$Id: asg1-scheme-sbi.mm,v 1.24 2017-03-29 16:21:41-07 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps112-wm/Assignments/asg1-scheme-sbi
URL: http://www2.ucsc.edu/courses/cmps112-wm/:/Assignments/asg1-scheme-sbi/

## 1. Overview

Scheme is a dynamically typed (mostly) functional language with a very simple syntax. In this assignment, you will write a Silly 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 nuking the function call stack.

## 2. A Silly Basic Interpreter

**NAME**

sbi.scm — a Silly Basic Interpreter

**SYNOPSIS**

**`sbi.scm`** *filename*

**DESCRIPTION**

The SB interpreter reads in an SBIR 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 SBIR 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.

(People used to spell the names of programming languages in all upper case because keypunches, such as the IBM 026 and 029, did not have lower case. Also, most printers usually had only upper case letters mounted, such as the IBM LN print train. A request to get upper and lower case, as with the IBM TN print train, would cause the job to go into an overnight queue.)

This version of basic is somewhat related, but no attempt is made to make it exactly the same. This description of the Silly Basic programming language, assumes that certain things are intuitively obvious. There are only two data types in the language: strings and real numbers. Strings are used only in **`print`** statements. There are no string variables. All variables are real

numbers.

**EWD498**

And don't forget about what Dijkstra said about this language :

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.

The EWD manuscript archive is at `http://www.cs.utexas.edu/~EWD/`.

**THE SBIR LANGUAGE**

This is a top-down definition of the SBIR 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 stick indicates alternation. Italics indicate nonterminal symbols and token classes, while quoted courier bold indicates literal tokens.

1. *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.

1. *Statement* → '`(`' '`dim`' *Array* '`)`'
   *Array* → '`(`' *Variable Expression* '`)`'

   The `dim` statement creates an array given by the variable name and inserts it into the Symbol table, replacing any previous variable, array, or function already in the Symbol table. The dimension of the array is given by the expression.

   Unlike C, the lower bound of the array is 1 and the upper bound is the dimension, which may be an arbitrary expression. The expression is rounded to the nearest integer before being used as the bound, which must be positive.

2. *Statement* → '`(`' '`let`' *Memory Expression* '`)`'
   *Memory* → *Array* | *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 *Array*, the store message is sent to the vector representing the array. If the Symbol table entry is not an array, an error occurs.

3. *Statement* → '`(`' '`goto`' *Label* '`)`'

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

4. *Statement* → '`(`' '`if`' '`(`' *Relop Expression Expression* '`)`' *Label* '`)`'
   *Relop* → '`=`' | '`<`' | '`>`' | '`<>`' | '`>=`' | '`<=`'

   The two *Expression*s are compared according to the given *Relop*, and if the comparison is true, control transfers to the statement, as for the `goto` statement. Note: `<>` is the symbol for not equal. The others should be obvious.

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

   Each of the operands is printed in sequence, with a space before *Expression* values. A newline is output at the end of the `print` statement. `print` statements are the only place *String*s may occur in SBIR.

6. *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 *Variable,* 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.

   The variable `inputcount` is inserted into the symbol table at end of execution of this statement and initialized to the number of values successfully read in. A value of −1 is returned to indicate end of file. If anything other than a number occurs, that token is discarded, an error message is printed, and scanning continues.

**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 arithmetic.

1. *Expression* → '`(`' *Binop Expression Expression* '`)`'
   *Expression* → '`(`' *Unop Expression* '`)`'
   *Expression* → '`(`' *Function Expression* '`)`'
   *Expression* → *Constant*
   *Expression* → *Memory*
   *Binop* → *Unop* | '`*`' | '`/`' | '`%`' | '`^`'
   *Unop* → '`+`' | '`-`'

   *Constant*s are numbers. Note that names of *Function*s, *Array*s, and *Variable*s all look like identifiers and their meaning is given by context. In particular, the syntax of a function call and an array subscript is ambiguous. The code for both is just to send a message to the Symbol table and get back a result.

   The expression `(% x y)` is equivalent to `(- x (* (trunc (/ x y)) y))`.

The expression `(^ a b)` is exponentiation, mathematically $a^b$.

**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 supported: **`abs`**, **`acos`**, **`asin`**, **`atan`**, **`ceil`**, **`cos`**, **`exp`**, **`floor`**, **`log`**, **`log10`**, **`log2`**, **`round`**, **`sin`**, **`sqrt`**, **`tan`**, **`trunc`**. There is no facility for the user to add functions to the symbol table, although they can be replaced. The variables **`pi`** and **`e`** are also initially part of the symbol table, and they, too, can be replaced.

Thus, if you like, you can follow the law in Indiana, according to *House Bill No. 246, Indiana State Legislature, 1897*, which purportedly attempted to set the value of $\pi$ to 3 [**`http://en.wikipedia.org/wiki/Indiana_Pi_Bill`**].

## 3. 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. A second 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`**). Make sure to add 0.0 to the denominator to ensure that you have a real number. Also look at the functions to see which ones need special treatment. While there is no way to input a complex number, note that some computations, such as **`sqrt(-1)`**, may produce them, and thus will be written out in MzScheme's complex number notation.

The directory **`test-sbir`** contains sample test data. You may ignore the directory **`src-sb`**, which contains source code and a translator from Basic to SBIR. You may also ignore the directory **`sbtran`**, which contains the SB to SBIR translator itself, written in Ocaml.

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

(b) **`*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.

(c) **`*variable-table*`** which holds the value of all variables. This is initialized witgh **`pi`** and **`e`** and is updated as needed during interpretation of the program. Arrays are created with **`make-vector`** and updated with **`vector-set!`**.

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

## 5. Examples directory

**`/afs/cats.ucsc.edu/courses/cmps112-wm/Languages/scheme/Examples/`**
**`http://www2.ucsc.edu/courses/cmps112-wm/:/Languages/scheme/Examples/`**

## 6. 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-1$ mzscheme
Welcome to Racket v6.1.
> (expt 2 64)
18446744073709551616
> ^D
```

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

```
export courses=/afs/cats.ucsc.edu/courses
export cmps112=$courses/cmps112-wm
export PATH=$PATH:$cmps112/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.

## 7. What to Submit

Submit two files: **`README`** and **`sbi.scm`**. It 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/cmps112-wm/usr/racket/bin/mzscheme -qr
```

Also, 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.

If you are doing pair programming, one partner should submit `sbi.scm`, but both should submit the `README` and `PARTNER` files, as specified in the pair programming guidlines.

Be sure to use `checksource` to verify basic formatting.