[L7] Concatenative Programming

Navigate to your CS1131 lab directory and create a new directory for this week's lab programs.

```
[userid@domain ~] mkdir cs1131/labs/lab7
[userid@domain ~] cd cs1131/labs/lab7
```

Getting Started

Whats a language without a library?

Up until this point we have been worried about the language and have developed all of our tests using primitive operators at the expense of concision and readability. It is now time to develop a standard library for Gloom to simplify the development of Gloom programs.

Over the course of the lab you will be completing five files: `shuf`, `cond`, `list`, `comb`, and `samp`. By convention we will be naming Gloom programs with the suffix `.gloom`.

Remember that the Gloom REPL will parse files given on the command line. Once you have completed a problem be sure to run the REPL with the library included. This is how your programs will be tested.

Before the lab's end, you must zip together your solutions and submit them. Instructions are given at the conclusion of the lab. If you finish all the problems during your lab session you are free to work on other homework or leave early.

Problems

First, we need to write more powerful stack shuffling words. In `shuf.gloom`:

1. Define the operators `pick` and `nip`.

   The operator `pick` duplicates the 3rd value on the stack and has stack effect `( x y z -- x y z x )`. It is the natural extension of the `dup` and `over` operators.

   The operator `nip` deletes the 2nd value on the stack and has stack effect `( x y -- y )`. It is the natural extension of the `drop` operator.

2. Define the operators `2dup`, `3dup`, `2drop`, and `3drop`.

   The operators `2dup` and `3dup` duplicate the top two and three values on the stack. `2dup` has stack effect `( x y -- x y x y )` and `3dup` has stack effect `( x y z -- x y z x y z )`.

   The operators `2drop` and `3drop` delete the top two and three values on the stack. `2drop` has stack effect `( x y -- )` and `3drop` has stack effect `( x y z -- )`.

Next we need to define a set of tools which simplify the use of booleans and conditionals in Gloom. In `cond.gloom`:

3. Define the operators `true` and `false`, as well as `and`, `or`, and `not`.

   The operators `true` and `false` push their corresponding integer values on to the stack.

   The operators `and` and `or` (stack effect `( t/f t/f -- t/f )`) take two boolean values and return the result of each boolean operation. The operator `not` is the unary boolean operator with stack effect `( t/f -- f/t )`.

4. Define the operators `<=`, `<`, `>=`, `=`. All four operators have stack effect `( x y -- t/f )`. The program `5 6 <=` would result in the value `-1` (true).

   All four operators can be defined using the primitive comparison operator `>` and various stack shuffling operators.

5. Define the operators `even?` and `odd?`. Both operators have stack effect `( x -- t/f )`.

The primitive types which operate on lists are somewhat clumsy. In particular, both `insert` and `remove` do not leave a reference to the list on the stack, which is often the desired behavior. This problem is meant to provide viable alternatives. In `list.gloom`

6. Define the operators `insert*` and `remove*`.

   The operator `insert*` inserts the specified value at the specified index, leaving a reference to the list on the stack. The primitive `insert` has stack effect `( value index list -- )` whereas the alternative `insert*` has stack effect `( value index list -- list )`.

   The operator `remove*` removes the value at the specified index, leaving a reference to the list on the stack. The primitive `remove` has stack effect `( index list -- )` whereas the alternative `remove*` has stack effect `( index list -- list )`.

Now we need capabilities which provide structured programming using lists as programs. In `comb.gloom`:

7. Define the operators `keep` and `2keep`.

   The `keep` operator takes a value and a list, evaluates the list, and finally restores the value on to the stack. The stack effect for `keep` is pretty complicated:

   ```
   ( ... a x list: ( ... a x -- ... b ) -- ... b x).
   ```

   The program `1 3 [ 1 - ] keep` would result in the values 1, 2, and 3. This is because `keep` first stores away a copy of the value 3, evaluates the list (decrementing 3), and then restores the stored value.

   The `2keep` operator takes two values and a list, evaluates the list, and finally restores the values on to the stack. The stack effect for `2keep` follows from the stack effect for `keep`:

   ```
   ( ... a x y list: ( ... a x y -- ... b ) -- ... b x y ).
   ```

   The program `1 3 [ 1 - ] 2keep` would result in the values 1, 2, 1, and 3. This is because `2keep` first stores away a copy of the values 1 and 3, evaluates the list (decrementing 3), and then restores the stored values.

8. Define the operators `dip` and `2dip`.

   The `dip` operator (stack effect `( x list -- x )`) takes a value and a list, removes the value, evaluates the list, and then pushes the value back on to the stack. It is a way of "dipping" below the value and evaluating commands. The program `2 [ 2 3 + ] dip mod` would result in the value 1.

   The `2dip` operator (stack effect `( x y list -- x y )`) takes two values and a list, removes both values, evaluates the list, and then pushes the values back on to the stack.

9. Define the operator `repeat`.

   The `repeat` operator evaluates the specified list the specified number of times. The stack effect for `repeat` is pretty complicated as well:

   ```
   ( ... n list: ( ... -- ... ) -- ... ).
   ```

   The program `1 4 [ 2 * ] repeat` would result in the value 16.

To finish off our standard library, we would like to provide a non-trivial sample program. In `samp.gloom`:

10. Define the operator `collatz`.

    The operator `collatz` (stack effect `( x -- length )`) returns the length of the collatz sequence with a specified initial value. For example, the program `3 collatz` would result in the value 8.

Submit

1. Navigate to the directory containing your solutions and create a zip file using the utility `zip`.

```
> zip -r Lab7.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 (Lab7).

4. Use the arrow keys to navigate to the file you would like to submit. Use the spacebar to select the file Lab7.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.