[P08] (Doom and) Gloom Part II

Due: 12:01am, Monday November 11th.

Description

In order to complete our implementation of the Gloom programming language, we must add operators for branching, looping, and defining other operators. We will also be adding operators for operating on lists so that they can be constructed at runtime.

Operator Definition

The operator definition operator in Gloom is `!`. To define an operator you specify two lists. The first list contains a list of operator names and the second list contains the operation you wish to associate with the operator names.

The primary use of the `!` operator is to define operators (as methods). For example

```
[ increment ] [ 1 + ] !
[ decrement ] [ 1 - ] !

42 increment 42 decrement
```

would output the values 43 and 41.

Operator definitions can be used as variables as well. For example

```
[ a b c ] [ 1 ] !
[ b ] [ 6 ] !

a b c
```

would output the values 1, 6, 1.

Nested use of operator definitions is possible as well. An operator could internally define and use variables, perhaps to simplify the use of the stack.

Branching

There is only one branching operator in Gloom: `if`. The `if` operator takes as arguments a boolean flag, a list to evaluate if the flag is true, and a list to evaluate if the flag is false.

As an example, a simple if statement in a operator definition follows.

```
[ to-zero ] [ dup 0 > [ 1 - ] [ 1 + ] if ] !
```

The `to-zero` operator will either increment or decrement the value on the stack depending on if the number is positive or negative. There is a small bug in the above definition, where the value 0 will be incremented, thus incrementing the value away from zero. To fix it, we add an additional conditional.

```
[ to-zero ] [ dup 0 > [ 1 - ] [ 0 over > [ 1 + ] [ ] if ] if ] !
```

The above would be slightly simpler with a valid definition of `<` or `=`. This will be a task in your next lab.

Looping

There is only one looping operator in Gloom: `loop`. The `loop` operator takes only a single argument, a list. The list is evaluated repeatedly until a stoping condition has been reached.

In order to indicate if the loop should continue, the list argument must always leave a boolean flag on the top of the stack specifying if the loop will continue of not. The `loop` operator always evaluates the list at least once. After each evaluation the operator pops the boolean flag off the stack and continues only if that operator is true.

A simple loop might be one which counts down from a specified value, dumping each value on to the stack.

```
[ iota ] [ [ dup 0 > [ dup 1 - -1 ] [ 0 ] if ] loop ] !
```

This type of loop is the base for a variety of other structures including a `for` loop.

From this simple structure it is possible to construct any other type of loop. However you'll find that loop structures such as `for` and `while` are less useful

in a functional language. Instead they are replaced with more specific looping patterns such as `map` (modifying each value of a list by some operation) and `reduce` (accumulating values of a list in some way). For example,

```
[ 1 2 3 4 ] [ 2 * ] map
```

would output the list `[ 2 4 6 8 ]` and

```
[ 1 2 3 4 ] 0 [ + ] reduce
```

would output the value 10.

Lists

The stack must only function with references to lists, and not the lists themselves. The distinction is necessary as list operators such as `insert`, `remove`, and `set` rely on the behavior. Performance is also saved by forgoing a list copy with every `dup`.

Producing a copy of a list can be done with the `copy` operator which takes a list and creates a new list with identical elements. The copy should be *shallow*; if the list to be copied contains other lists, those nested lists are not copied. This is consistant with the reference semantics of lists.

Interacting with lists can be done with five operators that mirror the list methods you have already implemented: `size`, `get`, `set`, `insert`, and `remove`. The stack effect for each operator is given in the requirements section.

The `append` operator appends the top list on the stack on to the second list and stores the result in to a new list. Pay close attention to the order and behavior of this operator. It is easy flip the order of the lists (producing a prepend).

Requirements

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.

In addition to the operators presented in the previous assignment, your interpreter must now handle the following.

- `!` `( list -- ... )`
- `if` `( t/f tlist flist -- ... )`
- `loop` `( -- )`
- `size` `( list -- n )`
- `copy` `( list -- list )`
- `get` `( i list -- val )`
- `set` `( val i list -- )`
- `remove` `( i list -- val )`
- `insert` `( val i list -- )`
- `append` `( list list -- list )`

Hints

When defining operators with `!`, it might be helpful to use a data structure which maps the operator name with the operator's definition. A `Map` should be suitable for encoding this type of relationship. However, why should we stop with operators defined with `!`? With a mapping from operator to definition, it should be possible to build your primitive operators with the same structure.

Grading

The point breakdown is as follows.

100	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 Program08.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 (Program08).

4. Use the arrow keys to navigate to the file you would like to submit. Use the spacebar to select the file Program08.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.