Preliminary information

This is your third 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).

You may use assignment, unless otherwise directed. You may not use a looping function such as while or for, unless otherwise directed. Do not use the comment-out-the-rest-of-the-file comment in your code. Make sure you submit text files, not binary files. 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 predicate named scoping that determines whether a symbol is bound, free, or undefined with respect to an object. The call:

   ```
   (scoping 'y ((define (zurp x) (define y (+ x 1)) this) 5))
   ```

   should return the symbol **bound** because y is a locally defined variable. The call:

   ```
   (scoping '*' ((define (zarp x) this) 5))
   ```

   should return the symbol **free** because * is not locally defined in the function zarp.

   Finally, the call:

   ```
   (scoping 'cs403sucks ((define (zirp x) this) 5))
   ```

   should return the symbol **undefined** because who would define a variable like that?

   Use the function ppTable to help you decompose scopes/objects. Do not use catch in your solution.

Example:
2. Define a function named replace that replaces all occurrences of a given symbol in a function definition with a given value. You can obtain the body of a function definition as follows:

```
(define (square x) (* x x))
(define body (get 'code square))
(inspect body)
```

The inspection displays the list (begin (* x x)) whose car is the symbol begin and whose cadr is the list (* x x).

You can update the body of the function as follows:

```
(set 'code '(begin (+ x x)) square)
(inspect (square 5))
```

The inspection should display the value 10. A call to replace might look like:

```
(replace square (list '* + '+ *))
```

This call would replace the times symbol with the adding built-in and the plus symbol with the multiplying built-in.

You can use the replace function to speed up the execution of any function.

Your replace function needs to work recursively. That is to say, it needs to replace the symbol in all local function definitions, including lambdas. You may find the predicate functions object? useful this feature as you should not descend into these kinds of objects. Neither should you process any quoted expression. You may assume the given function performs no assignments of any kind.

Example:

```
$ cat task2.args
(define (abs n) (if (< n 0) (- n) n))
(replace abs (list '< < '- -))
(inspect (get 'code abs))
$ scam -r task2.scm task2.args
(get (quote code) abs) is (begin (if (<builtin <(@)> n 0) (<builtin -(@)> n) n))
$
3. Define a double-ended queue class called `deque`. Your class should support the following methods:

- `((dq\'enqueueFront) value)`
- `((dq\'enqueueBack) value)`
- `((dq\'enqueueIndex) index value)`
- `((dq\'dequeueFront))`
- `((dq\'dequeueBack))`
- `((dq\'dequeueIndex) index)`
- `((dq\'display))`
- `(dq\'size)`

The enqueue methods should return the item enqueued while the dequeue methods should return the item dequeued. All methods should run in constant time except `enqueueIndex` and `dequeueIndex`. The `enqueueIndex` method runs in constant time when the index is a constant distance from the front and in linear time otherwise. Likewise, the `dequeueIndex` method runs in constant time when the index is a constant distance from the back and in linear time otherwise.

Example:

```sh
$ cat task3.args
(define dq (deque))
((dq\'enqueueFront) 3)
((dq\'enqueueBack) 5)
(inspect ((dq\'dequeueFront)))
(inspect ((dq\'dequeueBack)))
(inspect (dq\'size))
(print "-\"")
((dq\'display))
(println "-\")
$ scam -r task3.scm task3.args
((dq (quote dequeueFront))) is 3
((dq (quote dequeueBack))) is 5
(dq (quote size)) is 0
-[]-
$ 
```
4. NAND is considered a universal set as any logical expression can be built solely with NAND gates. Define the constructors inverter, and-gate, or-gate, nor-gate, xor-gate, and xnor-gate, built entirely from NAND gates. Name your NAND gate constructor nand-gate. Assume a nand-gate delay of 6 time units and hard-wire this number into your nand-gate function. Use the logic gate system of the text to implement NAND and the gates built with NANDs. If you wish, you can use Scam’s object system instead of the dispatch-function style of the text, which is available online. In either case, you must provide compatible make-wire, get-signal, set-signal!, make-agenda and propagate functions.

Use your gates to build a half-adder and a full-adder, as in the text. Place your nand-gate function by itself (with no other functions) in a file named nand.scm and include that file in your task file:

(include "nand.scm")

Example:

$ cat task3.args
(define the-agenda (make-agenda))
(define input-1 (make-wire))
(define input-2 (make-wire))
(define sum (make-wire))
(define carry (make-wire))
(probe 'sum sum)
(probe 'carry carry)
(half-adder input-1 input-2 sum carry)
(set-signal! input-1 1)
(propagate)
(set-signal! input-2 1)
(propagate)
$ scam -r task4.scm task4.args
...
$

5. Define an n-ary mutex constructor using Scam’s binary semaphore. Name your mutex constructor mmutex. Before starting this task, please read the chapter on Concurrency in The Sway Reference Manual first.

Example calls:

(define m (mmutex 3))
((m'p))
((m'v))

The function call ((m'p)) should return the symbol ACQUIRED. Your mutex should reject release calls if the process performing the release is not one of the processes holding the mutex. Therefore, the call ((m'v)) should return the symbol RELEASED or the symbol FORBIDDEN, as appropriate. You can use the gettid function to access the ID of the thread asking for the mutex. The sleep function may be useful for testing your mutex.

Example:

$ cat task5.args
(define m (mmutex 1))
(inspect ((m'p))
(inspect ((m'v))
$ scam -r task5.scm task5.args
((m'p)) is ACQUIRED
protected section
((m'v)) is RELEASED
$
6. Define a function named \textit{smush} that takes a stream of the form \((a, b, c, d, \ldots)\) and the operator \(\Phi\) and returns the stream \(((0, a), (1, a\Phi b), (2, a\Phi b\Phi c), (3, a\Phi b\Phi c\Phi d), \ldots)\). You will also need to define a stream displaying function named \textit{sdisplay}. See the example below for its behavior.

Example:

\begin{verbatim}
$ cat task6.args
 (define ones (cons-stream 1 ones))
 (sdisplay 12 (smush ones +))
 (println)
$ scam -r task6.scm task6.args
(0,1,1,1,2,2,1,3,3,1,4,...)
\end{verbatim}

For every triplet, the first number, \(i\), is the index of the second number in the original stream. The third number is the accumulation of the first.

7. Define a series of functions for mathematically manipulating a quadratic equation represented as a stream. The first, \textit{quad}, takes the three coefficients of the quadratic plus a step value. For example, \((\text{quad} 1 3 -4 0.01)\) would produce the stream representing the equation \(f = x^2 + 3x - 4\). The step argument, \(s\), gives the distance between evaluations of the quadratic, starting at zero. The stream produced would be equivalent to \((f(0), f(s), f(2s), f(3s), \ldots)\).

The second function, \textit{integrate}, produces a stream representing the integral of the given stream. You should use the trapezoid method to compute the area under the curve. The first element of the integrated stream would be zero; the second element would be computed from a trapezoid having \(f(0)\) as the left height and \(f(s)\) as the right height. The \textit{integrate} function takes two arguments, the stream to integrate and the step size. The \(i^{th}\) element of the resulting stream should approximate:

\[
\int_0^{iss} f(x)\,dx
\]

The third function, \textit{derivate}, produces a stream representing the derivative of the given stream. It should reverse the integration process. An integrated quadratic stream passed to derivate should produce the original quadratic stream. The \textit{derivate} function takes two arguments, the stream to differentiate and the step size.

A fourth function, \textit{same-stream?} will help you decide if a differentiated integral stream is the same as the original stream. Given two streams, \(s1\) and \(s2\), a count \(n\), and a positive threshold \(t\), returns true if all of the first \(nth\) elements (piecewise) differ by less than \(t\) and false otherwise.

Example:

\begin{verbatim}
$ cat task7.args
 (define qs (quad 1 0 0 1))
 (define is (integrate qs 1))
 (define ds (derivate is 1))
 (sdisplay 5 qs) (println)
 (sdisplay 5 is) (println)
 (sdisplay 5 ds) (println)
 (inspect (same-stream? qs ds 10 .001))
$ $(0.000000e+00,1.0000000000,4.0000000000,9.0000000000,16.0000000000,\ldots)
(0.000000e+00,0.5000000000,3.0000000000,9.5000000000,22.0000000000,\ldots)
(0.000000e+00,1.0000000000,4.0000000000,9.0000000000,16.0000000000,\ldots)
(same-stream? qs ds 10 0.0010000000) is #t
$ \end{verbatim}
Consider this series:

\[ 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \cdots \]

Define a function named `mystery` that returns the stream that holds the terms of the above series for a given \( x \). Define a function named `ps-mystery` that produces the stream of partial sums of \((\text{mystery } x)\). Define a function named `acc-mystery` that accelerates \((\text{ps-mystery } x)\) using the Euler transform. Define a function named `super-mystery` that produces a super accelerated stream using a tableau of ever-accelerated partial sum streams. All of these function takes \( x \) as their sole arguments. Define a function named `sym-mystery` that prints the symbolic result (as a LaTeX math expression) of the series sum, when taken out to infinity. For example, if the answer is \( \pi \), then \((\text{mystery } x)\) is $\pi$ would be printed. If the answer is \( \log \pi \), then \((\text{mystery } x)\) is $\log \pi$ would be printed. In this last example, note the lack of parentheses. If there is more than one reasonable way to represent the answer, prefer the one with the fewer number of characters in the LaTeX expression. Prefer symbolic renderings over numeric ones and integers over integers rendered as real numbers. If you are unsure which rendering is best, ask me privately. Note that the dollar signs in the output are mandatory. Do not share this answer in any way.

Example:

```
$ cat task8.args
(define x 0)
(sdisplay 5 (mystery x)) (println)
(sdisplay 5 (ps-mystery x)) (println)
(sdisplay 5 (acc-mystery x)) (println)
(sdisplay 5 (super-mystery x)) (println)
(symbolic-mystery)
$ scam -r task8.scm task8.args
(1.0000000000,-0.0000000000,0.0000000000,-0.0000000000,0.0000000000,...)
(1.0000000000,1.0000000000,1.0000000000,1.0000000000,1.0000000000,...)
(1.0000000000,1.0000000000,1.0000000000,1.0000000000,1.0000000000,...)
(1.0000000000,1.0000000000,1.0000000000,1.0000000000,1.0000000000,...)
(mystery x) is $\log \pi$
$}
```

Of course, \( \log \pi \) is not the answer. Or is it?

9. Exercise 3.71 in the text. Define a function named `ramanujan` that produces a stream of Ramanujan numbers. The function takes no arguments.

Example:

```
$ cat task9.args
(sdisplay 1 (ramanujan)) (println)
$ scam -r task9.scm task9.args
(1729,...)
$}
```

Handing in the tasks

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

```
submit proglan lusth test3
```

For your final submission, use the command:

```
submit proglan lusth assign3
```