Preliminary information

This is your second Scam assignment. To run your code, use the following command:

`scam FILENAME`

or

`scam -r FILENAME`

where `FILENAME` is replaced by the name of the program you wish to run. The `-r` option will automatically run a no-argument function named `main` on startup.

All assignment submissions should supply a program named `author.scm`. This program should look like:

```scm
(define (main)
    (println "AUTHOR: Rita Recursion rrita@crimson.ua.edu")
)
```

with the name and email replaced by your own name and email.

For each numbered task (unless otherwise directed), you are to provide a program named `taskN.scm`, with the `N` corresponding to the task number, starting at one (as in `task1.scm`, `task2.scm`, and so on).

You may not use `assignment` in any of the code you write. Nor may you use any looping function such as `while` or `for`. Do not use the comment-out-the-rest-of-the-file comment in your code. On any line of output, there should be no leading whitespace and no trailing whitespace other than a newline (except when otherwise directed).

ANY ATTEMPT TO FOOL THE GRADING SCRIPT, NO MATTER HOW SMALL, WILL RESULT IN A GRADE OF ZERO WITH NO ABILITY TO RESUBMIT. This penalty can be applied retroactively at any time during or after the semester.

Tasks

1. Define a function named `for-loop` that takes a list and a procedure. The `loop` function should repeatedly execute the procedure, supplying as an argument each of the values in the list in turn.

   For example, the call:

   ```scm
   (for-loop (range 0 10 1) (lambda (i) (inspect i)))
   ```

   should produce the following output:

   ```
   i is 0
   i is 1
   i is 2
   ...
   i is 9
   ```

   You will need to implement the `range` function, which returns a list of numbers generated from the arguments. The first argument specifies the start of the range (inclusive) while the second argument specifies the end (exclusive). The third argument is the step size.

   The `for-loop` function takes a list of numbers and passes them, one at a time, to its second argument.

   Example:
2. Currying is the process of providing one or more of the arguments to a function. The result of currying is a new function that accepts none, some, or all of the remaining, unspecified arguments. Define a function, named \textit{curry}, that curries the given function. As an example, the following pairs of expressions should evaluate to the same result:

\begin{verbatim}
(f a b c)
  ((curry f a) b c)

(g v w x y z)
  (((curry g v w ) x) y z)
\end{verbatim}

Note that \textit{curry} is variadic and that the syntax of variadic functions in Scam is different than that of Scheme.

An easy way to implement \textit{curry} is to use the \textit{apply} function, which, when given a function and a list of arguments, calls the given function with the given arguments. The two calls below are equivalent:

\begin{verbatim}
(+ 1 2 3 4)
  (apply + (list 1 2 3 4))
\end{verbatim}

You can determine the number of arguments a function expects by asking for the length of its formal parameter list. You can also ask for the definition name of a function:

\begin{verbatim}
(length (get 'parameters f)) ; determine the parameter count
(get 'name f) ; get the name of the function
\end{verbatim}

Your \textit{curry} function will only be tested with functions taking a fixed number of arguments.

Example:

\begin{verbatim}
$ echo "(define (f a b) (+ a b))" > task2.args
$ echo "1" >> task2.args
$ echo "1" >> task2.args
$ scam -r task2.scm task2.args
  (curry f) is <anonymous(@)>
  ((curry f) 1) is <anonymous(@)>
  (((curry f) 1) 1) is 2
$
\end{verbatim}

Your \textit{main} function will be supplied a two argument function and two arguments and should always report the definition name accurately.
3. Define the following classes and methods:

- Stack: constructor `Stack`; methods `push`, `pop`, `speek`, `ssize`
- Queue: constructor `Queue`; methods `enqueue`, `dequeue`, `qpeek`, `qsize`

Note: any method that would normally modify the state of the data structure has to return a new data structure, instead. All methods must work in amortized constant time. You may assume that every call to a peek method is followed by a pop or a dequeue, as the case may be.

Here is a routine which uses such classes:

```scheme
(define (loop stack queue)
  (define x (readInt))
  (if (eof?)
      (list stack queue)
      (loop (push stack x) (enqueue queue x))
  )
)

(define (popper s)
  (cond
   ((!= (ssize s) 0)
     (inspect (speek s))
     (popper (pop s))
   )
  )
)

(define (dequeuer q)
  (cond
   ((!= (qsize q) 0)
     (inspect (qpeek q))
     (dequeuer (dequeue q))
   )
  )
)

Here is a call:

```
(define oldstream (setPort (open "data.ints" 'read)))
(define data (loop (Stack) (Queue)))
(popper (car data))
(dequeuer (cadr data))
(setPort oldStream)
```

4. TBD
5. TBD
6. TBD
7. TBD
8. TBD
9. TBD
10. TBD

Handing in the tasks

For preliminary testing, send me all the files in your directory by running the command:

```
submit proglan lusth test2
```

For your final submission, use the command:

```
submit proglan lusth assign2
```