MetaC 1.0

David McAllester

August 7, 2020

Abstract

MetaC provides a notebook interactive development environment (a NIDE) for C programming. MetaC can also be described as a Lisp-inspired programming environment for C. The NIDE is 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 extends C with Lisp-like features. More specifically, it provides a C-like universal syntax, computed macros, backquote (quasiquote) [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.

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 **4**

1.1 Installation 4

1.2 Basic NIDE Commands 4

1.3 Definitions of Types and Procedures 7

1.4 The Global Variable Array Restriction 8

1.5 The Single Symbol Type Restriction 9

1.6 Backquote and Universal Syntax 9

1.7 Pattern Matching 9

1.8 Gensym and Macro Definitions 10

1.9 Errors and Breakpoints. 11

1.10 Printing. 13

1.11 Load Statements. 13

2 Miscellaneous Features **13**

2.1 Interning and Properties 13

2.2 Memory Frames 14

2.3 Undo Frames 14

2.4 Bootstrapping and File Expansion 16

2.5 Macro Effects 16

2.6 Macro Preliminary Definitions 17

2.7 List Operations 17

2.8 Throw, Catch, and Exceptions 18

2.9 Object-Oriented Programming 19

3 Universal Syntax **21**

3.1 Universal Expressions: cons-car-cdr 22

3.2 Universal Expressions: Phantom Brackets 22
3.3 The Reader Specification 24
3.4 Backquote and Pattern Matching Revisited 25

4 List of MetaC Primitives **26**

4.1 Expressions 27
4.2 Properties 27
4.3 Errors and Breakpoints 28
4.4 Reading and Printing 28
4.5 Macro Expansion 28
4.6 List Processing 29
4.7 Memory Frames 29
4.8 Undo Frames 29
4.9 Bootstrapping and File Expansion 29

1 Introduction and Overview

1.1 Installation

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]

1.2 Basic NIDE Commands

Emacs buffers visiting files ending in the extension .mc are placed in mc-mode which is an extension of c-mode. The mc-mode commands manipulate "sections" and "cells".

Figure 1 shows a simple MetaC file with six cells divided into three sections.

A "section" is a region of conceptually related code. As shown in figure 1 sections begin and end with section delimiter comments o the form

```
/** ============================================================================
 <section name>
============================================================================ **/
```

It is important that the interior of comments of this form are indented so as not to confuse cell boundaries. A cell begins at any nonempty line whose first character is other than white space (space or tab), a close character ')', '}', or ']', a slash '/' or the equals sign '='. The cell ends at the beginning of the next cell or at the end of the file if no next cell.

The commands of mc-mode are listed in fugure 2.

```
/** ========================================================================
 Hello world
======================================================================== **/

‘{hello world}

/** ========================================================================
 Procedure defininition
======================================================================== **/

expptr f(expptr exp){return exp;}

f(‘a)

/** ========================================================================
 Imperative Programming
======================================================================== **/

int x[10];

for(int i = 0; i < 10; i++)x[i] = i;

int_exp(x[5])

{int sum = 0; for(int i = 0; i < 10; i++)sum += x[i]; return int_exp(sum);}
```

Figure 1: A sample MetaC file

```
(define-derived-mode mc-mode
  c-mode "mc-mode"
  "Major mode for meta-c"
  (define-key mc-mode-map "\C-xc" 'make-section)
  (define-key mc-mode-map "\C-\M-u" 'MC:beginning-of-sec)
  (define-key mc-mode-map "\C-\M-d" 'MC:end-of-sec)

  (define-key mc-mode-map "\C-\M-s" 'MC:start-metac)
  (define-key mc-mode-map "\C-\M-x" 'MC:execute-cell)
  (define-key mc-mode-map "\C-\M-r" 'MC:load-region)
  (define-key mc-mode-map "\C-\M-a" 'MC:beginning-of-cell)
  (define-key mc-mode-map "\C-\M-e" 'MC:end-of-cell)
  (define-key mc-mode-map "\C-\M-c" 'MC:clean-cells)
  (define-key mc-mode-map "\C-\M-g" 'MC:indent-cell)

  (define-key mc-mode-map "\C-x‘" 'MC:display-error))
```

Figure 2: The Commands of mc-mode.

```
/** ============================================================================
 Hello world
============================================================================ **/

‘{hello world}
/** 1:hello world **/


/** ============================================================================
 Procedure defininition
============================================================================ **/

expptr f(expptr exp){return exp;}
/** 2:done **/

f(‘a)
/** 3:a **/

/** ============================================================================
 Imperative Programming
============================================================================ **/

int x[10];
/** 4:done **/

for(int i = 0; i < 10; i++)x[i] = i;
/** 5:done **/

int_exp(x[5])
/** 6:5 **/

{int sum = 0; for(int i = 0; i < 10; i++)sum += x[i]; return int_exp(sum);}
/** 7:45 **/
```

Figure 3: The result of executing the cells. Executing a cell inserts or replaces a comment giving the cell execution number (the count of executions since starting or restarting MetaC) and the execution result.

Figure 3 shows the state of the buffer after executing the cells in figure 1.

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. A MetaC cell can be either a statement or an expression. The result of successfully executing a C statement is simply "done". When given a C expression the REPL computes and prints the value which must be an instance of the MetaC data type expptr. A expptr is an abstract syntax tree as defined by a "universal" grammar. The procedure `int_exp` in figures 1 and `??` converts an integer to an expptr. Instances of the type expptr are "universal" in the sense that any parenthesis-balanced string parses as an expptr. Futhermore, parsing a string and then printing the resulting expression is guaranteed to return the same string. Hence instance of this type can also be viewed as representations of strings.

If a C statement in a cell executes a `return` outside of any procedure then that value is returned to the REPL and printed. This is shown in the last cell in figure 3.

1.3 Definitions of Types and Procedures

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

```
typedef struct myexpstruct{
   char * label;
   struct myexpstruct * car;
   struct myexpstruct * cdr;} myexpstruct, *myexp;
/** 7:done **/

myexp mycons(char*s,myexp x,myexp y){
    myexp cell=malloc(sizeof(myexpstruct));
    cell->label=s;
    cell->car=x;
    cell->cdr=y;
    return cell;}
/** 8:done **/

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

[1]If you load mc.el into a running emacs MetaC will not be available in buffers created before the load.

```
      ${myexp_exp(x->car)}
      ${myexp_exp(x->cdr)}};
}
/** 9:done **/

myexp_exp(mycons("foo",mycons("bar",NULL,NULL),NULL))
/** 10:foo bar nil nil nil **/
```

A cell can redefine a procedure 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.

1.4 The Global Variable Array Restriction

To facilitate 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.

```
int y[0] = 2;
/** 10:done **/
```

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

```
int y[1];
/** 10:done **/

y[0] = 2;
/** 11:done **/
```

We can continue with

```
y[0] += 1;
/** 11:done **/

int_exp(y[0])
/** 12:3 **/
```

Here assignments to `y[0]` are allowed but assignments to `y` are not — assignments to array variables are not allowed in *C*.

1.5 The Single Symbol Type Restriction

To simplify the implementation of MetaC all type expressions appearing in procedure signatures and global array declarations must be single symbols. Arbitrary types can be given single symbol names using `typedef`.

1.6 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.

```
expptr friend[0] = `{Bob Givan};
/** 13:done **/

int height[0] = 6;
/** 14:done **/

`{My friend ${friend[0]} is ${int_exp(height[0])} feet tall.}
/** 15:My friend Bob Givan is 6 feet tall. **/
```

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.

1.7 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 $. When a `ucase` is executed the patterns are tried sequentially and stops after the first match. If no pattern matches the `ucase` generates an execution error. Execution errors are handled by the gnu debugger gdb and are discussed below. One can always add a final pattern of {`$any`} to provide a default catch-all case if that is desired. 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. We have the following example.

```
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;
}
/** <n>:done **/

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.
  }
/** <n>: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.

```
int_exp(value(`{5+2*10}))
/** <n>:25 **/
```

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.

1.8 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.

```
umacro{dolist($x,$L){$body}}{
   expptr rest=gensym("rest");
   return
    `{for(expptr $rest = $L; cellp($rest); $rest = cdr($rest);){
      expptr $x=car($rest);
      $body}};
}
/** <n>:done **/

macroexpand(`{dolist(item,list){f(item);}})
/** <n>:
for
  (expptr _genrest33=list;
   cellp(_genrest33);
  _genrest33=cdr(_genrest33);
  ){expptr item=car(_genrest33);f(item);} **/
```

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.

1.9 Errors and Breakpoints.

The execution of a cell involves

1. reading the expression in the cell into an expptr data structure
2. macroexpanding the expression,
3. compiling the resulting C code
4. executing the compiled code

Errors can occur in any of these steps. If a 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 (placed in the buffer *MC compilation*). 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. In this case the "display error" command "\ M x-‘" will bring up a C file that was passed to the compiler. These errors contain the name of the C file passed to the compiler. Examining this C file is sometimes helpful.

There are two kinds of errors that can occur in the final execution phase. The first is a "dynamic check error". This is an error caught by a explicit check in the code such as a `ucase` expression where no pattern matches or a call to the MetaC error function `berror(char *)`. A dynamic error causes control to be passed to the gdb debugger. The continue command then return control to the nide and the value of the cell is printed as "dynamic-check error". It is also possible that execution leads to some other error, typically a segmentation fault. This also causes control to be passed to the gdb debugger but the continue command is not applicable in this case. In this case one can return to the NIDE by entering p NIDE() to gdb.

The MetaC procedure `breakpt(char *)` can be used for breakpoints from which execution can be continued. When `breakpt` is called the NIDE will invoke gdb 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 with the command "\C-\M-s".

1.10 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.

1.11 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.

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

```
int_exp(‘{foo(a)} == ‘{foo(a)})
/** <n>: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 an error to the NIDE.

MetaC clears the base memory frame after every cell execution. It is therefore important not to be using stack allocated memory between cell executions.

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.

`restart_undo_frame(<n>)` is a procedure that restarts the nth undo frame. The base frame has a conceptual index of -1 and cannot be restarted. Undo frame 0 is the first restartable frame. When in frame n (possibly $n = -1$), executing `restart_undo_frame(`$n+1$`)` will push an undo frame and leave the system in frame $n+1$. The pushed undo frame can then act as a "checkpoint". After running many cells and building up additional procedure and variable definitions, re-executing `restart_undo_frame(`$n+1$`)` will remove the procedure and variable definitions, and undo all undo-trailed effects, that were done in undo frame $n+1$ or higher. This provides an alternative to restarting the kernel when, for example, one wants to change the signature of a procedure. Calls to `restart_undo_frame` can be distributed through a file under development to establish various checkpoints from which one can restart.

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. The macro expan-

sion phase of a cell execution adds the initialization forms generated by macro expansion to the beginning of the C code which is then compiled and executed.

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.

2.7 List Operations

The macro `deflists(<type>)` creates operations on lists of the given type. For example

```
deflists(expptr)
```

expands to

```
typedef struct expptr_list_struct{
    expptr first;
    struct expptr_list_struct * rest;}expptr_list_struct, * expptr_list;

expptr_list expptr_cons(expptr x, expptr_list y){
   expptr_list cell = (expptr_list) undo_alloc(sizeof(expptr_list_struct));
   ...}

expptr_list expptr_append(expptr_list x, expptr_list y){...}

expptr exptr_nth(expptr_list x, int n){...}

int expptr_member(expptr x, expptr_list y){...}

umacro{push_expptr($x,$y)}{return `{undo_set($y,expptr_cons($x,$y))}}
```

```
umacro{exptr_iter($x,$y){$body}}{...} //executes body for each x in y

umacro(expptr_listfun($args)}{...} //creates a list of any number of arguments
```

The macro `pushprop(x, getprop(y,prop))` pushes x onto the list stored in the given property of expptr y. This cannot be easily type checked and it is assumed that the given property is a list of pointers.

The list operations are predefined for `expptr` and `voidptr` and automatically defined for each class declared in the object system.

2.8 Throw, Catch, and Exceptions

The following throw and catch macros are contained directly in the MetaC source code.

```
#define CATCH_DIM 1000
int *catch_freeptr;

jmp_buf *catch_stack;
int *error_flg;

#define throw_check() \
  {if(catch_freeptr[0] == 0) \
    {fprintf(stderr,"\n uncaught throw --- C process fatal\n"); cbreak(); exit(1);}}

#define catch_check() \
   {if(catch_freeptr[0] == CATCH_DIM){berror("catch stack exhausted");}}

#define throw_error() \
   {throw_check(); error_flg[0]=1; longjmp(catch_stack[catch_freeptr[0]-1], 1);}

#define catch_error(body) \
 {catch_check(); error_flg[0]=0;\
  if(setjmp(catch_stack[catch_freeptr[0]++]) == 0)\
   {body; catch_freeptr[0]--;\
    } else \
     {catch_freeptr[0]--;\
      if(!error_flg[0])\
        {fprintf(stderr, "uncaught throw caught as error\n"); cbreak();}}}

#define throw() \
   {throw_check(); error_flg[0]=0; longjmp(catch_stack[catch_freeptr[0]-1], 1);}
```

```
#define continue_throw() \
   {throw_check(); longjmp(catch_stack[catch_freeptr[0]-1], 1);}

#define catch(body) {catch_check(); if(setjmp(catch_stack[catch_freeptr[0]++]) == 0)\
  {body; catch_freeptr[0]--;} else{catch_freeptr[0]--; if(error_flg[0])continue_throw();}}

#define unwind_protect(body, cleanup) \
  {catch_check(); if(setjmp(catch_stack[catch_freeptr[0]++]) == 0)\
  {body; catch_freeptr[0]--;} else { catch_freeptr[0]--; cleanup; continue_throw();}}
```

The following extends the above to general exceptions.

```
expptr exception[0] = NULL;

expptr exception_value[0];

umacro{catch_excep{$exception}{$body}{$handler}}{
  return `{
    catch({$body})
      if(exception[0]){
if(exception[0] == `$exception){
  exception[0] = NULL;
  $handler}
else continue_throw();}};
}

umacro{throw_excep{$exception;$value}}{
  return `{
    expcetion[0] = `{$exception};
    exception_value[0] = $value;
    throw();};
}
```

2.9 Object-Oriented Programming

MetaC provides an object system where the collection of classes must be tree structured (no multiple inheritance) with a built-in root class called `object`. Classes are declared with the macro `defclass`. The class declaration

```
defclass{foo{object; expptr x;}}
```

generates the structure definition

```
typedef struct foo_struct{
```

```
  int class_index;
  expptr x;} foo_struct, *foo;
```

Given this declaration, the declaration

```
defclass{bar{foo; foo y; bar z;}}
```

generates the structure definition

```
typedef struct bar_struct{
  int class_index;
  expptr x;
  foo y;
  bar_struct * z;} bar_struct, *bar;
```

The general form is similar to a structure definition in C except that the parent class is named and the class being defined can be used naively as the type of an instance variable. Each class inherits instance variables from its parent and an instance of a class can be safely cast as an instance of any parent class.

Instances of any class `foo` in the class hierarchy can be allocated with the macro call `new(foo)`. Objects are initialized with their class index but no other instance variables are initialized. Instances are allocated from undo memory so that the allocation is undone when the current undo frame is cleared or popped.

Methods are declared with `defmethod`. For example we have

```
defmethod{expptr g(foo self; expptr x;){return x;}}
```

which creates a foo-specific definition of g

```
expptr g_52(foo self_37, expptr x);.
```

Each method is associated with a dispatch table indexed by class. This method definition installs this procedure pointer in the method table for g at the class index for the class `foo` and also at the class indeces for the subclasses of the class `foo`. Once a method is defined at a class it cannot be redefined at any parent or any child of that class.

The above method definition also defines `g` as a macro such that `g(x,y)` expands to inline code which extracts the class index from `x` and extracts the procedure pointer from the method table for g and calls that procedure on x and y with x cast as `foo`. Here the class `foo` can be a proper superclass of the actual class of x. The overhead of a method call as opposed to an ordinary procedure call is a single one-dimensional array reference. There is currently a limit of 100 classes so that the method tables are small.

The system maintains the invariant that methods are inherited from superclasses as new methods and classes are incrementally defined. In particular,

when a new class `foo` is defined the method tables for methods defined on parent classes of `foo` must be updated to include a procedure pointer entry for the subclass `foo`.

Within the body of a method instance variables are defined as free variables. However, the current implementation does not support assignments `x=v` for instance variable `x` within methods — do `self->x = v;` instead and reember to use `setf-undo` if you want effects undone (initialization does not have to be undone as the memory is reclaimed in any case).

List operations are automatically created for each class.

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 ⟨⟨`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 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 4. The printer simply removes the phantom brackets and prints the string that the expression represents.

`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 4: **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. section 3.3.

To reduce the clutter of brackets we will adopt two conventions for supressing brackets — a list convention and a binary operator convention. The list convention abbreviates ⟨`zero` ⟨`one` ⟨`two three`⟩⟩⟩ as ⟨`zero one two three`⟩. The binary operation convention involves connective atoms — atoms whose string consists entirely of characters classified as a connectives as described in section 3.3. For example, the string {`a + b`} reads as {⟨⟨`a +`⟩ `b`⟩}. We then adopt the convention of writing this as {⟨`a + b`⟩}. The MetaC reader bracketing of

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

is

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

which can be abbreviated using the binary operation convention as

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

or abbreviated with the list convention as

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

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}.

  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"

$$
\begin{array}{rcll}
E & ::= & E_\infty \mid E_p & \\
E_\infty & ::= & A \mid \langle A\ E_\infty \rangle & \\
A & ::= & \text{QUOTE} \mid \text{SYM} \mid P \mid \langle\ \text{`}\ A \rangle \mid \langle\ \$\ A \rangle \mid \langle \langle \backslash\ B \rangle\ A \rangle \mid \epsilon & \\
B & ::= & \$ \mid \langle \backslash\ B \rangle \mid \epsilon & \\
P & ::= & (E) \mid \{E\} \mid [E] & \\
E_p & ::= & \langle \langle E_\ell\ \text{CONN}_p \rangle\ E_r \rangle & p \in \mathcal{R},\ \ell > p,\ r \geq p \\
E_p & ::= & \langle \langle E_\ell\ \text{CONN}_p \rangle\ E_r \rangle & p \in \mathcal{L},\ \ell \geq p,\ r > p
\end{array}
$$

Figure 5: **The grammar defining the MetaC reader.** The nonterminal SYM generates symbol atoms; CONN_p generates connective atoms of precedence p; and QUOTE generates string quotation atoms. The reader is defined by a deterministic left-to-right shift-reduce parser in which the ϵ-productions for A and B are selected only when no other option is available. The nonterminal E_∞ generates "arguments" — expressions not containing connectives outside of parentheses. The nonterminal E_p generates expressions whose root connective has priority p (and whose internal connectives have priority at least p). $\mathcal{R}$ is the set of right-associative precedence levels and $\mathcal{L}$ is the set of left-associative levels. Generated cells of the form $\langle w\ \epsilon \rangle$ or $\langle \epsilon\ w \rangle$ are replaced by w with the exception that every binary operation must have a (possibly null) syntax node for each of its two arguments. Having two arguments for all occurrences of binary operations simplifies pattern matching.

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 5.

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 $\langle$` $\{e\}\rangle$ 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.

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

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.

$$`\{`\{`\{\$\{x\} + \langle\backslash\$\rangle\{y\} + \langle\backslash\backslash\$\rangle\{z\}\}\}\} \tag{3}$$

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 $\langle \$\ x \rangle$ 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 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.