COSC 341 WINTER 2009 Programming Project #1

Distributed: January 27, 2009 **Due**: February 19, 2009, 7:30pm

Synopsis: Write an interpreter for a programming language (called "SL") that has the syntactic and semantic specification given below. Provide input to the interpreter (i.e., source programs in SL) in the shell or under control of your IDE.

The executing interpreter provides a prompt (">") to the user. The user enters a single statement in SL. The interpreter calculates the result, stores any side-effects, and returns the result.

The program can be written in Java, C/C++. Other languages, check with me. Don't bother checking for perl. Perl is too perfect a fit for this project (and is the language you probably *should* use for this task). However, perl will do too many things for you that I want you able to do yourself. No perl. Also no Pascal or Delphi.

Language Syntax:

Meta symbols: | , ::= * +

Terminals:

(, **)** , **if**, **while set**, **begin**, + , * , = , < , **print**

Non terminals:

input, arglist, variable, expression, value, variable, optr, op, integer, name

Start symbol:

input

Productions:

input	::=	expression
arglist	::=	**(** variable* **)**
expression	::=	value
	|	variable
	|	**(if** expression expression expression **)**
	|	**(while** expression expression **)**
	|	**(set** variable expression **)**
	|	**(begin** expression+ **)**
	|	**(** optr expression* **)**
	|	**(bye)**
optr	::=	op
value	::=	integer
op	::=	+ | - | * | = | < | **print**
variable	::=	name
integer	::=	sequence of digits, possibly preceded by minus sign
name	::=	sequence of characters not an integer and not including a blank, (,) .

Language Semantics

The only values are integers. *false* is represented by 0, *true* is represented by any other integer.

`(if e1 e2 e3)`: If e1 evaluates to 0 then return the value of e2, else return the value of e3.

`(while e1 e2)`: If e1 evaluates to 0, return 0. Otherwise, evaluate e2 then re-evaluate e1. Continue until e1 evaluates to 0.

`(set x e)`: Evaluate e, assign that value to x, return that value.

`(begin e1, e2, … en)`: Evaluate each of e1, e2, … en in order. Return the value of en.

`( bye )` : A somewhat klugey way to exit the program. First print out a reasonably formatted symbol table, including names and values, then do a clean exit.

Symbol table

Variables are not declared ahead of time. When a variable is used (e.g., as one of an arglist, as an expression, or in a `set` statement) you do the following: Check the symbol table to see if the variable is already present. If not present, then create a new entry for the variable.

Java has some nice data structures for this. The rock-bottom simplest (but not so nice) is an array of objects, where each object contains name and value. The `set` statement will search for the object by its name, then modify the object's value field.

Structure of the interpreter

The simplest structure is a `read-eval-print` loop. The main body of the program contains these three calls. `Read` an expression, `evaluate` that expression, then `print` the value.

`eval` is the most interesting part and can incorporate a recursive descent parser if you wish. Eval will certainly be a recursive function. For example, consider

```
( print ( begin ( set a ( + 3 2 ) ) ) )
```

Assuming you don't call `eval` on integers, then `eval` will be called four times (`print, begin, set, +`). Here is the expression tree that visualizes this.

Errors
When the interpreter is not able to complete interpretation of an expression, it must output some information to the user, do any internal clean up necessary (e.g., don't continue on the current input), then prepare for the next input from the user (e.g., provide the prompt).

To simplify this, any side effects that were a result of partial execution of a statement will not be unrolled -- they will remain in the symbol table.

The minimum output you need to provide is the following:
"Error. Current expression:" < text of the current expression goes here >

You may provide more useful output if you desire.

Testing
You will need to test exhaustively. You are responsible for making sure your program works for legal input. Here is a short list of examples:

```
>    3

>    ( print 3 )

>    x

>    ( set x 2 )

>    x

>    ( print x )

>    ( + x 3 )

>    ( print ( + x 3 ) )

>    ( set y ( - x 3 ) )

>    y

>    ( if  0 ( set a -3 ) ( set a 3 ) )

>    ( if x ( set a -4 ) ( set a 4 ) )

>    ( if ( < x 3 ) ( set a -5 ) ( set a 5) )

>    ( begin ( print x ) ( set x ( - x 1 ) )
```

```
>     ( while x ( begin (print x ) ( set x ( - x 1 ) ) ) )

>     ( bye )
```

Write programs

(1) Write a program that computes `b^2 - 4 * a * c`. To supply different actual values of `a`, `b`, and `c`, use copy and paste, then manually replace the values in the three set statements:

```
(begin
      (set a 4)
      (set b 2)
      (set c -1)
      . . .
)
```

(2) Write a program that computes the mean and median of three numbers x, y, z. Again, make the code generic so that you can hardcode different values for x, y, z easily.

Turn in:

- You'll need to demo this program. We'll organize demos later.
- Hard list of program
- Hard copy screen shot of testing statements given above. (You'll be demoing other tests)
- Hard copy screen shot of b*b - 4 * a * c computed for two different value sets
- Hard copy screen shot of mean and median computed for two different value sets

- Email versions accepted ONLY on the due date itself.

Grading:

Meets specifications	80%
Meets style standards	10%
Code elegance	10%

Grading is based on 100%. No late projects accepted!

Bonus points given for code turned in early. 2% per day, up to 20% maximum. Get your project time stamped by the front office or by me.

Start now.

Development advice: *Start small. Debug one thing at a time. KISS. Don't get bogged down. If you get stuck somewhere, try to make progress elsewhere.*

Add to your debugging test cases as you develop new functionality. Every time you add new functionality, run through the entire set of test cases.

Get the read-eval-print *skeleton* in place first thing.
Then, go in the following order:

- Tokenize.
- expression ::= value
- Control the symbol table next and do expression ::= variable
- print
- set
- begin
- . . .