MetaC 1.0

David McAllester

September 19, 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 compiling and executing C statements and incremental procedure definitions in a persistent C process. This is done by compiling and loading dynamic load libraries.

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).

Acknowledgement: I would like to thank Bob Givan for his aid in exercising and polishing MetaC and his continuing efforts on MathZero.

Contents

1 Introduction and Overview

1.1 Installation

To install the REPL:

1. Clone the MetaC repository in a local MetaC directory.
2. Edit the last line of the file mcA.c to give your MetaC directory.
3. From a shell running in your MetaC directory type `make MC`.
4. From the same shell type `MC` to get the command line prompt `MC>`.

To install the NIDE:

1. Clone the MetaC repository in a local MetaC directory.
2. Edit the last line of the file mcA.c to give your MetaC directory.
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. From a shell running in your MetaC directory type `make NIDE`.
7. Start or restart Emacs.[1]

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-z s` (control-z followed by s) to start (or restart) the C kernel. Typing `C-z x` 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] If you load mc.el into a running emacs MetaC will not be available in buffers created before the load.

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 a MetaC universal syntax expression which prints as `hello world`. MetaC universal syntax expressions are 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 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
```

Note that the last procedure has output type `expptr`. This is the type of the value returned by a backquote expression — the type of MetaC universal syntax expressions.

```
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 restriction ensures meaningful C type checking in the presence of dynamic linking.

1.5 The Global Variable Array Restriction

To simplify dynamic 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
```

MetaC treats `int y[0] = 2;` as equivalent to

```
MC> int y[1];

done

MC> y[0] = 2;

done
```

We can continue with

```
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 — assignments 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.

1.8 Pattern Matching

MetaC provides a pattern matching case construct `ucase` (universal case). The basic 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
```

The last clauses of the last procedure uses a condition feature where a pattern can have the form {<pattern>}.(<condition>) where <condition> is a C expression which must be non-zero in order for the clause to be selected.

```
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*$y` will not match the expression `a+b*c` because the reader brackets `a+b*c` as ⟨`a` +⟨`b` ∗ `c`⟩⟩. The pattern `$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 occurrence 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{dolist($x,$L){$body}}{
```

```
   expptr rest=gensym("rest");
   return
    `{for(expptr $rest = $L; cellp($rest); $rest = cdr($rest);){
      expptr $x=car($rest);
      $body}};
}

done

MC>macroexpand(`{dolist(item,list){f(item);}})

for
  (expptr _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 `dolist` defines a procedure to compute the macro expansion and installs that procedure on a macro property of the symbol `dolist`. 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 macro can also be attached to a binary connective. For the macro pattern {`$x .> $y`}. the macro is attached to the two-character binary connective .>.

It is sometimes useful to see the result of a single step of macro expansion. The procedure `macroexpand1(expptr)` returns the result of a single step of macro expansion on the root of the given argument.

1.10 The Notebook IDE

Starting the kernel with the key C-z s creates a running C process. When a cell is executed with the command C-z x the text of the cell is passed to the C process from which a value is returned and then printed 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-z 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.

Cell Boundaries. A cell begins at any nonempty line whose first character is other than white space (space or tab), a close character ')', '}', or ']', or a slash '/'. The cell ends at the beginning of the next non-empty line whose first character is other than white space or a close character or at the end of the file if no such line exists.

Note that a comment starting at the beginning of a line terminates a cell.

Errors and Breakpoints.

If the cell contains unbalanced parentheses, or non-ascii unicode characters, the NIDE will print `"reader error"` as the value of the cell and display an error message.

If an error occurs while macro-expanding the cell expression, the NIDE will print `"macro expansion error"` as the value of the cell and enter the gdb debugger. Such an error may be due to a bug in a user-written macro and gdb can be useful in this case. Typing the continue command, which can be abbreviated as just `"c"`, will abort the cell execution and return to the NIDE.

If an error occurs while attempting to compile the macro-expanded cell, the NIDE will print `"compilation error"` as the value of the cell and display the compiler errors.

If an error is trapped by dynamic checking while executing the compiled code for the cell, the NIDE will print `"dynamic-check execution error"` as the value of the cell and enter the gdb debugger. Typing the continue command to gdb will abort execution and return to the NIDE. The procedure `berror(char *)` can be called by the user to declare a dynamic-check error.

If an error is trapped directly by gdb during execution, for example a segmentation fault in user code, the NIDE will print `"gdb-trapped execution error"` as the value of the cell and enter the gdb debugger. In this case the continue command is ineffective and one must type `p NIDE()` to return to the NIDE.

The MetaC procedure `breakpt(char *)` can be used for breakpoints from which execution can be continued. When `breakpt` is called the NIDE will invoke the debugger with no error message printed in the NIDE but with the string passed to `breakpt` printed in the first line in the debugger window. The continue command will continue execution from the breakpoint.

MetaC is designed to minimize the risk of memory corruption when returning to the NIDE from an error. See the discussions of unwind protection in sections 2.2 and 2.3. If there is a concern over memory corruption one can always 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 procedure `mcpprint(expptr e)` will pretty-print the expression e to the *Messages* buffer. When in the gdb debugger entering `p pp(<exp>)` will pretty print the given expression.

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

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

C-z x executes the cell containing the cursor.

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

C-z b executes the entire buffer

C-z B executes the buffer up to the current cursor position.

C-z a moves to the beginning of the current cell.

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

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

C-z c removes all cell values. This is useful before a git commit as it can reduce or eliminate file differences.

Load Statements. A cell in a code file containing `load(<filename>)` (with no semicolon) will load the cells of the named file (which should be a string constant) into the persistent C process. The extension .mc will be automatically added and should not be explicitly given. Load statements can also be given to the REPL to load (and link) C files into the running REPL process.

2 Miscellaneous Features

2.1 Interning and 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 are represented by the same data structure with the same memory address. This is sometimes called "hash-consing" and can be expressed in Lisp terminology by saying that "equal" implies "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 facilitates the implementation of forward chaining inference algorithms — we do not want to waste time noticing the same consequence over and over again. Interned expressions with property lists can also be viewed as a form of no-SQL database. Interning 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)` returns 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_int(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 polymorphic property setting and retrieving. However, these features are not present in release 1.0.

2.2 Memory Frames

A memory frame is analogous to a stack frame but is only loosely tied to the C stack. The macro `in_memory_frame(<statement>)` will execute the given statement inside a new memory frame. The procedure `stack_alloc(int nbytes)` returns a pointer to a block of the given size allocated from the current memory frame. When `in_memory_frame(<statement>)` exits the memory-frame free pointer (or "stack pointer") is reset. This frees all memory allocated from the frame. This architecture differs from traditional stack allocation in

that recursive procedures can operate with a single memory frame and return newly allocated memory as values without copying. The allocated storage will not be deallocated until exit from an enclosing use of `in_memory_frame`. MetaC pre-allocates the heap (stack space) used for memory frames. The deallocation (resetting of the freeptr) is unwind protected — it will occur when an error is thrown beyond the frame entry point. This avoids a memory leak when returning from the an error to the REPL or NIDE.

MetaC clears the base memory frame after every evaluation in the REPL and after every cell evaluation in the NIDE. It is therefore important not to be using stack allocated memory between evaluations.

2.3 Undo Frames

Undo frames are similar to memory frames but with the added feature of undoing effects tied to the frame. The macro `exp_from_undo_frame(<expression>)` creates a new undo frame and executes the given expression in the new frame. Memory is allocated from the current undo frame with `undo_alloc(int nbytes)`. Effects 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 allocated from that frame is reclaimed by resetting a free pointer and all locations assigned by `undo_set` are restored to the values they had before 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 existed on frame entry.

The macro `exp_from_undo_frame(<expression>)` computes the value of the given expression in a new undo frame, then pops that undo frame while copying the computed value into the parent undo frame. The value must be a universal syntax expression (an `expptr`). This is used for returning the "value" or "result" of a large computation while freeing the allocation and undoing the effects of the computation. The macro `exp_from_undo_frame(<expression>)` has unwind protection so that the undo frame is popped (with memory deallocated and effects undone) if an error is thrown in the execution of the expression.

The procedure `clean_undo_frame(expptr e)` is similar to `exp_from_undo_frame` but replaces the current undo frame with a fresh one into which the given expression is copied. In this case there is no allocation from the parent frame. The procedure `clean_undo_frame(expptr e)` can be used for garbage collection as in `(install_live_stuff(clean_undo_frame(compute_live_stuff())))`.

The procedures `exp_from_undo_frame` and `clean_undo_frame` both use expression copying that runs in time proportional to the dag size of the expression. The dag size can be exponentially smaller than the tree size.

`exp_from_undo_frame` is a macro that expands to a C expression. Recall that C expressions have values while C statements do not. Macros that expand to expressions (with values) typically expand to compound expressions of the form ({... <var>;}) where the value returned is the value of the final variable after executing the previous statement. Compound expressions are a GNU C extension.

The base undo frame is not cleared between evaluations. The base frame is holding properties needed for the proper functioning of MetaC. The base frame should not be manually cleared. It is not difficult to write a collector for collecting the state used by the MetaC system which could then be integrated into a user-written state collector for garbage collecting the base frame. However MetaC 1.0 does not support garbage collecting the base undo frame. One can always restart the C process.

2.4 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 executable `expandA` expands backquote expressions appearing in the input file. The file `mcB.mc` implements `ucase` using backquote and the excutable expandB expands both backquote expressions and ucase statements. The makefile also includes.

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

Here the file `mcC.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 code for the expand commands. This procedure macro-expands the file `f1` and writes the result to file `f2`. The expansion is done using whatever macros are defined in the current state — the expansion uses whatever procedures and operators currently have macro expansion procedure pointers on their property lists.

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 the cells of that file.

2.5 Macro Effects

Macros expansion can 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. The procedure `add_init_form` is provided to the user for use in defining complex macros requiring this feature.

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 initialization forms are run before the macro expansion of the input or cell is run.

2.6 Macro Preliminary Definitions

Macro expansion can also generate preliminary definitions. It is possible to implement syntactic closures (lambda) as a macro. A closure consists of a procedure pointer together with values for the free variables 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 expansion of the lambda macro. The construction of the procedure definition is a "preamble" to the macro expansion. MetaC provides the primitive `add_preamble(expptr definition)` for creating preambles during macro expansion. 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. MetaC 1.0 supports preambles but does not provide a macro for lambda expressions.

3 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 $\langle\langle$`one two`$\rangle$ `three`$\rangle$ prints as `one two three` which reads as $\langle$`one` $\langle$`two three`$\rangle\rangle$. 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 tree structure. 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. For languages and frameworks implemented as macro packages in MetaC one can guarantee that the universal syntax tree structure has compositional semantics.

3.1 Universal Expressions: cons-car-cdr

A universal syntax expression is either an atom (a wrapper around a string), a pair $\langle e_1 e_2 \rangle$ 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). For each of the types atom, pair and parenthesis expression there is a constructor procedure, a predicate, and accessor functions as follows.

```
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);
```

3.2 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 3.3. 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 {⟨`one` ⟨`two three`⟩⟩}. Examples of strings and the phantom bracking imposed by the reading those strings is given in figure 1. 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 ⟨`zero` ⟨`one` ⟨`two three`⟩⟩⟩. will be written as ⟨`zero one two three`⟩. Note that these "lists" are atom-terminated rather than the Lisp convention of nil termination.

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

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-braching 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

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

```
Hello World          ⇒  ⟨Hello World⟩
one two three        ⇒  ⟨one ⟨two three⟩⟩
                     =  ⟨one two three⟩
x + y                ⇒  ⟨⟨x +⟩ y⟩
                     =  ⟨x + y⟩
x + y * z            ⇒  ⟨x + ⟨y * z⟩⟩
(x + y) * z          ⇒  ⟨(⟨x + y⟩) * z⟩
foo(int x)           ⇒  ⟨foo (⟨int x⟩)⟩
foo(int x, float y)  ⇒  ⟨foo (⟨⟨int x⟩ , ⟨float y⟩⟩) ⟩
(foo (int x, float y) bar)  ⇒  (⟨⟨foo (int x, float y)⟩ bar⟩)

(foo b c)            ⇒  (⟨foo b c⟩)
(foo (b) c)          ⇒  (⟨⟨foo (b)⟩ c⟩)
(foo (b) (c) d e)    ⇒  (⟨⟨foo (b) (c)⟩ d e⟩)

($ f b c)            ⇒  (⟨⟨$ f⟩ b c⟩)
($ f (b) (c) d e)    ⇒  (⟨⟨⟨$ f⟩ (b) (c)⟩ d e⟩)
($ {f(x)} (b) (c) d e)  ⇒  (⟨⟨⟨$ {f(x)}⟩ (b) (c)⟩ d e⟩)
```

Figure 1: **Examples of Reader Bracketings.** Bracketings are shown for the expression that results from 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 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 the other examples. Expressions are printed to files, the REPL or the NIDE without the brackets — the brackets are "phantoms" that show the tree structure. Because `f(x)` in generally interpreted as (high-precedence) application, it is best to avoid the Lisp convention of using space as a list connective. When designing a syntax for lists it is better to use semicolon or comma as in `(f,(x),b)`. The bracketing (parsing) done by the reader is specified formally in section 3.3.

both of which abbreviate the same full bracketing

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

3.3 The Reader Specification

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

The MetaC preprocessor replaces each C-style comment with a space character. When processing an entire file, as is done in the MetaC procedure `file_expressions` described in section 2.4, the preprocessor divides the file into cells using the same conventions as is used in the NIDE. A file must be parenthesis-balanced within each cell.

Lexicalization segments a pre-processed character string into a sequence of atoms. The MetaC lexer preserves all non-white characters. For the MetaC lexer each atom is 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 `foo_bar1`.
- A connective. A connective is a character string consisting of only "connective characters". Grouped by precedence, the connective characters are

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

  The characters are listed from low precedence (weakly binding) to high precedence (strongly binding). A connective —a string of connective characters — has the precedence of its first character. The "null connective" ϵ intuitively represents the space character used as a connective. The null connective and the null argument are discussed below.
- A quoted string. A character string starting and ending with the same string quotation character. For example `"foo bar"`, `"!:&?;"` or `'a'`.
- A special character atom. These are atoms whose strings are one character long where that character is one of the special characters `‘`, `$`, and \.

Two strings will be called whitespace-equivalent if they lexicalize to the same sequence of atoms.

The reader is specified by the grammar shown in figure 2. To simplify the specification of the reader we enclose the given string in parentheses so that, without

$$
\begin{array}{rcl}
P & ::= & (E_p) \mid \{E_p\} \mid [E_p] \\
E_p & ::= & \langle E_\ell \ \mathrm{CONN}_p \ E_r \rangle \quad p \in \mathcal{R},\ \ell > p,\ r \geq p \\
E_p & ::= & \langle E_\ell \ \mathrm{CONN}_p \ E_r \rangle \quad p \in \mathcal{L},\ \ell \geq p,\ r > p \\
E_4 & ::= & \langle E_\ell \ E_r \rangle \quad \ell > 4,\ r \geq 4 \\
P^* & ::= & \epsilon \mid \langle P \ P^* \rangle \\
E_\infty & ::= & \mathrm{QUOTE} \mid \langle S \ \ P^* \rangle \mid \langle \text{`} \ P \rangle \mid P \mid \epsilon \mid \mathrm{JUNK} \\
S & ::= & \mathrm{SYM} \mid \mathrm{VAR} \\
\mathrm{VAR} & ::= & \langle \$ \ \mathrm{SYM} \rangle \mid \langle \$ \ P \rangle \mid \langle \backslash \ \mathrm{VAR} \rangle \\
\mathrm{JUNK} & ::= & \text{`} \mid \mathrm{SJUNK} \\
\mathrm{SJUNK} & ::= & \$ \mid \backslash \mid \langle \backslash \ \mathrm{SJUNK} \rangle
\end{array}
$$

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; 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 largest finite 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. ϵ 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_∞ expressions must be maximal and that the use of the ϵ production for E_∞ must be minimized. The reader is implemented as a deterministic shift-reduce process. See the text for details.

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 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 have a nonterminal E_∞ 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 precedence levels and a set $\mathcal{R}$ of right associative precedence levels. There is a special precedence level 4 for combining expressions with the “null connective”. The null connective is left-associative. The null connective is higher precedence than semicolon and comma but lower precedence than all other connectives. The nonterminal E_∞ includes an epsilon production allowing null arguments.

The nonterminal S ranges over symbols or variables where VAR ranges over variables. Variables are handled 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 grammar is ambiguous. For example the string {`foo (a) (b)`} can be parsed as either the left-branching structure $\{\langle\langle \text{SYM}\ P^* \rangle\ P\rangle\}$ or the right-branching structure $\{\langle \text{SYM}\ P^* \rangle\}$. The reader is of course deterministic — {`foo (a) (b)`} is read as right-branching. While a deterministic grammar for the 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_∞ substrings. The maximal requirement implies that {`foo (a) (b)`} is parsed as the single (right-branching) E_∞ expression $\langle$`foo (a) (b)`$\rangle$. The maximil requirement also implies that the string {`$ foo (a) (b)`} is read as the single E_∞ expression $\langle\langle$`$ foo`$\rangle$ (`a`) (`b`)$\rangle$.

After identifying maximal E_∞ expressions we are left with a sequence of arguments (the E_∞ 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 ϵ production for E_∞ and also add an ϵ 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 4 connective 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.

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 ϵ represents the null connective at precedence level 4. 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. The universal syntax precedence conventions have 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.

3.4 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 ⟨‘ $\{e\}$⟩ is the expression (tree) e with subexpressions (subtrees) of the form ⟨\$ x⟩, where x is a symbol, replaced by the C value of x and subexpressions (subtrees) of the form ⟨\$ $\{w\}$⟩ 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 σ and a pattern $\{e\}$ we define the expression $\sigma(e)$ to be the result of replacing each subexpression (subtree) of e of the form ⟨\$ x⟩ where x is a symbol atom with

$\sigma(x)$. A pattern expression (tree) $\{e\}$ matches an expression (tree) w with substitution σ 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.

4 List of 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 primitives. 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;
typedef struct expstruct{...} * expptr;
```

4.1 Expressions

The macros `backquote` and `ucase` expand into C code using these procedures.

```
expptr string_atom(charptr 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 respectively are expression-valued.

```
expptr int_exp(int i);

int exp_int(expptr s);
```

4.2 Properties

The macro `umacro` expands to code that sets the macro property of some atom to a procedure pointer.

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

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

expptr gensym(charptr s);
```

4.3 Errors and Breakpoints

```
void berror(charptr s);

void breakpt(charptr s);

void NIDE();
```

4.4 Reading and Printing

The procedure `file_expressions` takes a file name and returns a list of the expressions (in universal syntax) contained in the cells of the file. The procedures `pprint` and `mcpprint` and the macro `mcprint` print to the emacs *Messages* buffer in the NIDE.

```
expptr file_expressions(charptr name);

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

void mcpprint(expptr e);

void mcprint(...)
```

4.5 Macro Expansion

```
expptr macroexpand(expptr e);

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

void add_init_form(expptr statement);

void add_preamble(expptr aux_definition);
```

4.6 List Processing

In Lisp lists are terminated with the special value `nil`. In MetaC lists any atom in the cdr of a cell is treated as `nil`.

```
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);
```

4.7 Memory Frames

```
in_memory_frame(<statement>)

voidptr stack_alloc(int nbytes);
```

4.8 Undo Frames

```
void exp_from_undo_frame(<expression>);

voidptr undo_alloc(int nbytes);

void undo_set(voidptr loc, voidptr val);
```

```
void clean_undo_frame(expptr e);
```

4.9 Bootstrapping and File Expansion

```
file_expressions(char * fname);

mcexpand(char * fname1, char * fname2);

init_fun(fname)
```

References

[1] A. Bawden. Quasiquotation in lisp. In *Proceedings of the 1999 ACM SIGPLAN 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.