[P07] (Doom and) Gloom

Due: 12:01am, Monday November 4th.

Description

At its core, the Gloom programming language is an integer calculator. Expressions in Gloom consist of integer literals and mathematical operators and are specified using a parenthesis-free postfix notation where operators follow their operands. The expression `1 2 + 3 *` is read left to right and corresponds to the infix expression `(1 + 2) * 3`.

Postfix notation expressions can easily be read and evaluated using a stack. Reading left to right, each literal is pushed on to the stack. When an operator is read, the operator's arguments are popped off the stack, the operation is performed, and the result is pushed back on to the stack. Literals and operators are always space delimited and appear as a sequence of words.

Operators

There are only five arithmetic operators ($+$, $-$, $*$, $/$, mod) and one comparison operator ($>$) in Gloom. In time we will synthesize the remaining basic operations using these building blocks.

Booleans are not a primitive type in Gloom. Instead Gloom assumes that 0 is the false value and -1 is the true value.

The stack effect describes the parameters and results of each operation. Stack effect notation is a short comment of the form `( before -- after )` describing the change to the top of the stack. The $-$ operator has a stack effect `( a b -- c )` indicating that the top two stack entries are exhausted and a new value $c = a - b$ is pushed on to the stack. The `mod` operator is a true modulus, unlike Java's remainder (`%`) operator.

In order to construct complex, stand-alone expressions we must be able to shuffle values on the stack. This language has four basic stack shuffling operators: dup, drop, over, and swap. Each of these are apply named: `dup` duplicates and `drop` deletes the top value of the stack, `over` duplicates the second value (immediately under the top value), and `swap` swaps the two top values. The stack effects for each of these methods are given later.

Lists

Another primitive type in Gloom is the list. Lists are specified using square brackets with zero or more literals and operators given in between. A simple list containing the first few square values would be `[ 1 4 9 16 ]`. The list is pushed on to the stack as a single entry and may be interacted with using a set of operators which we will be implementing in the next project.

A more complex list might contain an operator, such as `[ 1 + ]` or `[ dup * ]` which represent incrementing and squaring an integer respectively.

Any list may be executed using the `eval` operator. The simple squared values list when evaluated would be interpreted as any other Gloom expression; 1, 4, 9, and 16 would be pushed on to the stack. An expression such as `41 [ 1 + ] eval` would push 41 and the list `[ 1 + ]` on to the stack and then `eval` would execute the list, incrementing 41. In the next project we will use lists to define Gloom operators.

Comments are a special form of list which are specified using parentheses with zero or more literals and operators given in between. The interpreter will discard comments. A careful reader will note that stack effect notation is a type of comment.

Both lists and comments may be nested arbitrarily. A hint for parsing lists and comments is given later.

Type Predicates

Type predicate operators function on the type of the top element on the stack. The `int?` and `list?` predicates pop an element off the stack and returns true (-1) if that element is of type int or list respectively. Otherwise the predicate operators return false.

Retain Stack

Using lists and a single stack is sufficient to make Gloom as powerful as any other programming language. However, building the higher-level semantics is fairly clumsy and difficult.

In order to simplify the process, Gloom has a second stack called the retain stack. Only the `>r` and `r>` operators utilize the retain stack. The first, `>r` (to retain), pops the top value on the main stack and pushes the value on to the retain stack. The second, `r>` (retain to), reverses that operation.

Do not worry if you don't immediately understand why the retain stack is helpful. The test cases given in this project and the next should help clarify its necessity.

Concatenative Languages

Gloom is an interpreted, stack-based, concatenative programming language.

In *applicative languages* such as C or Java, expressions consist of function application to arguments. A *concatenative language* consists of a sequence of words (literals and functions) which operate on a central data structure. For example, the applicative program

```
y = foo(x)
z = bar(y)
w = baz(z)
```

would be written as the sequence of functions

```
foo bar baz
```

Juxtaposing expressions in a concatenative language indicates the composition of functions.

As Gloom is stack-based, the central data structure is a pair of stacks. However, there are other design choices available. Om uses prefix notation and passes the remainder of the program as the data from function to function and Deque uses a double-ended queue as its primary data structure.

Requirements

This project is implemented in two parts. The first part involve writing data structures, and the second part involves writing an interpreter for the Gloom programming language.

You are not allowed to use any imports with the exception of the `Scanner`, `NoSuchElementException` classes. You may use the `Map` and `HashMap` classes in the interpreter as well.

Data Structures

A `Stack` and `List` class are given with stubbed out methods. You must implement the stubbed out methods to provide the appropriate functionality. You may implement any type of stack/list you prefer as performance is not an important factor. You may also add behavior to these classes, however you may not alter the given method headers.

Gloom

An `Interpreter` class is given with stubbed out methods. You are responsible for writing an interpreter which reads a set of tokens from `Scanner` and evaluates the expression.

Your interpreter must be able to interpret integer and list literals as well as the following operators, listed with their stack effects.

- `dup   ( x -- x x )`
- `over  ( x y -- x y x )`
- `drop  ( x -- )`
- `swap  ( x y -- y x )`
- `>r    ( x -- )`
- `r>    ( -- x )`
- `+     ( x y -- z )`
- `-     ( x y -- z )`
- `*     ( x y -- z )`
- `/     ( x y -- z )`
- `mod   ( x y -- z )`
- `>     ( x y -- t/f )`
- `eval  ( list -- ... )`
- `int?  ( x -- t/f )`
- `list? ( x -- t/f )`

The `Gloom` class is a read-evaluate-print loop (REPL) for the Gloom language. It will read an expression as a line from standard in, pass the tokenized line to the interpreter, and then print the resulting stack. This is how you will interact with the interpreter and test your program by hand.

The `GloomTests` class runs a set of test expressions, comparing the resulting stack to a given answer. You are free to add to this set of tests for your own purposes.

Hints

As lists may be nested to an arbitrary depth, it may be helpful to envision parsing input recursively. You begin by parsing the expression, placing literals and operators in to a list. When an opening square bracket is encountered, you recursively call the parser which then parses input until a closing square bracket is encountered. Once the list has been completely parsed the recursive call returns the list of tokens.

Parsing comments is identical to parsing lists, except comments are discarded after being parsed.

You may want to implement an operator named clear which resets the interpreter to its initial state. This will help with interactive debugging.

Grading

The point breakdown is as follows.

25	Stack correctly implemented
25	List correctly implemented
50	Gloom functionality passes all given tests

The set of tests for grading are given in the `GloomTest` class. The main method will run the set of tests and report the results.

Your code should conform to the courses coding style especially those concerning variable names, whitespace, and class, (main) method, and inline comments. You can lose up to 30 points for bad style or commenting.

Reminder — a program which fails to compile on the lab machines (under Java 1.6) will receive 0 points.

Submission

1. Create a zip file of your program using the utility `zip`.

```
> zip -r Program07.zip ...
```

2. Run the submit program.

```
> submit
```

   A text-based interface will appear in the terminal.

3. Use the arrow keys and enter to select a class (cs1131), section (sec1), and assignment (Program07).
4. Use the arrow keys to navigate to the file you would like to submit. Use the spacebar to select the file Program07.zip (an astrisk will appear next to the filename). If the filename is incorrect or you attempt to submit a different file, submit will not accept it.
5. Hit return to submit the selected file(s) and confirm the submission.
6. Verify that the file appears in the "Already Submitted" window pane.

You can use submit at any time to verify that the files have been turned in. The date/time of each submission is also listed.