Magic Tutorial #S-1: The scheme command-line interpreter

Rajit Manohar

Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Read reference [1]

Commands introduced in this tutorial:

:scm-echo-result, :eval, lots of scheme functions

Macros introduced in this tutorial:

(None)

1 Introduction

Magic's original command-line interpreter has some limitations. Some of these include the absence of definitions with arguments, block structure, the ability to define complex functions. We describe an extension which is almost completely backward compatible with the existing magic command-line syntax, but permits the use of Scheme on the command-line.

2 Backward compatibility

To permit existing magic source files to work within the scheme interpreter, we have had to sacrifice one feature of the magic command-line syntax. Single quotes can only be used to quote a single character. The reason for this limitation is that *unmatched* quotes are used by scheme to stop evaluation of the next input symbol.

Parentheses are used by the scheme interpreter. If you use parentheses outside single or double quotes in your magic source files, you might find that the source files don't work properly. To circumvent this problem, simply put your parentheses in double quotes. You can also use backslashes to quote parentheses as in:

:macro \("echo hello"

Another thing you may notice is that floating-point numbers are parsed as such, and therefore a command such as

:echo 5.3

would display the string **5.300000**. If you really want the string **5.3**, use:

:echo "5.3"

If this difference is undesirable, the scheme interpreter can be turned off at compile-time. Talk to your local magic maintainer if you want this done. We feel that the minor trouble taken in modifying existing magic source files will be outweighed by the advantage of using a more powerful layout language.

3 The scheme interpreter

The interpreter supports a subset of the scheme language. The features of scheme that are missing include character types, vector types, file input/output, complex numbers, the distinction between exact and inexact arithmetic, quasi-quotations, and continuations.

3.1 Command-line interaction

When interacting with the command-line of magic, the interpreter implicitly parenthesizes its input. For example, the command

:paint poly

would be interpreted as the scheme expression

(paint poly)

This has exactly the same effect as the original expression, because all existing magic command-line functions are also scheme functions. Since the valid magic commands vary from window to window, the return value of the function is a boolean that indicates whether the command was valid for the current window.

The boolean **scm-echo-result** controls whether or not the result of the evaluation is displayed. If the variable does not exist, or the variable is not boolean-valued, the result of evaluation is not echoed. Since the input is implicitly parenthesized, typing in

:scm-echo-result

would not display the value of the variable, since it would be evaluated as:

(scm-echo-result)

To display the value of a variable, use the built-in procedure **eval** as follows:

:eval scm-echo-result

This would result in the expression:

(eval scm-echo-result)

which would have the desired effect (note that for this to actually display anything, the value of **scm-echo-result** must be **#t**, and so examining its value is really a futile exercise—which is why it is an example, of course!).

3.2 Types of arguments

Since scheme expressions are typed, we may need to examine the type of a particular expression. The following functions return booleans, and can be used to determine the type of an object.

#t and **#f** are constants representing the booleans true and false. A standard scheme convention is to name functions that return booleans with a name ending with "?". The built-in functions conform to this convention.

The expression *expr* is evaluated, and the type of the result of evaluation is checked.

(boolean? *expr***)**	**#t** if *expr* is a boolean
(symbol? *expr***)**	**#t** if *expr* is a symbol
(list? *expr***)**	**#t** if *expr* is a list
(pair? *expr***)**	**#t** if *expr* is a pair
(number? *expr***)**	**#t** if *expr* is a number
(string? *expr***)**	**#t** if *expr* is a string
(procedure? *expr***)**	**#t** if *expr* is a procedure

For example,

(boolean? #t)=> *#t*
(number? #t)=> *#f*

3.3 Lists and pairs

A pair is a record structure with two fields, called the car and cdr fields (for historical reasons). Pairs are used primarily to represent lists. A list can be defined recursively as either the empty list, or a pair whose cdr field is a list. The following functions are used to extract these fields and to construct new pairs and lists.

(**car** *pair*)	the car field of *pair*
(**cdr** *pair*)	the cdr field *pair*
(**cons** *obj1 obj2*)	a new pair whose car field is *obj1* and cdr field is *obj2*
(**list** *arg1* . . .)	a new list consisting of its arguments
(**null?** *list*)	**#t** if *list* is the empty list
(**length** *list*)	the number of elements in *list*

For example,

(car '(a b c))=> *a*
(cdr '(a b c))=> *(b c)*
(cons 'a '(b c))=> *(a b c)*
(list 'a 'b 'c)=> *(a b c)*
(null? '(a b))=> *#f*
(null? ())=> *#t*

The car field and cdr field of a pair can be set using the following two functions.

(**set-car!** *pair obj*)	sets the car field of *pair* to *obj*. It returns the new pair
(**set-cdr!** *pair obj*)	sets the cdr field of *pair* to *obj*. It returns the new pair

These two functions have *side-effects*, another feature that distinguishes scheme from pure lisp. Another naming convention followed is that functions that have side-effects end in "!".

Try the following sequence in magic:

(define x '(a b))=> *(a b)*
(set-car! x 'c)=> *(c b)*
(set-cdr! x '(q))=> *(c q)*
(set-cdr! x 'q)=> *(c . q)*

After the last statement, the value of x is no longer a list but a pair.

3.4 Arithmetic

The interpreter supports both floating-point and integer arithmetic. The basic arithmetic functions are supported.

(+ *num1 num2*)	the sum *num1+num2*
(- *num1 num2*)	the difference *num1-num2*
(* *num1 num2*)	the product *num1*num2*
(/ *num1 num2*)	the quotient *num1/num2*
(truncate *num*)	the integer part of *num*

The division operator checks for division by zero, and promotes integers to floating-point if deemed necessary. Floating-point numbers can be converted into integers by truncation. The range of a number can be checked using the following predicates:

(zero? *num*)	**#t** if *num* is zero
(positive? *num*)	**#t** if *num* is positive
(negative? *num*)	**#t** if *num* is negative

3.5 Strings

The interpreter supports string manipulation. String manipulation can be useful for interaction with the user as well as constructing names for labels.

(string-append *str1 str2*)	the string formed by concatenating *str1* and *str2*
(string-length *str*)	the length of string *str*
(string-compare *str1 str2*)	a positive, zero, or negative number depending on whether *str1* is lexicographically greater, equal to, or less than *str2*
(string-ref *str int*)	the numerical value of the character stored at position *int* in *str* (The first character is at position 0.)
(string-set! *str int1 int2*)	sets character in string *str* at position *int1* to *int2*
(substring *str int1 int2*)	returns substring of *str* from position *int1* (inclusive) to *int2* (exclusive)

Strings can be used to convert to and from various types.

(number->string *num*)	the string corresponding to the representation of *num*
(string->number *str*)	the number corresponding to *str*
(string->symbol *str*)	a symbol named *str*
(symbol− >string *sym*)	the string corresponding to the name of *sym*

3.6 Bindings and functions

An object (more accurately, the *location* where the object is stored) can be bound to a symbol using the following two functions:

(define *sym expr*)	bind *expr* to *sym*, creating a new symbol if necessary and return *expr*
(set! *sym expr*)	bind *expr* to an existing symbol *sym* and return *expr*

(Note: these functions do not evaluate their first argument.)

The difference between the two is that **define** introduces a new binding, whereas **set!** modifies an existing binding. In both cases, *expr* is evaluated, and the result is bound to the symbol *sym*. The result of the evaluation is also returned.

(define x 4)=> *4*

Functions can be defined using lambda expressions. Typically a function is bound to a variable. If required, a lambda expression or built-in function can be applied to a list.

(lambda *list obj***)**	a new function

(Note: a lambda does not evaluate its arguments.)

list is a list of symbol names, and *obj* is the expression that corresponds to the body of the function. For example,

(lambda (x y z) (+ (+ x y) z))=> *#proc*

is a function that takes three arguments and returns their sum. It can be bound to a symbol using **define**.

(define sum3 (lambda (x y z) (+ (+ x y) z)))=> *#proc*

Now, we can use **sum3** like any other function.

(sum3 5 3 8)=> *16*

A function can be applied to a list using **apply**.

(apply *proc list***)**	apply *proc* to *list*

(Note: both *proc* and *list* are evaluated before application.)

list is used as the list of arguments for the function. For instance, an alternative way to sum the three numbers in the example above is:

(apply sum3 '(3 5 8))=> *16*

An alternative method for creating bindings is provided by the **let** mechanism.

(let *binding-list expr***)**	evaluate *expr* after the bindings have been performed
(let* *binding-list expr***)**	evaluate *expr* after the bindings have been performed
(letrec *binding-list expr***)**	evaluate *expr* after the bindings have been performed

The *binding-list* is a list of bindings. Each binding is a list containing a symbol and an expression. The expression is evaluated and bound to the symbol. In the case of **let**, all the expressions are evaluated before binding them to any symbol; **let***, on the other hand, evaluates an expression and binds it to the symbol before evaluating the next expression. **letrec** permits bindings to refer to each other, permitting mutually recursive function definitions. The evaluation order is defined to be from left to right in all cases. After performing the bindings, *expr* is evaluated with the new bindings in effect and the result is returned.

let bindings can be used in interesting ways. An example of their use is provided later.

Scheme is an eager language, and only a few functions do not evaluate all their arguments (definitions and conditionals). Evaluation can be controlled to some degree using the following two functions:

(quote *obj***)**	the unevaluated object *obj*
(eval *obj***)**	evaluates object *obj*

3.7 Control structures

Since scheme is a functional programming language, functions that are usually written using loops are written using recursion. Conditional constructs are used to terminate the recursion. These constructs are slightly different in that they do not evaluate all their arguments (otherwise recursive functions would not terminate!).

(if *expr arg1 arg2***)**	evaluate *expr* and evaluate one of *arg1* or *arg2*

The **if** construct evaluates its first argument (which must result in a boolean), and if the result is **#t** evaluates *arg1* and returns the result; otherwise *arg2* is evaluated and returned.

For instance, the standard factorial function might be written as:

(define fact (lambda (x) (if (positive? x) (* x (fact (- x 1))) 1)))

A more complicated form of conditional behavior is provided by **cond**.

(cond *arg1 arg2 ...***)**	generalized conditional

Each argument consists of a list which contains two expressions. The first expression is evaluated (and must evaluate to a boolean), and if it is true the second expression is evaluated and returned as the result of the entire expression. If the result was false, the next argument is examined and the above procedure is repeated. If all arguments evaluate to false, the result is undefined.

For instance if **x** was a list, the expression

(cond ((null? x) x) ((list? x) (car x)) (#t (echo "error")))

would return the empty list if **x** was the empty list and the first element from the list otherwise. When **x** is not a list, an error message is displayed. Note that **echo** is a standard magic command.

Often one needs to evaluate a number of expressions in sequence (since the language has side-effects). The **begin** construct can be used for this purpose.

(begin *arg1 arg2 . . .* **)**	sequencing construct

begin evaluates each of its arguments in sequence, and returns the result of evaluating its last argument.

3.8 Interaction with layout

All standard magic commands are also scheme functions. This permits one to write scheme functions that interact with the layout directly. Apart from the standard magic commands, the following scheme functions are provided so as to enable the user to edit layout.

(getbox)	a list containing four members (llx lly urx ury)
(getpaint *str***)**	a list containing the boxes from layer *str* under the current box that have paint in them
(getlabel *str***)**	a list containing the labels under the current box that match *str*
(magic *sym***)**	forces *sym* to be interpreted as a magic command

The pairs (llx,lly) and (urx,ury) correspond to magic coordinates for the lower left and upper right corner of the current box. **getpaint** returns a list of boxes (llx lly urx ury), and **getlabel** returns a list of tagged boxes (label llx lly urx ury) which contain the label string. **magic** can be used to force the scheme interpreter to interpret a symbol as a magic procedure. The evaluation returns the specified magic command.

3.9 Miscellaneous

Some additional functions are provided to enable the user to debug functions.

(showframe)	display the current list of bindings
(display-object *obj***)**	display the type and value of *obj*
(error *str***)**	display error message and abort evaluation
(eqv? *obj1 obj2***)**	checks if two objects are equal
(collect-garbage)	force garbage collection

The following is a complete list of the built-in scheme variables that can be used to control the interpreter.

scm-library-path	a colon-separated path string
scm-echo-result	a boolean used to determine if the result of evaluation should be displayed
scm-trace-magic	controls display of actual magic commands
scm-echo-parser-input	controls display of the string sent to the scheme parser
scm-echo-parser-output	controls display of the result of parsing
scm-stack-display-depth	controls the number of frames displayed in the stack trace output when an error occurs during evaluation
scm-gc-frequency	controls the frequency of garbage collection

3.10 Libraries

The following function loads in a file and evaluates its contents in order.

(load-scm *str***)**	reads scheme commands in from the named file
(save-scm *str obj***)**	appends *obj* to the file *str*, creating a new file if necessary

The file can be in the current directory, or in any of the locations specified by a string containing a colon-separated list of directory names stored in **scm-library-path**.

The format of these files differs from standard magic source files because the contents of a line are not implicitly parenthesized. In addition, semicolons are used as a comment character; everything following a semicolon to the end of the current line is treated as a comment.

For instance,

define f (lambda (x) x)

would define **f** to be the identity function when placed in a magic source file (so as to provide backward compatibility). The same definition would result in an error if placed in a scheme source file.

(define f (lambda (x) x))

The above expression should be used in the scheme file to achieve the same effect.

References

[1] H. Abelson and G.J. Sussman, *Structure and Interpretation of Computer Programs*.

[2] H. Abelson *et al.*, *Revised Report on the Algorithmic Language Scheme*.