CS234: DATA TYPES AND STRUCTURES (F'01)

Assignment 2: Interpreter for a Simple Language.

Assigned: 10/09

Due: 10/25, electronically by 11:30a.m.

Overview: The assignment develops an *Interpreter for a simple imperative programming language* that can manipulate very large integer values using the **APNumber** ADT from Assignment 1. The solution should be developed in two parts: an ADT for representing a computer store (an AVL or a Splay Tree that stores the variable identifier—**APNumber** value pairs) and a simple parser—interpreter (based on building and traversing an expression tree).

1 Tree based Store

Design and implement an ADT "**Store**" based on an AVL or a Splay search tree (you can choose the implementation here) that allows storing and retrieving arbitrary-precision integers given a variable identifier (a string).

The ADT should support (at least) the following interface:

- **Store()**
  constructor; takes no input, creates a new empty instance of **Store**.
- **public void store(String id, APNumber n)**
  takes as its input a string and a an **APnumber** and stores the **APnumber** in the **Store** under the name given by the string. If a variable with this name already exists, the old value is replaced by the new value, otherwise a new entry in the **Store** is created.
- **public boolean delete(String id)**
  takes as its input a string and deletes the variable identified by the string from the **Store**. The function returns `true` if the specified variable was found and deleted from the **Store** and `false` otherwise.
- **public APNumber get(String id)**
  takes as its input a string and looks up the value of the variable identified by the string in the **Store**. If the variable is not found a `null` reference must be returned.
- **public void print()**
  print out the contents of the Store instance alphabetically by the variable identifiers as required for the output of the 2nd part of A2.

Notes:

- You MUST implement the data structures used by the **Store** on your own (in particular, you must not use Vectors, Trees, or similar classes in JAVA).
- The number of variables in an **Store** should only be limited by the availability of memory.

- In a PASCAL implementation, the **Store** argument to the **insert**, **delete**, and **get** methods must be made explicitly by passing a reference to a **Store**.
- The signatures for both JAVA and PASCAL are available on the assignment page.

2 The Interpreter

To finish the assignment use the **APnumber** and **Store** ADTs to develop an *Interpreter for a simple language*.

Here is the syntax of the language: (the bold-face type denotes the keywords in the language):

$$
\begin{array}{rcl}
\mathit{cmd} & ::= & \textbf{skip} \\
 & | & \textbf{let}\ \mathit{var} \mathbin{\textbf{:=}} \mathit{exp} \\
 & | & \textbf{(}\ \mathit{cmd}\ \textbf{;}\ \mathit{cmd}\ \textbf{)} \\
 & | & \textbf{if}\ \mathit{exp}\ \textbf{then}\ \mathit{cmd}\ \textbf{else}\ \mathit{cmd}\ \textbf{fi} \\
 & | & \textbf{while}\ \mathit{exp}\ \textbf{do}\ \mathit{cmd}\ \textbf{od}
\end{array}
$$

where *var* is a variable identifier (a string of *lowercase* letters a,...,z) and *exp* is an arithmetical expression in postfix notation with the same syntax used in Assignment 1 with one additional "operation": if a lowercase string appears in the expression, it signifies looking up the value associated with the variable identified by this string in the **Store** (if the variable does not exist an error must be reported).

The meaning of programs in the above language is defined as follows:

- **skip** (a no operation): do nothing.
- **let** *var* **:=** *exp* (an assignment): in a **Store** the variable identified by a string *var* is associated with the value obtained by evaluating the postfix arithmetic expression *exp*. In addition, *exp* must evaluate to a *single* value (unlike in Assignment 1); if more than one value is produced, an error must be reported.
- **(** c_1 **;** c_2 **)** (a sequential composition): execute c_1 and then execute c_2.
- **if** *exp* **then** c_1 **else** c_2 **fi** (an if-conditional): evaluate a postfix expression *exp* and then, if the result is non-zero, execute c_1; otherwise execute c_2. Again, *exp* must evaluate to a single value or an error must be reported.
- **while** *exp* **do** *cmd* **od** (a while-loop): evaluate a postfix expression *exp* and, if the result is non-zero execute *cmd* and repeat; otherwise terminate the loop. Similarly to the above two cases, *exp* must evaluate to a single value or an error must be reported.

You may assume that the keywords **skip**, **let**, **:=**, **(**, **;**, **)**, **if**, **then**, **else**, **fi**, **while**, **do** , and **od** may not (and will not) appear elsewhere in the program (e.g., as a part of a variable name).

The interpreter reads the program from *standard input*, builds an *expression tree* for commands (*cmd*), evaluates the program starting with a new empty **Store**, and (if the execution terminates) prints out the final values of all variables used in the program in alphabetical order. The **APNumber**s should be printed 50 digits a line (as in the example below).

The output *must* look as in the following example (further samples may be posted on the class web page and the newsgroup).

Example: A sample session with the interpreter:

```
% Interp
 ( let i:=100 ;
   ( let j := 1 ;
     while i do ( let j := j i * ;
                  let i := i 1 -
                )
             od
   )
 )
<end-of-file>
i = 0
j = 93326215443944152681699238856266700490715968264381
    62146859296389521759999322991560894146397615651828
    62536979208272237582511852109168640000000000000000
    00000000
```

Assignment submission. On the assignment due date you are to submit a completed implementation of the interpreter. In particular, this requires that you write a function in PASCAL/JAVA for each of the operations of the **Store** ADT.

Your submission should include the following three files:

1. `Store.pas` (`Store.java`) that contains a *Pascal unit* (*Java Class*) implementation of the **Store** ATD.
2. `Interp.pas` (`Interp.java`) containing the implementation of the interpreter, and
3. `Output.txt` that contains test cases that exercise your implementation (the test cases should demonstrate that your implementation is correct and part of the grade will depend on the quality of your test cases).
4. If you are using JAVA, you may submit additional `xxx.java` files that implement additional classes needed in your solution. However, you must make sure that the complete assignment (i.e., the arithmetic ADT, the interpreter, and any additional files can be compiled using a single `javac *.java` command.
5. you are **responsible** for reading any clarifications for the assignment posted to the newsgroup, in particular for additional submission instructions posted there.

Final **important** note: you must include a comment at the start of each function of the **Store** ADT indicating the complexity of the worst case running time of the function in the terms of the size of its inputs. In particular, for your implementations of the operations for the **Store** ADT you must indicate the complexity of the operation in the size of the operands. Your comment must clearly indicate how you have measured the problem size.