
```

## 2.2   Universal Expressions: Phantom Brackets

The MetaC reader maps a character string to a universal syntax expression. A specification of the reader is given in section 2.3. The expression produced by the reader can be represented by "phantom brackets" around the given string where there is a pair of brackets for each cons cell. For example the expression {one two three} reads as $\{\langle$one $\langle$two three$\rangle\rangle\}$. Examples of strings and the phantom bracking imposed by the reading those strings is given in figure ref-fig:reader. The printer simply removes the phantom brackets and prints the string that the expression represents.

To reduce the clutter of brackets we will adopt two conventions for supressing brackets when exhibiting phantom brackets. The first convention is the Lisp right-branching convention of not showing all the cell brackets for right-associative (right-branching) sequences. For example $\langle$zero $\langle$one $\langle$two three$\rangle\rangle\rangle$. will be written as $\langle$zero one two three$\rangle$. Note that these "lists" are atom-terminated rather than the Lisp convention of nil termination.

The second convention is that we will sometimes write $\langle\langle a_1 \ o\rangle \ a_2\rangle$ where $o$ is a connective atom as the left-branching structure $\langle a_1 \ o \ a_2\rangle$. For example, the expression {a + b} reads as $\{\langle\langle$a $+\rangle$ b$\rangle\}$ which is then abbreviated as $\{\langle$a $+$ b$\rangle\}$. Left-branching for binary connectives gives a C-consistent treatment of semi-colon as a binary connective while also supporting the interpretation of semi-colon as a statement terminator. This is discussed in more detail below.

It will generally be clear from context whether we using the Lisp right-branching

$$
\begin{array}{rcl}
\texttt{Hello World} & \Rightarrow & \langle\texttt{Hello World}\rangle \\
\texttt{one two three} & \Rightarrow & \langle\texttt{one } \langle\texttt{two three}\rangle\rangle \\
 & = & \langle\texttt{one two three}\rangle \\
\texttt{x + y} & \Rightarrow & \langle\langle\texttt{x +}\rangle\texttt{ y}\rangle \\
 & = & \langle\texttt{x + y}\rangle \\
\texttt{x + y * z} & \Rightarrow & \langle\texttt{x + }\langle\texttt{y * z}\rangle\rangle \\
\texttt{(x + y) * z} & \Rightarrow & \langle(\langle\texttt{x + y}\rangle)\texttt{ * z}\rangle \\
\texttt{foo(int x)} & \Rightarrow & \langle\texttt{foo }(\langle\texttt{int x}\rangle)\rangle \\
\texttt{foo(int x, float y)} & \Rightarrow & \langle\texttt{foo }(\langle\langle\texttt{int x}\rangle, \langle\texttt{float y}\rangle\rangle)\,\rangle
\end{array}
$$

Figure 1: **Examples of Reader Bracketings.** Bracketings are shown for the expression that results form reading the given strings. A complete bracketing shows a pair of brackets for every expression pair (cons cell). The second and third example show two somewhat informal conventions for dropping some of the brackets — general sequences are assumed to be right-associative and binary connective applications are assumed to be left-associative. These conventions are used in other examples. Expressions are printed without the brackets — the brackets are "phantoms" that show the tree structure. The bracketing (parsing) done by the reader is specified formally in section 2.3. Since the bracketing does not affect the represented string, the precise bracketing is often unimportant.

list convention or the left-branching connective convention. A set of examples of strings and the phantom brackets generated by the reader is given in figure 1.

To emphasize the significance of left-branching binary connectives we note that the MetaC reader bracketing of

$$\{e_1 \; ; \; e_2 \; ; \; e_3 \; ; \; e_4\}$$

can be written using either the left-branching binary connective convention as

$$\{\langle e_1 \; ; \; \langle e_2 \; ; \; \langle e_3 \; ; \; e_4\rangle\rangle\rangle\}$$

or the right-braching sequence convention as

$$\{\langle\langle e_1 \; ;\rangle \langle e_2 \; ;\rangle \langle e_3 \; ;\rangle e_4\rangle\},$$

both of which abbreviate the same full bracketing

$$\{\langle\langle e_1 \; ;\rangle \langle\langle e_2 \; ;\rangle \langle\langle e_3 \; ;\rangle e_4\rangle\rangle\rangle\}.$$

## 2.3  The Reader Specification

The MetaC reader can be described in three phases — preprocessing, lexicalization, and parsing.

The MetaC preprocessor removes C-style comments and divides files into segments where each segment is to be read as a separate expression. 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 file must be parenthesis-balanced within each segment. For the REPL a segment ends at the first return character not occurring inside parentheses.

Lexicalization segments a pre-processed character sting into to a sequence of atoms. The MetaC lexer preserves all non-white characters. For the MetaC lexer each atom is one of the following.

- Symbols, 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 `foo_bar1`.

- Connectives. A connective is a character string consisting of only "connective characters" — the connective characters consist of all non-alphanumeric, non-white, non-parenthesis, characters other than the three special characters `$`, ` and `\`.

- Quoted strings. A character string starting and ending with the same string quotation character. For example `"foo bar"`, `"!:&?;"` or `'a'`.

- Single character atoms for parenthesis characters and the three special characters `, `$`, and `\`.

Two strings will be called whitespace-equivalent if they lexicalize to the same sequence of atoms. Two strings that lexicalize equivalently under the MetaC lexer will also lexicalize equivalently under the C lexer.

The reader is specified by a the grammar shown in figure 2. To simplify the specification of the reader we enclose the given string in parentheses so that, without loss of generality, the input is assumed to be of the form $\{s\}$ where $s$ is the lexical sequence to be read. The grammar has a nonterminal $P$ for parenthesis expression which we take as the top level nonterminal. The nonterminals SYM and QUOTE range over alpha-numeric string atoms and quoted string atoms respectively. For each positive integer $p$ we have a nonterminal $\text{CONN}_p$ ranging over connective atoms of precedence $p$. We have a nonterminal $E_p$ for expressions formed with connectives of precedence $p$. We also has a nonterminal $E_\infty$ for expressions that are formed at higher precedence than any connective expression. The precedence levels are divided into a set $\mathcal{L}$ of left-associative

$$
\begin{aligned}
P &::= (E_p) \mid \{E_p\} \mid [E_p] \\
E_p &::= \langle\langle E_\ell \ \mathrm{CONN}_p\rangle\ E_r\rangle \quad p \in \mathcal{R},\ \ell > p,\ r \geq p \\
E_p &::= \langle\langle E_\ell \ \mathrm{CONN}_p\rangle\ E_r\rangle \quad p \in \mathcal{L},\ \ell \geq p,\ r > p \\
E_3 &::= \langle E_\ell \ E_r\rangle \quad \ell > 3,\ r \geq 3 \\
E_\infty &::= \mathrm{QUOTE} \mid \langle S\ \ P^*\rangle \mid \langle\text{'}\ P\rangle \mid P \mid \epsilon \mid \mathrm{JUNK} \\
S &::= \mathrm{SYM} \mid V \\
V &::= \langle\$\ \mathrm{SYM}\rangle \mid \langle\$\ P\rangle \mid \langle\backslash\ V\rangle \\
P^* &::= \epsilon \mid \langle P\ P^*\rangle \\
\mathrm{JUNK} &::= \text{'} \mid \mathrm{SJUNK} \\
\mathrm{SJUNK} &::= \$ \mid \backslash \mid \langle\backslash\ \mathrm{JUNK}\rangle
\end{aligned}
$$

Figure 2: **The grammar defining the MetaC reader.** The input string is assumed to be of the form $(s)$ so that endpoints are explicitly labeled. The nonterminal SYM generates non-empty alpha-numeric strings; $\mathrm{CONN}_p$ generates non-empty connective strings of precedence $p$; and QUOTE generates string quotations. $\mathcal{R}$ is a set of right-associative precedence levels and $\mathcal{L}$ is a set of left-associative levels. Only the highest precedence level is left-associative. MetaC classifies all printable ASCII characters as being either alpha-numeric, connective, string quotations, parenthesis characters or one of the three special characters ', \$ or \\. The reader will accept any parenthesis-balanced string. $\epsilon$ denotes the empty string. Generated cells of the form $\langle w\ \epsilon\rangle$ or $\langle \epsilon\ w\rangle$ are replaced by $w$. Although the grammar is ambiguous, the reader is deterministic as specified by the constraints that $E_\infty$ expressions must be maximal and that the use of the $\epsilon$ production for $E_\infty$ must be minimized. See the text for details.

precedence levels and a set $\mathcal{R}$ of right associative precedence levels. There is a special precedence level 3 for combining expressions with the "null connective". The null connective is left-associative. The null connective is higher precedence that semicolon and comma but lower precedence than all other connectives. The nonterminal $E_\infty$ includes an epsilon production allowing null arguments.

The nonterminal $S$ ranges over symbols or variables where $V$ ranges over variables. Variables are handles specially in pattern matching (`ucase`) and pattern instantiation (backquote). Expression of the form $\langle S\ P^*\rangle$ include `foo`, `foo(x)`, `foo(int x){return x;}`, `$x`, `$f(x)`, and `${f(x)}`.

The above grammar is ambiguous. For example the string `{foo (a) (b)}` can be parsed as either the left-branching structure $\{\langle\langle \mathrm{SYM}\ P^*\rangle\ P\rangle\}$ or the right-branching structure $\{\langle \mathrm{SYM}\ P^*\rangle\}$. The reader is of course deterministic — `{foo (a) (b)}` is read as right-branching. While a deterministic grammar for the

16

reader can be given, it is simpler to specify a deterministic parsing process for the above ambiguous grammar. The implementation runs a deterministic left-to-right shift-reduce process. However, the parser is easier to specify as operating in global stages. In the first stage one identifies all maximal $E_\infty$ substrings. The maximal $E_\infty$ requirement specifies that {foo (a) (b)} is parsed as a single (right-branching) $E_\infty$ expression. However the string {$ foo (a) (b)} is read as the longer $E_\infty$ expression {$\langle\langle$! foo$\rangle$ $\langle$(a) (b)$\rangle\rangle$}. The grammar does have the property that any given string has a unique segmentation into maximal $E_\infty$ expressions each of which has a unique structure.

After identifying maximal $E_\infty$ expressions we are left with a sequence of arguments (the $E_\infty$ expressions) and connectives. However, it is possible that this sequence contains multiple consecutive connectives or multiple consecutive arguments. We then place an argument between any two consecutive connectives using the $\epsilon$ production for $E_\infty$ and also add an $\epsilon$ term at the beginning of the sequence if the sequence starts with a connective and an epsilon argument at the end if the sequence ends in a connective. We now have a sequence of arguments and connectives starting and ending with arguments and where there are no two consecutive connectives. Next set $k$ to be the largest precedence of any connective and form all the $E_k$ substrings. We then repeatedly decrement $k$ and identify all the $E_k$ substrings until we have identifies all $E_p$ substrings for all $p$. At this point the entire string must be parsed as $E_p$ where $p$ is the smallest precedence of the connectives where we think of adjacent arguments as having an implicit precedence 3 connetive between them.

This process can parse any parenthesis-balanced string.

The above specification of the reader can result in cells of the form $\langle e\ \epsilon\rangle$ or $\langle \epsilon\ e\rangle$. The implementation avoids producing such cells and the reader can be viewed as replacing $\langle e\ \epsilon\rangle$ or $\langle \epsilon\ e\rangle$ by $e$ so that all cells are of the form $\langle e_1\ e_2\rangle$ where $e_1$ and $e_2$ are expressions representing non-empty strings. An empty parenthesis string () is read as a parenthesis expression containing an atom for the empty string.

Grouped by precedence, the binary connective characters are

$$\{\texttt{;}\}\ \{\texttt{,}\}\ \{\texttt{|}\}\ \{\epsilon\}\ \{\texttt{\&, ?, !}\}\ \{\texttt{= } \sim, <, >\}\ \{\texttt{+, -}\}\ \{\texttt{*, /}\}\ \{\texttt{\%, \^{}, .}\}\ \{\texttt{:, @, \#}\}.$$

The precedence of a binary connective is determined by its first character. The characters have low to high (outer to inner) precedence in the order given with symbols in the same group having the same precedence. Here $\epsilon$ represents the null connective at precedence level 3. The precedence can be summarized in outer to inner order as semicolon, comma, bar, the null connective, Boolean connectives, binary predicates, three levels of binary functions, and innermost connectives which can be viewed as providing a way of building structured atoms. This universal syntax precedence conventions has various divergences with C syntax. However, when writing a macro package in MetaC one can simply adopt the MetaC precedence conventions for the source code and use

parentheses where necessary in the generated C code.

## 2.4 Backquote and Pattern Matching Revisited

The semantics of backquote is defined in terms of cons-car-cdr view of expressions independent of the MetaC reader. In the typical case, the C value of a backquote expression $\langle ' \ \{e\}\rangle$ is the expression (tree) $e$ with subexpressions (subtrees) of the form $\langle \$ \ x\rangle$, where $x$ is a symbol, replaced by the C value of $x$ and subexpressions (subtrees) of the form $\langle \$ \ \{w\}\rangle$ replaced by the expression which is the C value of the string represented by the expression $w$.

Unfortunately this 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, MetaC supports nested backquotes. 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 symbol $ can be included in the value of a backquote expression 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). We can even have multiple layers 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).

As with backquote, pattern matching is defined in terms of the cons-car-cdr view of expressions independent of the MetaC reader. We define a substitution to be a mapping from symbol atoms to expressions. For a substitution $\sigma$ and a pattern $\{e\}$ we define the expressions $\sigma(e)$ to be the result of replacing each subexpression (subtree) of $e$ of the form $\langle \$ \ x\rangle$ where $x$ is a symbol atom with $\sigma(x)$. A pattern expression (tree) $\{e\}$ matches an expression (tree) $w$ with substitution $\sigma$ if $\sigma(e) = w$. An exception to this is the case where a variable occurs multiple times in the pattern. In release 1.0 multiple occurrences of a variable in a pattern is not supported.

# 3  Miscellaneous Features

## 3.1  Interning and Expression Properties

In MetaC expressions are interned and have property lists. Interning means that
two expressions which have the same tree structure and sequence of leaves —
that are "equal" in Lisp terminology – represented by the same data structure
with the same memory address — are "eq". Hence we have

```
MC> int_exp('{foo(a)} == '{foo(a)})
1
```

Interned expressions with properties provide the functionality of dictionary data
structures such as C++ hashmaps or Python dictionaries. Interning also facili-
tates the implementation of forward chaining inference algorithms — we do not
want to waste noticing the same consequence over and over again. Interned
expressions with property lists can be viewed as a form of no-SQL database. In-
terning also allows directed acyclic graphs (dags) to be represented compactly as
expressions. This is very convenient for the representation of machine-generated
justifications (proofs).

The procedure `setprop(expptr exp, expptr prop, void * value)` sets the
given property of the given expression to the given value. The procedure
`getprop(expptr exp, expptr prop, void * default)` the given property of
the given value or the default value if no property value has been assigned.
MetaC also provides the procedures `setprop_int(expptr e, expptr p, int
n)` and `getprop(expptr e, expptr p, int default)`. It is not difficult to
define polymorphic macros for setting and extracting properties where the type
of the property is passed as an argument. It is also not difficult to assign types
to particular properties so that the type argument is not needed in polymor-
phic property setting and retrieving. However, these features are not present in
release 1.0.

## 3.2  Memory Frames

A MetaC memory frame is analogous to a stack frame but is independent of the
C stack. The procedure `push_memory_frame()` pushes a memory frame. The
procedure `stack_alloc(int nbytes)` returns a pointer to a block of the given
size allocated from the current MM-frame. The procedure `pop-memory-frame()`
pops the current memory-management frame freeing all allocation done in that
frame by resetting the appropriate free pointer. MetaC pre-allocates the heap
used by `stack_alloc`.

## 3.3   Undo Frames

Undo frames are similar to memory-frames but with the added feature of undoing effects tied to the frame. The procedure `push_undo_frame();` enters a new undo frame. Memory is allocated from the current undo frame with the procedure `undo_alloc(int nbytes)`. Effect are tied to the current undo frame with the C preprocessor macro `undo_set(void * loc, void * val)`. On exit from an undo frame all memory locations assigned by `undo_set` are restored to the values they had on entry to the frame. In MetaC all expression allocation and property assignments are tied to the current undo frame. On exit from the undo frame the database represented by expression properties is restored to the state that existing on frame entry.

The procedure `value_from_undo(expptr exp)` exits the current undo frame returning the given expression as a value. This is implemented by first pushing a new memory-frame, copying the give expression value into the new memory-frame; poping the current undo frame; copying the returned expression back into undo memory, and then popping the memory-frame holding the expression. These transfers are all done in time proportional to the dag size of the expression returned.

## 3.4   Bootstrapping and File Expansion

MetaC is bootstrapped. The makefile for MetaC includes the following.

```
mcB.c : expandA mcB.mc
 ./expandA mcB.mc mcB.c
```

The executable `expandA` is implemented entirely in $C$ but implements the backquote macro. The file `mcB.mc` implements `ucase` using backquote. The makefile also includes.

```
mcC.c : expandB mcC.mc
 ./expandB mcC.mc mcC.c
```

Here the file `mcB.mc` implements additional macros using both backquote and ucase. In the MetaC makefile this is continued up to mcE.mc and expandE. MetaC provides the procedure `mcexpand(char * f1, char * f2)` which is used in the expandA and expandB commands above. This procedure macro-expands the file `f1` and writes the result to file `f2`. This macro-expansion is done using whatever procedures and operators currently have macro procedure pointers on their property lists, i.e., the expansion is done using whatever macros are defined in the current state.

The procedure `mcexpand(char * f1, char * f2)` is implemented using the MetaC procedure `file_expressions(char * f)`. This procedure takes a file name and returns a list of the expressions contained in that file.

## 3.5   Macro Expansion Effects

Macros expansion can also have effects. The macro `umacro` expands to a procedure definition of the macro expansion code but also generates a statement which installs that procedure pointer in the macro property of the head symbol. During macro expansion the MetaC primitive `add_init_form(expptr statement)` is called with a statement that installs the procedure pointer on the appropriate property list.

In file expansion the initialization forms generated during macro expansion are incorporated into an initialization procedure. The macro `init_fun(<fname>)` macro expands to a definition of the given procedure name with no arguments but with the sequence of initialization forms in its body. In the REPL and IDE the intialization forms are run after the macro expansion of the input or cell is run.

## 3.6   Preliminary Definitions for Macros

A macro can also generate a preliminary definition. It is possible to implement syntactic closures (lambda expressions) with a macro. A closure consists of a procedure pointer bundled up with values for the free variables in the body of the lambda expression. The lambda macro must first create the procedure that executes the body of the lambda expression and then incorporate that procedure pointer into the expression for constructing the closure data structure. The construction of the procedure definition is a "preamble" to expression returned as the expansion of the macro. MetaC provides the primitive `add_preamble(expptr definition)` for creating a definition of a procedure used in the expression returned by the macro. In file expansion these preamble definitions are inserted into the output file before the expression generated by the macro. In the REPL and NIDE these preamble definitions are installed along with the expression created by the macro.

The MetaC procedure `full_macro_expansion(expptr e)` returns a list consisting of the preamble expressions, the result of the macro expansion, and the initialization forms.

# 4 The MetaC Primitives

We now give a list of the MetaC primitives and pre-installed type definitions. The macros backquote, ucase, and umacro expand to code built on these primitves. We start with type definitions needed for the single atom type restriction needed for the limiting type parsing abilities of MetaC.

```
typedef char * charptr;

typedef void * voidptr;

typedef FILE * FILEptr;
```

Next we have the lowest level interface functions for expressions. The macros `backquote` and `ucase` expand into C code using these procedures.

```
typedef struct expstruct{...}*expptr;

expptr string_atom(char * s);

int atomp(expptr e);

charptr atom_string(expptr a);

expptr cons(expptr x, expptr y);

int cellp(expptr e);

expptr car(expptr x);

expptr cdr(expptr x);

expptr intern_paren(char openchar, expptr arg); \\ openchar must be one of '(', '{' or '['.

int parenp(expptr e);

expptr paren_inside(expptr e);

char constructor(expptr e); //used for paren expressions.
```

We also have integer-expression conversions. The REPL and NIDE require that inputs and cells are expression-valued.

```
expptr int_exp(int i);

int exp_int(expptr s);
```

Next we have expression properties. The macro umacro expands to code that sets the macro property of some atom to a procedure pointer. Gensym is also important in macro definitions.

```
void setprop(expptr e, expptr key, expptr val);

expptr getprop(expptr e, expptr key, expptr defaultval);

void setprop_int(expptr e, expptr key, int val);

int getprop_int(expptr e, expptr key, int defaultval);

expptr gensym(char * s);
```

We also have error exceptions and break points.

```
void berror(charptr s);

void breakpt(charptr s);

void return_to_NIDE();
```

We now consider reading and printing. The procedure file_expressions takes a file name and returns a list of the expressions (in universal syntax) contained in the file. The last two print to the *Messages* buffer in emacs in the NIDE. The last expressions is a macro.

```
expptr file_expressions(char *name);

void pprint(expptr e, FILEptr f, int indent); //indent is typically 0

void mcpprint(epptr e);

void mcprint(...)
```

We have the following two macro-expansion procedures. The first does full macro expansion — the result is guaranteed not to contain macros. The second does one step of macro expansion — the result may contain more macros.

```
expptr macroexpand(expptr e);

expptr macroexpand1(expptr e); //this result may contain unexpanded macros
```

We also have the following list-processing procedures and macros.

```
expptr append(expptr l1, expptr l2);

expptr reverse(expptr l);

typedef   expptr exp_to_exp(expptr);
typedef   void exp_to_void(expptr);

expptr mapcar(exp_to_exp f, expptr l);

void mapc(exp_to_void f, expptr l);

int length(expptr l);
```

We have the procedures related to memory frames.

We have the procedures related to undo frames.

```
void push_undo_frame();

void return_from_undo(expptr e);

void undo_set(voidptr loc, voidptr val);

voidptr undo_alloc(int size);
```

We also have the procedures related to file expansion.

```
file_expressions(char * fname);

mcexpand(char * fname1, char * fname2);

init_fun(fname)
```

Finally we have lower level macro expansion procedures.

```
add_init_form(expptr stement);
```

```
add_preamble(expptr definition);

full_expansion(expptr e);
```

# References

[1] A. Bawden. Quasiquotation in lisp. In *Proceedings of the 1999 ACM SIG-PLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA, January 22-23, 1999. Technical report BRICS-NS-99-1*, pages 4–12, 1999.