```
$Id: asg1-scheme-mbir.mm,v 1.18 2020-12-25 15:41:40-08 - - $
PWD: /afs/cats.ucsc.edu/courses/cse112-wm/Assignments/asg1-scheme-mbir
URL: https://www2.ucsc.edu/courses/cse112-wm/:/Assignments/asg1-scheme-mbir/
```

## 1. Overview

Scheme is a dynamically typed (mostly) functional language with a very simple syntax. In this assignment, you will write a Mini Basic language interpreter in Scheme. The interpreter will read in an intermediate language program, parse it, and then interpret it. No looping constructs may be used, so it is critical that certain parts use proper tail-recursion to avoid function call stack overflow.

## 2. Examples Directory and Readme

**/afs/cats.ucsc.edu/courses/cse112-wm/Languages/scheme/Examples/**

Also look at the references in **README.html** or **README.text** in this directory for some tutorials and references. When using google, be sure to look for references to Racket and Mzscheme. There are other implementations of Scheme which are not necessarily exactly compatible.

## 3. Running mzscheme Interactively

It will be very convenient for you to run **mzscheme** interactively for testing purposes simply by invoking it from the command line, as in:

```
-bash$ mzscheme
Welcome to Racket v7.4.
> (expt 2 128)
340282366920938463463374607431768211456
> (sqrt -1)
0+1i
> (acos -1)
3.141592653589793
> ^D
```

To do this, be sure to put it in your **$PATH**. This can be done by putting the following lines in your **.bash_profile**:

```
export PATH=$PATH:/afs/cats.ucsc.edu/courses/cse112-wm/usr/racket/bin
```

Of course, you may prefer to collapse these multiple shell commands into a single line. If you use a different shell, then setting your **$PATH** will be done differently.

To use the arrow keys on the keyboard to edit previous lines in interactive mode, put the following line in a file **$HOME/.racketrc**:

```
(require readline)
```

Or, after starting **mzscheme**, enter the above command before any other interaction. Or, put the following line in your **.bash_profile**:

```
alias wscheme='rlwrap mzscheme'
```

Then start **mzscheme** with the command **wscheme** instead. This only works at the command line, so don't use it in the **#!** line of the program script. **rlwrap** is a program that wraps another program in the readline utility, allowing use of common terminal editing facilities, such as the use of the arrow keys.

## 4. A Mini Basic Interpreter

### NAME

mbir.scm — a Mini Basic Interpreter

### SYNOPSIS

`mbir.scm` [`-d`] *filename*

### DESCRIPTION

The `mbir` interpreter reads in an `mbir` program from the file whose name is specified in the argument list, stores it in a list, and then interprets that intermediate representation. During interpretation, numbers are read from the standard input and results written to the standard output.

Error messages are printed to the standard error. The first error, whether during compilation or interpretation, causes a message to be printed and the program to exit with an exit code of 1.

### OPTIONS

None.

### OPERANDS

The single filename argument specifies an `mbir` program to be run.

### EXIT STATUS

If the program completes without error, 0 is returned. If not, 1 is returned.

### HISTORY

BASIC (Beginner's All-purpose Symbolic Instruction Code) was designed at Dartmouth College, NH, by John Kemeny and Thomas Kurtz in 1965. A variation of that language was ROM BASIC, distributed by IBM on their original PC in 1980.

Mini Basic is somewhat related. This description of the Mini Basic programming language, assumes that certain things are intuitively obvious. There are only two data types in the language: strings and numbers. Strings are used only in `print` statements. There are no string variables. Variables are floating point numbers.

Edsger W. Dijkstra: *"It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration."* — EWD498.
EWD manuscripts are archived at `http://www.cs.utexas.edu/~EWD/`.

### THE MBIR LANGUAGE

This is a top-down definition of the `mbir` language, specified using a variation of Backus-Naur Form (BNF), the format used to specify Algol-60, yet another one of the ancient languages. In the metanotation, brackets indicate that what they enclose is optional, braces indicate that what they enclose is repeated zero or more times, and a bar indicates alternation. Italics indicate nonterminal symbols and token classes, while quoted courier bold indicates literal tokens.

(a) *Program* → '(' { '(' *Linenr* [ *Label* ] [ *Statement* ] ')' }... ')'

A program consists of zero or more statements, each of which might be identified by a label. Labels are kept in a namespace separate from the *Variable* namespace and do not conflict with each other. The program terminates when control flows off the last statement. A statement with neither a label nor a statement is considered just a comment and not put into the statement list.

## STATEMENTS

Statements are the only organizational structure in the language and are executed one by one in sequence, except when a control transfer occurs. There is no block structure or nesting.

(a) *Statement* → '(' '`dim`' *Arrayref* ')'
*Arrayref* → '(' '`asub`' *Variable Expression* ')'

The `dim` statement creates an array given by the variable name and inserts it into the array table, replacing any previous array already in the array table. The dimension of the array is given by the expression. All values in the array are initialized to `0.0` (as a real). The expression is rounded to the nearest integer before being used as the bound, which must be positive. Since the size of the vector must be an integer, use `(make-vector (exact-round size) 0.0)` to create the array. A subscript $i$ must be in the range $0 \le i < n$, where $n$ is the dimension.

(b) *Statement* → '(' '`let`' *Memory Expression* ')'
*Memory* → *Arrayref* | *Variable*

A `let` statement makes an assignment to a variable. The expression is first evaluated. For a *Variable*, its value is stored into the Symbol table, replacing whatever was there previously. For an *Arrayref*, the store message is sent to the vector representing the array. If the Symbol table entry is not an array, an error occurs.

(c) *Statement* → '(' '`goto`' *Label* ')'

Control transfers to the statement referred to by the *Label*. An error occurs if the *Label* is not defined.

(d) *Statement* → '(' '`if`' '(' *Relop Expression Expression* ')' *Label* ')'
*Relop* → '=' | '<' | '>' | '!=' | '>=' | '<='

The two *Expressions* are compared according to the given *Relop*, and if the comparison is true, control transfers to the statement, as for the `goto` statement.

(e) *Statement* → '(' '`print`' { *Printable* }... ')'
*Printable* → *String* | *Expression*

Each of the operands is printed in sequence. A space is printed before each expression in the list of values, but not before strings. Expresions are evaluated to a number before being printed. A newline is output at the end of the `print` statement. `print` statements are the only place *String*s may occur in `mbir`.

(f) *Statement* → '(' '**input**' *Memory* { *Memory* }... ')'

Numeric values are read in and assigned to the input variables in sequence. Arguments might be elements of an array. For each value read into a *Memory*, the value is inserted into the Symbol table under that variable's key. For arrays, the array must already exist and the subscript not be out of bounds.

If an invalid value (anything that is not a **number?**) is read, the value returned is **nan**. If end of file is encountered, the value returned is **nan** and the variable **eof** is entered into the symbol table with the value 1. The value of **nan** can be computed using the expression **(/ 0.0 0.0)**. Counterintuitively, the expression **(= nan nan)** is false.

## EXPRESSIONS

Expressions consistitute the computational part of the language. All values dealt with at the expression level are real numbers. Invalid computations, such as division by zero and infinite results do not cause computation to stop. The value just propagates according to the rules of real or complex arithmetic or produces not a number (nan). The result of using complex numbers is undefined.

(a) *Expression* → '(' *Binop Expression Expression* ')'
*Expression* → '(' *Unop Expression* ')'
*Expression* → '(' *Function Expression* ')'
*Expression* → *Constant*
*Expression* → *Memory*
*Binop* → '+' | '−' | '*' | '/' | '^'
*Unop* → '+' | '−'

*Constant*s are numbers. Names of *Function*s, *Arrayref*s, and *Variable*s all look like identifiers and their meaning is given by context. **(^ x y)** is exponentiation ($x^y$)

## LEXICAL SYNTAX

*Comment*s being with a semi-colon and end at the end of a line. *String*s are delimited by double-quote marks ("). *Number*s consist of digits, an optional decimal point, and an optional exponent. Keywords and *Variable* names are atoms. All of this is taken care of by Scheme's builtin **read**.

## BUILTIN SYMBOLS

In addition to the operators that are part of the language, the following functions are part of the function table : **abs**, **acos**, **asin**, **atan**, **ceil**, **cos**, **exp**, **floor**, **log**, **log10**, **round**, **sin**, **sqrt**, **tan**, **trunc**.
The following are part of the initial variable table :
**nan = (/ 0.0 0.0) ; eof = 0.0 ; pi = (acos −1.0) ; e = (exp 1.0)**.

## 5. Program Structure

The program will be read in by Scheme's **read** function, and represented internally as a list of statements, each statement having its own structure. After reading in the program, all labels must be put into a hash table, the key being the label itself and the value being the particular statement it refers to.

Interpretation will then proceed down the list from the first statement to the last. The interpreter stops when it runs off the end of the list. A control transfer is executed by fetching the address of a statement from the label table.

All variables are either real numbers or vectors of real numbers. Another hash table is used whose keys are variable names and whose values are real numbers, vectors of real numbers, or single parameter functions. An array subscript operation and a function call are syntactically ambiguous, but are disambiguated at run time by checking the symbol table. An uninitialized variable should be treated as 0.

Your program should not crash, no matter what the input. If a detectable unforseen condition happens due to user error, a message should be printed, giving the name of the file and the statement number.

The usual arithmetic results for infinities are printed by the runtime system, and these should be generated wherever possible. Division by zero, for example, should produce one of these quantities (`+inf.0`, `-inf.0`, `+nan.0`). Add 0.0 to all input numbers to ensure that they are converted to real numbers. Also look at the functions to see which ones need special treatment.

You may ignore the directory `mini-basic.d/mb-programs.d` which contains source code and a translator from Mini Basic to `mbir`. You may also ignore the directory `translator`, which contains the `mb` to `mbir` translator itself, written in Ocaml.

## 6. Functional Style

Programming should be done in entirely functional style, except for maintenance of the symbol tables. That means do not use any imperative functions except as outlined below. In Scheme, imperative functions end with a bang (`!`) to indicate that an assignment is being made. Symbol tables are created with `make-hash` and updated with `hash-set!`. The symbol tables are as follows:

(a) `*stmt-table*` contains all of the individual statement interpreters. It is initialized at the beginning of execution and thereafter never modified.

(b) `*function-table*` is used to hold all of the functions, which include the operators. This is initialized when the program begins using a `for-each` loop containing a `lambda`. (See the example `symbols.scm`).

(c) `*variable-table*` holds the value of all variables, and is updated as needed during interpretation of the program. Whenever a variable in the symbol table is not found, the value 0 is returned. The variable table is initialized with the variables described in the section "builtin symbols".

(d) `*array-table*` is used to hold all arrays defined in the program. Arrays and variables are in separate namespaces. Arrays are created with `make-vector` and updated with `vector-set!`.

(e) `*label-table*` is used to hold addresses of each line, one level up from statements. This is initialized by scanning the list returned by `(read)` when the program begins.

Except for `hash-set!` and `vector-set!` as outlined above, no imperative functions are permitted. Use functional style only.

### 7. Pseudocode Outline

The data structure consists of a recursively nested list:

(a) The top level list consists of a sequence of lines. Each line is pointed at by the **car** of a cell in the top level list.

(b) Each line consists of a line number, an optional label, which is always a **symbol?**, and an optional statement, which is always a **pair?**. Use **null?** to determine whether not something exists. Do not use **list?**.

(c) A statement consists of a keyword followed by operands, mostly expressions.

(d) An expression uses prefix notation in standard Scheme format.

A suggested outline and description of some of the functions follows:

(a) After reading in the program, make one pass over the top level, checking for a label in each line. Each label should be inserted into the label hash with a pointer to the top level node (not the line).

(b) Write a function **interpret-program** takes the top level list as an argument and checks to see if there is a statement.

  (i) If there is no statement, call **interpret-program** recursively with the **cdr** of the top level node as the continuation.

  (ii) If there is a statement, look up the keyword in the statement hash and call **interpret-***statement*, where *statement* is the keyword found in the statement.

  (iii) This function should then call **interpret-program** with its continuation for a statement that is not a control transfer, or for a statement that is a control transfer that is not taken.

  (iv) If this function has a control transfer, or if it is a statement that has a control transfer that is taken, call **interpret-program** recursively with the **cdr**, as explained above.

(c) Write separate functions **interpret-***statement* for each one of the keyword in the language.

(d) The function **evaluate-expression** is called by a statement interpreter.

  (i) It looks up the function in the function table.

  (ii) It uses **map** to call **evaluate-expression** for each of the arguments to the function.

  (iii) Then use **apply** to apply the function to the list of results obtained.

  (iv) Subscripting arrays will require a special case.

### 8. What to Submit

Submit files: **README** and **mbir.scm**. (And **PARTNER** if you are doing pair programming.) **mbir.scm** must be runnable by using it as the command word of any shell command, and hence the execute bit must be turned on (**chmod +x**). It will be run as a shell script, and hence the first line must be the following hashbang:

```
#!/afs/cats.ucsc.edu/courses/cse112-wm/usr/racket/bin/mzscheme -qr
```

Make sure that the Unix command `which mzscheme` responds with the same executable. Important note: This must be the ***first*** line in your script, and your id should be after it. Be sure there are no carriage return characters in the file.

If you are doing pair programming, one partner should submit `mbir.scm`, but both should submit the `README` and `PARTNER` files, as specified in the pair programming guidelines. If you are ***not*** pair programming, read the section "Solo programming".

Be sure to use `checksource` to verify basic formatting. This script, and other scripts, such as `cid` and `elimcr`, are in
     `/afs/cats.ucsc.edu/courses/cse112-wm/bin`
which should also be put in your `$PATH` environment variable.

The `.score/` subdirectory contains instructions to the graders. Be sure your program runs with the test script. If your program runs when typed in manually from the command line, but not using the script, you will receive no points for execution and testing.

## 9. Debugging

To make it easier to debug, run the program interactively and call functions directly from the interactive REPL (read eval print loop). This can be done in several ways:

(a) Use interactive mode in `mzscheme` and call individual functions interactively at the prompt:
```
-bash$ mzscheme
> (load "mbir.scm")
> ...
```

(b) Put the following line in your `.bash_profile`:
```
alias debug-mbir='mzscheme -f mbir.scm -i -- -d'
```
Then, for debugging purposes only, you may use the command
     `debug-mbir` *program*`.mbir`
to run the program. After loading `mbir.scm`, it turns on the `*DEBUG*` flag and then runs the program. After that you see the interactive prompt for the REPL (read eval print loop) which you can then use to call functions directly from the command line.

(c) Another alternative is to put the definition of `*DEBUG*` in the file itself and comment out the call to `main`. Then call `main` or other functions directly from the REPL.

When running the program in producton mode, call it directly from the command line using the command `mbir.scm`, without using the name `mzscheme` at the command line.