CS403: Programming Languages

Assignment 2

Version 1c

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:

```
(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). For this assignment, your *main* functions will need to evaluate an arbitrary number of expressions in the input file.

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:

   ```
   (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 function should run in linear time.

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

   Example:

```
$ cat task1.args
(println (range 0 10 5))
(for-loop (range 0 10 5) (lambda (x) (inspect x)))
$ scam -r task1.scm task1.args
(0 5)
x is 0
x is 5
$ scam -r task1.scm task1.args
$
```

As a reminder, you will need to evaluate every expression in the input file. To do so, you will need version 2.4c of Scam or higher. Use the standard read pattern:

```
(define env this)
(define (iter expr)
     (if (not (eof?)) (begin (eval expr env) (iter (readExpr))))
     )
(iter (readExpr))
```

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 *curry*, that curries the given function. As an example, the following pairs of expressions should evaluate to the same result:

```
(f a b c)
((curry f a) b c)

(g v w x y z)
(((curry g v w ) x) y z)
```

Note that *curry* is variadic and that the syntax of variadic functions in Scam is different than that of Scheme.

An easy way to implement *curry* is to use the *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:

```
(+ 1 2 3 4)
(apply + (list 1 2 3 4))
```

You can determine the number of arguments a function expects by asking for the length of its formal parameter list:

```
(length (get 'parameters f))        ; determine the parameter count
```

Your *curry* function will only be tested with functions taking a fixed number of arguments.

Example:

```
$ cat task2.args
(define (f a b) (+ a b))
(inspect ((curry f) 1))
(inspect (((curry f) 1) 1))
$ scam -r task2.scm task2.args
((curry f) 1) is <function anonymous(@)>
(((curry f) 1) 1) is 2
$
```

3. Define the following classes and methods:

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

   Note: Your classes will be functional and thus 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 (ACT). The size methods can work in linear time, but if you can achieve the ACT time bound, you can get a point of extra credit. Assume a C-style object system, meaning the methods exist outside of the "class" and that the object is passed as the first argument to these methods. Moreover, these methods return an object (not necessarily the same object that was passed in), with the exception of the peek and size methods. Also, it is *not* a requirement that you use Scam's object system. In fact, your implementation will likely be simpler if you do not.

   Here is a testing program which uses such classes:

```
; read some ints and place them in both a stack and queue
(define (loop stack queue)
    (define x (readInt))
    (if (eof?)
        (list stack queue)
        (loop (push stack x) (enqueue queue x))
        )
    )

; empty out a stack, printing the values as they come off
(define (popper s mode)
    (cond
        ((!= (ssize s) 0)
            (if (= mode 1) (inspect (speek s)))
            (popper (pop s) mode)
            )
        )
    )

; empty out a queue, printing the values as they come off
(define (dequeuer q mode)
    (cond
        ((!= (qsize q) 0)
            (if (= mode 1) (inspect (qpeek q)))
            (dequeuer (dequeue q) mode)
            )
        )
    )

(define (main)
    (define oldstream (setPort (open (get ScamArgs 1) 'read)))
    (define mode (get ScamArgs 2))
    (define data (loop (Stack) (Queue)))
    (setPort oldStream)
    (println "popping...")
    (popper (car data) mode)
    (println "dequeuing...")
    (dequeuer (cadr data) mode)
    )
```

   Example:

```
$ cat task3.args
(println (speek (pop (pop (push (push (push (Stack) 3) 2) 1)))))
(println (qpeek (dequeue (dequeue (enqueue (enqueue (enqueue (Queue) 1) 2) 3)))))
$ scam -r task3.scm s task3.args
3
3
$
```

4. Define a function name *let*->lambdas* that, when given the source code of a function whose body is a *let**, converts it to the source code of a function with nested lambdas. For example,

   ```
   (let*->lambdas `(define (f x) (let* ((y 2) (z y)) (+ x y z))))
   ```

   should return the list:

   ```
   (define (f x) ((lambda (y) ((lambda (z) (+ x y z)) y)) 2))
   ```

   The *let** may have one or more variables in its variable definition list and you may assume the given function, if it contains a *let**, has the *let** as its body. You do not need to convert *let**s recusively.

   Example:

   ```
   $ cat task4.args
   (println (let*->lambdas `(define (f x) (let* ((y 2) (z y)) (+ x y z)))))
   $ scam -r task4.scm task4.args
   (define (f x) ((lambda (y) ((lambda (z) (+ x y z)) y)) 2))
   $
   ```

5. Implement two functions, *create* and *pred*. The first takes an Arabic number and produces the equivalent Church numeral. The second produces the predecessor of the given Church numeral. Use the style of Church numerals as found in the textbook. Both functions should implement iterative processes.

   You will likely need to peruse the web for this task.

   Example:

   ```
   $ cat task5.args
   (println (((create 4) (lambda (x) (+ x 1))) 0))
   (println (((pred (create 4)) (lambda (x) (+ x 1))) 0))
   $ scam -r task5.scm task5.args
   4
   3
   $
   ```

6. Define a function, named *powerSet*, which produces the power set of a set in *canonical* order. The power set of:

   ```
   (s e t)
   ```

   can be represented as:

   ```
   (nil (s) (e) (t) (s e) (s t) (e t) (s e t))
   ```

   or:

   ```
   (nil (t) (e) (e t) (s) (s t) (s e) (s e t))
   ```

   as well as many other ways. Only the first representation shown is in canonical order. Enforcing canonical order at any step should only take linear time.

   You may assume that the *powerSet* function is only passed a list of symbols, as in:

   ```
   (powerSet '(a b))
   ```

   Example:

   ```
   $ cat task6.args
   (println (powerSet '(a b)))
   $ scam -r task6.scm task6.args
   (nil (a) (b) (a b))
   $
   ```

   Your *main* function will need to set the display of *nil* to the symbol `nil`:

   ```
   (setNilDisplay 'nil)
   ```

7. Do exercise 2.37, defining *matrix-*-vector*, *transpose*, and *matrix-*-matrix*, along with all other support functions. Unlike the exercise, store the matrix in column-major format (as is done in Fortran). In column-major format, the example matrix would be represented as `((1 4 6) (2 5 7) (3 6 8) (4 6 9))`.

   You must follow the style of the book. Note that we are only changing *how* the matrix is represented; the matrix location $M_{i,j}$ still refers to row i, column j.

   Example:

   ```
   $ cat task7.args
   (println (transpose '((1 0) (0 1))))
   (println (matrix-*-matrix '((1 2) (3 4)) '((1 0) (0 1))))
   $ scam -r task7.scm task7.args
   ((1 0) (0 1))
   ((1 2) (3 4))
   $
   ```

8. Consider this node and tree constructor and tree displayer:

   ```
   (define (node value left right)
       (define (display) (print value))
       this
       )

   (define (newBST value)
       (node value nil nil)
       )

   (define (displayBST root)
       (define (iter root indent)
           (if (valid? root)
               (begin
                   (iter (root'right) (string+ indent "    "))
                   (print indent)
                   ((root'display))
                   (println)
                   (iter (root'left) (string+ indent "    "))
                   )
               )
           )
       (iter root "")
       )
   ```

   Define an *insertBST* function that takes a binary search tree and a value and returns a new binary search tree that includes that value. You may assume that only unique values are inserted into the tree.

   Example:

   ```
   $ cat task8.args
   (define t0 (newBST 5))
   (define t1 (insertBST t0 2))
   (define t2 (insertBST t1 8))
   (displayBST t2)
   $ scam -r task8.scm task8.args
       8
   5
       2
   $
   ```

   Note that the BST is displayed sideways.

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
```

Change log

- Version 1c - Mar 20 09:17:45 - fixed example output in task 3 and added extra credit subtask
- Version 1b - Mar 15 10:19:38 - fixed peek description in task 3
- Version 1a - Mar 14 17:00:53 - clarified the object system in task 3