CS403: Programming Languages

Assignment 3

Preliminary information

This is your third Scam assignment.

At the top of your assignment, place a definition similar to the following:

```
    (define (author)
        (println "AUTHOR: Izzy Iterator iiter@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 function with a name of the form *runN*, with the *N* corresponding to the task number (as in *run1*, *run2*, etc.). This function is in addition to all other requested functions. These run functions will test your implementation and are to take no arguments. For example, if task 5 is:

> 5. Implement factorial so that it implements a recursive process. Name your function *fact*.

you should provide a *run* function similar to:

```
    (define (run5)
        (print "The result of (fact 3) is " (fact 3))
        (println "[it should be 6]")
        ...
        )
```

Woe betide students who provide insufficient testing should their implementations prove to be incorrect! If you do not complete an exercise, do not provide a run function for that exercise. If you omit the run function corresponding to an exercise, it will be assumed that you are skipping that part and you will receive no credit for that exercise.

When you have completed testing of your run functions, comment out any calls to them (but do not comment out the definitions).

I will provide a test script which performs minimal testing of your implementation. If your program does not pass the provided test script without failing, I will not grade your exercise and you will receive zero credit for the assignment.

You may use assignment and loops for this set of tasks, unless otherwise directed.

You will also be graded on style, including, but not limited to, the proper nesting of helper functions that are not expected to be called directly.

Tasks

1. Define a function named *nonlocals*, which returns a list of all the nonlocal variables referenced in a function definition. For example,

   ```
   (define (square x) (* x x))
   (inspect (nonlocals square))
   ```

   should produce the following output:

   ```
   (nonlocals square) is (begin *)
   ```

   since `*` is the only nonlocal variable in the function body. The symbol *begin* is also in the list because Scam wraps the body of a function with a *begin* to make the function body appear to be a Scam block. **Each nonlocal variable should appear exactly once in the resulting list.** Given a closure, one can retrieve the formal parameters and body with:

```
(get 'parameters square)
(get 'code square)
```

respectively. Both the parameters and the function body are simple Scam lists which can be manipulated with c*ar*, *cdr*, *set-car!*, and *set-cdr!*. Note: a symbol anywhere in the body represents a variable if it is not wrapped in a call to *quote*.

You may assume that all local variables are defined before use.

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 '* +)
   ```

   You can use the *replace* function to speed up the execution of any function. Your run function should demonstrate such a speed up with a recursive Fibonacci function.

   Your *replace* function needs to work recursively. That is to say, it needs to replace the symbol in any local function definitions, including lambdas. The function should only replace *r-value* occurrences of the symbol.

3. *Note: you may want to do this task last, as it is quite involved. On the other hand, you need an AVL tree for your DPL and you should be able to easily translate your Scam code to your language.*

   Define a binary search tree class named *avl*, which implements a self-balancing AVL tree. Your class should support the following methods:

   ```
   insert
   find
   size
   statistics
   ```

   The *statistics* method does a level order traversal, printing out, in turn, each node value and the height of the left subtree minus the height of the right subtree. The two values are to be separated by a colon. The node outputs themselves are separated by spaces and the entire output is terminated by a newline.

   The other methods have their obvious meanings.

   The *size* method should run in constant time. The *insert* and *find* methods should run in $\theta(\log n)$ time.

   Examples:

   ```
   (define t (avl))
   ((t 'insert) 3)
   ((t 'insert) 4)
   ((t 'insert) 5)
   ((t 'insert) 1)
   ((t 'insert) 0)
   ((t 'find) 5)     ; should return #t
   ((t 'find) 7)     ; should return #f
   ((t 'size))       ; should return 5
   ((t 'statistics)) ; should print 4:1 1:0 5:0 0:0 3:0
   ```

   Assume your tree will be populated with unique values. Do no error checking; assume good input. You must use Scam's object system and you must base your implementation on the pseudocode found at: `http://beastie.cs.ua.edu/avl/index.html`.

4. Using the imperative style of the text, implement a constraint network for the formula for determining the gravitation force between two objects:

$$f = G\frac{m_1 m_2}{r^2}$$

Name your network constructor `gravity`. Use a value of 0.00667300 for G. The function *gravity* should take four connectors as arguments, corresponding to the force (f), the mass of object 1 (*m1*), the mass of object 2 (*m2*), and the distance between the two objects (r).

Provide the following accessor and mutator functions *get-value*, *set-value!*, and *forget-value!* as described on page 289 of the text.

5. Define a one-time synchronization barrier object using using Scam's binary semaphore. Name this function *barrier*. You can use the *gettid* function to access the ID of the thread asking for the mutex.

Example calls:

```
(define b (barrier))
((b'set 3))             ; set the barrier for three threads
...
((b'install))           ; the actual barrier
((b'remove))            ; remove the barrier
```

For this example, once three threads reach the installation point, the barrier is removed and all threads can pass.

Your run function should test your barrier. The *sleep* function may be useful for this purpose.

6. The *big gulp stream* is the infinite stream of integers whose only prime factors are 7 and 11. The first few elements of the stream are `[7,11,49,77,...]`. Name the stream constructor *big-gulp* and provide a stream-display function:

```
(define bgs (big-gulp))
(stream-display bgs 4) ;should print [7,11,49,77,...]
```

7. Given a stream that represents:

$$x^2 + 3x - 4$$

show that $\dfrac{\mathrm{d} \int x^2 + 3x - 4 \,\mathrm{d}x}{\mathrm{d}x}$ is equal to: $x^2 + 3x - 4$.

Show this by subtracting the integrated-then-differentiated stream from the original stream and demonstrating that the resulting stream is composed of zeros (or small numbers near zero). Your run function should display a sufficient portion of the original stream, the integrated stream, the derivative of the integrated stream, and a stream representing the difference of the original stream and the differentiated stream. Name your streams *poly*, *intPoly*, and *divIntPoly*, and *difference*, respectively. The *intPoly* stream should be produced by passing *poly* through an *integrating* function and the *divIntPoly* strean should be produced by passing the resulting stream through a differentiating function. Name your integrating function *integral*, your differentiating stream, *differential*, and your stream difference function *subStreams*.

To start, create the *poly* stream by making use of the *signal* function:

```
(define (signal f x dx)
    (cons-stream (f x) (signal f (+ x dx) dx))
    )
```

Start your *poly* stream at zero. Next integrate and differentiate this stream using the *integral* and *differential* functions, respectively:

```
(define (integral s dx)
    ...
    )

(define (differential start s dx)
    ...
    )
```

Note: the integral stream should show the total area under the curve from the starting value to the i^{th} *dx*. For example, the stream:

```
[0 1 4 9 16 25 ...]
```

represents the $y = x^2$ polynomial with a *dx* of 1. The integral of this stream should be:

```
[0 1 5 14 30 55 ...]
```

using a crude rectangular approximation of the area under the curve between two adjacent points. Also, you will need to use the same *dx* for the *signal*, *integral*, and *differential* functions. Finally, you will need to pass in the first value of the original stream to the differential function so you can unwind the integral stream.

Extra credit for more sophisticated integral and differential functions.

8. Consider this series:

$$1 - \tfrac{x^2}{2!} + \tfrac{x^4}{4!} - \tfrac{x^6}{6!} + \tfrac{x^8}{8!} - ...$$

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 `(mystery x)`. Define a function named *acc-mystery* that accelerates `(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.

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