Course Project Deliverable #4: RuM Interpreter (Applying Semantics)
COSC 252, Fall 2016
Jeremy Bolton

Over the course of the term you will design and build a programming language. Specifically you will build an interpreter for the programming language specified below. Your fourth deliverable is the fully functional interpreter of this language/grammar.

Instructions:

Apply semantics to RuM – the result is a fully functional interpreter.

Details:

Semantics:

I. Data types
   a. ints
   b. floats
   c. strings
   d. lists
   e. booleans
   f. objects (user defined)

II. Expressions
   a. Operators: +, -, - , * , /, ^, ()
   b. Conform to standard operator precedence and associativity
   c. Implicit (widening) coercion for numerics
   d. Include + for string concatenation. If one operand is not a string, coerce its value to a string value. The resulting value is bound to type string.

III. Boolean Expressions (extra credit +1)
   a. Operators: || , &&, !
   b. Short circuit

IV. Assignments
   a. Assume that a variables scope is bound when its first assignment is made.
   b. Type of a variable is bound during assignment to type of RHS expression

V. Scope
   a. RuM will employ static scope
   b. All assignments and definitions directly input into interactive mode will be bound to global scope. Also you can assume all class and function definitions are bound to global scope.

VI. Control Structures

- If statements:
  - What types of expressions do you allow in if- statements? What should evaluate to true / false?
  - Should if statements blocks have a separate scope/environment?
- Loop statements
  - What types of expressions do you allow in if- statements? What should evaluate to true / false?

    - Should if statements blocks have a separate scope/environment?
- Function invocations
    - Each time a function is invoked, a new (blank) scope is created.
        - Arguments are bound to parameter names in new scopes symbol table (as appropriate, consider pass by reference scenario which may be different)
        - Once function execution completes, the value and type of the function definitions named variable to be returned, is returned as the evaluation of the function invocation.
- Classes and Objects
    - Include inheritance when appropriate keyword is used. If no inheritance is specified in a class def, then that class will inherit (directly) from the Object class.
    - Objects are created using keyword new(<string>) ,where <string> is the name of the class.
- Pre-population of global symbol table (some built-ins)
    - You may wish to implement separate symbol tables for classes, functions, and variables (or not)
    - Prepopulate class Object, with
        - member method toString(), which seemingly returns a <string> containing the variable name of the calling object.
        - Member method get(<string>), which returns the value of member variable with name <string>.
        - Member method set(<string>, val), which sets the calling objects member named <string> with value val.
    - Prepopulate class List, with overrides
        - member method toString(), which seemingly returns a <string> containing the contents of the calling list.
        - Member method get(<int>), which returns the value of the item at index <int>.
        - Member method set(<int>, val), which sets the list value to val at index <int>.
            - if <int> is negative, throw runtime error
            - if <int> is larger than the current size of the list, then increase the size of the list and “zero-fill” the internal vacant spots.
    - Prepopulate the function table with
        - print(<expr>), which will print to screen the evaluation of < expr >
        - copy(<expr>), which will create a *deep* copy of the value of <expr>
- Runtime Stack
    - Must maintain appropriate environments with scope information.
    - Must maintain activation records to track callers return location.
    - Scoping, environment, and activation records should be able to facilitate recursion.
- Runtime Errors
    - When runtime errors are detected, the interpreter should print an intuitive message describing the error, stop execution and return control the appropriate scope, continue in interactive mode.
    - Since RuM is implicitly typed, all built-in operators and functions should perform runtime type checking and throw runtime errors when appropriate.
        - Examples:
            - “runtime error: attempted access of variable not defined: xSum”

        - "runtime error: operator not defined on operand of type Object: obj + 5"
- Memory Management and copies.
    - Note there are no explicit pointers in RuM; however the understanding of this concept is important for RuM programmers.
    - All assignments will be (by default), a shallow copy (e.g. copy by reference.)
        - Programmers can use the copy function for deep copies
    - Arguments will be (by default), a deep copy (except when the & token is used)
    - Return values will be (by default), a deep copy
    - Since no explicit use of pointers is available to the programmer, memory management is largely managed by the interpreter. All variables bound to a scope that has been destroyed should be properly deallocated by the interpreter (no memory leaks). NOTE: *Be careful not to deallocate a variable passed in by reference if the bound scope is still alive, that is, if a variable is still accessible – do not deallocated it! Hint: Employ a scheme to determine whether a variable is "garbage" or not.*

Please feel free to use the Project Discussion board in BB as a means to communicate/note considerations and bugs.

Tips:

- You will need to insert semantics directly into your top-down parsing functions.
- As a result your parsing functions will likely have return values, structures, or objects
- You will need to implement a structure(s) or object(s) to represent symbol tables, function tables, and class tables: the runtime environment. You will also need to implement structures to maintain the runtime stack, activation record (dynamic link), and lexical scope (static link).

Example.
Consider the following simple grammar. Assume <num> and <var> are appropriately characterized by tokenizer.

```
<program>  -> <stmt_L>
<stmt_L>   -> <stmt> {<stmt>}
<stmt>     -> <var> = <expr>;  | <expr>
<expr>     -> <term> {(+|-) <term>}
<term>     -> <num>          | <var>
```

The function designed to parse a stmt might look similar to the following pseudocode:

```
Val  parseStmt(){
   // must determine which RHS to choose – use lookahead
   // In this grammar, we need two lookaheads here!
  // append(toPrint, "Enter Statement\n")
   if next next token is ASSIGN_OP // assignment statement
         Val var = parseVar();
         Val op = parseAssign();
         Val expr = parseExpr();
         Val assign = addToEnv(runTimeStack, var, expr); // global runTimeStack, helps maintain scope
         return assign;
   else // expression
        Val var = parseExpr();
         return var;

   //append(toPrint, "Exit Statement\n")
   //return toPrint;
}
```

Grading Notes, approx. scoring guide:

- The program must compile, otherwise a grade of zero will be assigned.
- Evaluates simple expressions: 0 – 40%
- Assigns and retrieves variables: ~ 50%
- Correctly evaluates control statements: ~ 60%
- Correctly defines evaluates functions (with static scope) using runtime stack: ~ 80%
- Correctly implements classes with inheritance and dynamic dispatch: ~ 90%
- Identifies and appropriately handles runtime errors: ~ 95%
- Provides appropriate garbage collection and deallocation: ~100%

Milestones

Design and basic semantics. Please spend an appropriate amount of time designing your solution. The scope of this project is deep and a good design will save you time in the long term. I encourage you to create UML class diagrams or ERDs to design the basic structures needed to implement Values and symbol tables (and later on plan to design structures for the runtime stack and objects (inheritance, dynamic dispatch, …)). Incorporate runtime errors and messages into each stage**.**

The lectures will co-align with these milestones; I strongly encourage you to follow this schedule.

- *Milestone 1: 11/9:* Basic expressions, types, Values, and symbol table for variables (approximate time. 12 hours)
- *Milestone 2: 11/18:* control structures, function defs, function invocations, and runtime stack (including scope, activation records, …) (approximate time. 14 hours)
- *Milestone 3: 11/24:* classes (add lists here), inheritance, dynamic dispatch (approximate time. 10 hours)
- *Milestone 4: 11/26:* Final Testing

Sample Test Files will be posted in BB.

Output: *Please comment-out all output related to the parse tree.* The interpreter should evaluate any input sentence and, by default, should print out the "last" value computed. If the input is multiple statements, only print out the last evaluated statement. For function definitions and class definitions, you can simply print out the name of the function or class. For Example:

Input: 3+4
Ouput: *7*

Input: x = 2+5/1;
Output: *x is 7*

Input: x = 1; y= 10;
z = y+10;
Output: *z is 20*

Input: if(x == 9) // lets say x is 9
z = z + 1
x = x -1;
endif
Output: *x is 8*