CS403: Programming Languages

Assignment 3

Version 3b

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).

Tasks

1. Define a function named *staticScope*, which, when given an object, displays all the bindings in the static chain of that procedure object. The variable *this* is always bound to the current static scope. The *cadr* of *this* yields the set of variables bound in the current scope, while the *caddr* yields the corresponding list of variable values. The *cadr* of the lit of values yields the enclosing static scope.

   For example, consider executing the following code:

   ```
   (define (f x)
       (define (g y)
           (define zz 3)
           (staticScope 3 this)
           (+ x y)
           )
       g
       )

   (define z (f 3))

   (z 4)
   ```

   The call to *staticScope* should produce (at least) the following output:

```
#[environment L0]
    zz : 3
    y : 4
#[environment L1]
    g : #[user defined function (g y)]
    x : 3
#[environment L2]
    f : #[user defined function (f x)]
```

There may be additional variables depending on what else is defined in the program.

The first argument to the function gives the number of scope levels to be printed. Giving a 0 as the number of levels should result in all levels being printed. `L0` labels the innermost scope, while `L1` labels the immediate enclosing scope, and so on. You should skip over the predefined variables and only list the explicitly defined local variables. Note there may be higher environments in your result.

Example:

```
$ echo "(define (f x) (staticScope 1 this))" > task1.args
$ echo "(f 0)" >> task1.args
$ scam -r task1.scm task1.args
#[environment L0]
    x : 0
$
```

Both expressions read in by *main* should be evaluated.

2. Define a function named *compile* that replaces all the non-local variables in a function body with the values found in the definition environment. Should any of those values be function objects themselves, those objects would need to be compiled as well. The *compile* function should return its modified argument. The behavior of the compiled function must be unchanged, except it should run faster. You can retrieve the body and definition environment of a function with:

```
(define body (get 'code square))
(define denv (get '__context square))
```

respectively.

Here's an example. Suppose we have the following definitions:

```
(define (square x) (multiply x x))
(define (multiply a b) (* a b))
```

Then:

```
(include "pretty.lib")
(compile square)
(pretty square)
(pretty multiply)
```

should output:

```
(define (square x)
    (<function multiply(a b)> x x)
    )
(define (multiply a b)
    (<built-in *(@)> a b)
    )
```

You will need to test whether or not the value of a symbol is a user-defined function or not (use the *closure*? function). Note, that if *multiply* references a user-defined function, then that function would also be compiled.

Use *eval*, *catch*, and *error*? to decide if a symbol in the function body is bound or free. If it is bound, leave the symbol in the body unchanged. Otherwise, replace the symbol with its value. You'll need to test for the existence of a symbol in the definition environment of the function being compiled. You need not worry about nested functions. Note: closures are also objects in Scam.

You should use *set-car*! and *set-cdr*! to make replacements.

Include the *pretty.lib* library to gain access to the *pretty* function.

Note: a symbol in the body represents a variable if it is not wrapped in a call to *quote*.

Example:

```
$ echo "(define (f a b) (+ a b))" > task2.args
$ scam -r task2.scm task2.args
(define (f a b)
    (<built-in +(@)> a b)
    )
$
```

Your *main* should evaluate and then compile the read-in expression. Finally, it should pretty print the compiled expression.

3. Define a function, named *bst*, that constructs a binary search tree. The object that the constructor returns should implement the basic BST methods: *size*, *insert*, *find*, *delete*, *root*, *walk*, and *next*. The *size* method should return the number of nodes in the tree. The *root* method should return the key stored at the root of the tree. The *find* method should return the value associated with the given key and nil if the key is not in the tree. The behavior of the *walk* method is to return the "least" value in the BST, according to the comparator passed to the constructor. The *next* method should return the next value in an in-order traversal of the tree, with subsequent nodes being visited with each successive call to *next*. The *next* method should return nil when the search is exhaused If an insertion or deletion occurs in the middle of the traversal, the behavior of *next* is unspecified. Note that having *find* and *next* return nil is a particularly bad design choice because it disallows nil values being stored in the tree (but it simplifies your task).

   For testing purposes, all methods should handle an empty tree without failure. For those methods for which an empty tree is inappropriate, the return value is unspecified. Here are some sample series calls:

```
(define tree (bst <))           ; smaller keys to the left
(tree 'insert 3 "mary")         ; return value not specified
(tree 'insert 2 "jill")
(tree 'insert 6 "mark")
(tree 'size)                    ; should return 3
(tree 'find 6 =)                ; should return "mark"
(tree 'find 5 =)                ; should return nil
(tree 'walk)                    ; should return "jill"
(tree 'next)                    ; should return "mary"
(tree 'next)                    ; should return "mark"
(tree 'next)                    ; should return nil
(tree 'delete 2 =)              ; should return "jill"
((bst <) 'delete 2 =)           ; return value not specified
```

   Your *main* function should insert each key-value pair supplied and then perform a complete walk of the newly made tree. It should then delete the root node of the tree and then do another complete walk. Each walk should be introduced with a print statement (shown in the example below). A complete walk of a tree should run in $O(n)$ time.

   Example:

```
$ echo "((0 0) (1 1))" > task3.args
$ scam -r task3.scm task3.args
First walk:
0
1
Second walk:
1
$
```

   Note: keys and values may be any type and a walk on an empty tree produces no output.

4. Using the imperative style of the text, implement a constraint network for the formula for determining the velocity right before impact when an object is dropped from a given height. The formula is:

$$v = \sqrt{2gh}$$

where g is the gravitational acceleration and h is the drop height. While g is specified/reported in SI units, h is specified/reported in feet and v should be specified/reported in furlongs per fortnight. The constant g determined from the latitude using the formula:

$$g = g_{45} - \frac{1}{2}(g_p - g_e) \times \cos(2 \times lat \times \frac{\pi}{180})$$

where

- $g_p = 9.832 \frac{m}{s^2}$
- $g_{45} = 9.806 \frac{m}{s^2}$
- $g_e = 9.780 \frac{m}{s^2}$
- lat = latitude (between -90 and 90 degrees)

Name your network constructor *speed*. Your network *speed* should take three connectors as arguments, the velocity, the latitude, and the height, in that order.

Provide the following accessor and mutator functions *get-value*, *set-value!*, and *forget-value!* as described on page 289 of the text. Use a value of 3.14159265358979323846 for π. Use a value of 6012.884753042233335719399 to convert meters per second to furlongs per fortnight. Use a value of 3.28083989501312335958 to convert from meters to feet.

Example:

```
$ echo "50000" > task4.args             #furlongs per fortnight
$ echo "60" >> task4.args               #latitude
$ scam -r task4.scm task4.args
the drop height should be around ??? feet
$
```

Round the result to 12 decimal places. Hint: use the *fmt* function.

5. Define a 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 semaphore.

Example calls:

```
(define b (barrier 3))  ; make a barrier for three threads
...
((b'start))
...
((b'finish))
```

For this example, once three threads reach the starting point, the barrier is opened and the three threads (and only those three threads) can pass. When those three threads reach the finish line, three more threads can pass (and so on).

Do not provide a main function for this task.

6. Define a variadic function named *pfs* that, when given some prime numbers *a*, *b*, ..., *z*, creates an infinite stream of integers whose only prime factors are *a*, *b*, ..., *z*. These integers should have the form $a^A b^B c^C ... z^Z$, where *A*, *B*, *C* and so on range from 0 upwards (except the exponents cannot all be zero). The numbers in this infinite stream should appear in ascending order.

   Example:

```
$ # (stream-display 3 (pfs 2))
$ echo 3 > task6.args
$ echo 2 >> task6.args
$ scam -r task6.scm task6.args
(2 4 8 ...)
$
```

   Constraints: *stream-display* should not print any whitespace after the closing parenthesis.

7. Define a function named *twinPrimes* that produces a stream of twin prime pairs. Two primes are twinned if they differ in magnitude by *n* or less.

   Example:

```
$ # (stream-display 4 (twinPrimes 3)) #display the first two twin prime pairs
$ echo 4 > task7.args
$ echo 3 >> task7.args
$ scam -r task7.scm task7.args
((2 3) (2 5) (3 5) (5 7) ...)
$
```

   Constraints: *stream-display* should not print any whitespace after the closing parenthesis. The twin primes should be ordered by increasing value of the first prime in the pair, with the second prime breaking ties.

8. Consider this summation:

$$\sum_{n=1}^{\infty} (-1)^{n+1} \frac{x}{y^n}$$

   Define a function named *sum* that returns the stream that holds the terms of the above series for a given *x* and *y*. Define a function named *psum* that produces the stream of partial sums of `(sum x y)`. Define a function named *acc-psum* that accelerates *psum* stream using the Euler transform. Define a function named *super-acc-psum* that produces a super accelerated stream using a tableau of ever-accelerated partial sum streams. All of these function takes *x* and *y* as their arguments.

   Example:

```
$ echo 1 >> task8.args          # number of stream elements to display
$ echo 3 >> task8.args          # x
$ echo 5 >> task8.args          # y
$ scam -r task8.scm task8.args
sum returns (0.6000000000 ...)
psum returns (0.6000000000 ...)
acc-psum returns (0.5000000000 ...)
super-acc-psum returns (0.6000000000 ...)
$
```

   Constraints: *stream-display* should not print any whitespace after the closing parenthesis.

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:

   ```
   $ # (stream-display 1 (ramanujan))
   $ echo 1 > task9.args
   $ scam -r task9.scm task9.args
   (1729 ...)
   $
   ```

   Constraints: *stream-display* should not print any whitespace after the closing parenthesis.

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
```