CS403: Programming Languages

Assignment 1

Version 2c

Preliminary information

This is your first 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 file containing 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 example, if task 5 is:

> 5. Implement factorial so that it implements a recursive process. Name your function *fact*. It will take a non-negative integer argument.

you should create a program file named *task5.scm*. The program should look like:

```
(define (main)
    (setPort (open (getElement ScamArgs 1) 'read))
    (define arg (readExpr))
    (println (fact arg))
    )

(define (fact n)
    (if (< n 2) 1 (* n (fact (- n 1))))
    )
```

The expression beginning with *setPort* sets the input file pointer to the file named by the first command line argument. The file should contain a parenthesized list of the arguments to be passed to the *fact* function. The *readExpr* call in the second expression reads this list of arguments and returns them to the apply function, which passes these arguments to the *fact* function. Here is one way to run the *task5* program.

```
$ echo "5" > task5.args
$ scam -r task5.scm task5.args
120
$
```

The filename *task5.args* is the first command-line argument. The `-r` option informs Scam to run the *main* function after the program has been loaded. Note that purpose of your *main* function is to test the functions you are to define; it should contain no other logic.

For printing, it may be of use to know that you can have actual tabs and newlines within a string, as in:

```
(println "
    The quick brown fox

          m
       u       p
    j            e
                   d

    over the lazy dog
")
```

which will print out as:

```

The quick brown fox

      m
   u       p
j            e
               d

over the lazy dog
```

A useful debugging function *inspect*. Here is an example usage:

```
(inspect (+ 2 3))
```

which produces the output:

```
(+ 2 3) is 5
```

Another useful debugging function is *pause*. It takes no arguments, stopping execution until a newline is entered from the keyboard.

If you have trouble loading a file, try using the comment-out-the rest-of-the-file marker (`;$`) in your code. You can move this marker around until you find the malformed expression in your code. **DO NOT** leave a `;$` marker in any code you submit for grading. I will be injecting code at the the bottom of your submissions and having that marker present will comment out this injected code.

You may not use assignment in any of the code you write. Nor may you use any looping function such as *while* or *for*. You may not use lists or arrays, unless otherwise specified.

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. Consider colorizing a value between 0 and 100 so that each integer value corresponds to an CYM color. To determine the CYM color, the intensity of the cyan, yellow, and magenta colors are determined individually. To compute the *cyan* intensity, the integer value is scaled between 0 and 255 according to a quarter cycle of a left-shifted sine wave. For example, a value of 0 would correspond to a cyan value of 255 while a value of 100 would correspond to a cyan value of zero. The *yellow* value is computed likewise with a half-cycle of an inverted up-shifted sine wave. For example, values of 0 and 100 would correspond to a yellow value of 255, while a value of 50 would correspond to a yellow value of 0. Finally, a *magenta* value is computed with a three-quarters of a cycle of an left- and up-shifted sine wav. For example, a value of 0 would yield a magenta value of 255, while a value of 100 would yield a magenta value of 127.5 (converted to 127 - see below). The *sin* and *cos* functions will be useful for this task.

   Use a value of 3.14159265358979323846 for π.

   Your task is to define a function named *cym* which takes a single value as its argument. Your function should return the corresponding CYM values as hexadecimal string. For example, if all the color values are zero, then the function should return the string:

   ```
   #000000
   ```

If all the values are 255, the resulting string should be:

```
#FFFFFF
```

You may find the *string+* function useful for concatenating substrings.

Note: all computed color values should be rounded in the following manner: add 0.00000001 to the color value and then truncate. For example, if the actual value computed by a color function is 138.87645673, the function should report 138. The *int* function can be used for this purpose.

Example:

```
$ echo "0" > task1.args
$ scam -r task1.scm task1.args
(cyan 0) is 255
(yellow 0) is 255
(magenta 0) is 255
(cym 0) is #FFFFFF
$
```

2. The Mandelbrot set (for examples, see `http://www.softlab.ece.ntua.gr/miscellaneous/mandel/mandel.html` is a set of planar points, a point `(x,y)` being in the set if the following iteration never diverges to infinity:

   $$r = r \times r - s \times s + x$$

   and

   $$s = 2 \times r \times s + y$$

   with r and s both starting out at 0.0. While we can't iterate forever to check for divergence, there is a simple condition which predicts divergence: if $r \times r + s \times s > 4$ is ever true, either r or s will tend to diverge to infinity. Processing of a point continues until divergence is detected or until some threshold number of iterations has been reached. If the threshold is reached, the point is considered to be in the Mandelbrot set. Obviously, the higher the threshold, the higher the confidence that the point actually is in the set. The points *not* in the Mandelbrot set can be categorized as to their resistance to divergence. These points are often colorized, as in the previous task.

   Define a function, named *mandelbrot*, that takes a threshold as its single argument. and returns another function that can be used to test whether or not a point is in the Mandelbrot set using the given threshold. The returned function takes two arguments, the x-coordinate, and the y-coordinate of the point to be tested and it returns the resistance (i.e., the number of iterations until the divergence test succeeds). The return value should be 0 if the point described by the *x*- and *y*-coordinates is in the Mandelbrot set (i.e., reaches the threshold). You should test for divergence before you test for reaching the threshold.

   Example:

   ```
   $ echo "100 0.5 0.5" > task2.args
   $ scam -r task2.scm task2.args
   ((mandelbrot 100) 0.5000000000 0.5000000000) is 5
   $
   ```

3. Define a function named *root-n* which creates a function for calculating the n^{th} root of a given argument. Note that for a number x and a guess y, a better guess for the second root of x is $\frac{y+\frac{x}{y}}{2}$ and a better guess for the third root of x is $\frac{2\times y+\frac{x}{y^2}}{3}$. Extrapolate this pattern to figure out how to define and return a function that calculates the n^{th} root. The form of your returned function should follow that in the text for square root.

   Test for convergence by comparing consecutive guesses to see if they are *close enough*; do not compare with strict equality or against an absolute difference. Report the result to 15 decimal places.

   Example:

   ```
   $ echo "2 144" > task3.args
   $ scam -r task3.scm task3.args
   ((root-n 2) 144) is 12.000000000000000
   $
   ```

4. Define a function, named *crazyTriangle*, that constructs a function that will print out *n* levels of Pascal's triangle, but with a twist. The leftmost and rightmost numbers at each level are not necessarily ones, as with Pascal's triangle, but are given as the first and second arguments of *crazyTriangle*. The returned function takes a single argument, which is the number of levels in the triangle to be printed. The output produced by `((crazyTriangle 1 1) 6)` would be six levels of Pascal's triangle:

```
     1
    1 1
   1 2 1
  1 3 3 1
 1 4 6 4 1
1 5 10 10 5 1
```

The output produced by `((crazyTriangle 1 2) 6)` would be:

```
     1
    1 2
   1 3 2
  1 4 5 2
 1 5 9 7 2
1 6 14 16 9 2
```

Note that the apex is always the first argument.

Your triangle printing function must print one level to a line with lower levels above upper levels. The widest level must have no preceeding spaces; all other levels can only have spaces preceeding the first value in the level. All levels must have only a newline following the last value in the level. Finally, your levels need to be centered around the apex (but don't worry if the triangle skews rightward with multi-digit entries).

Your function must implement a tree-recursive process and should not overflow an integer *while computing a triangle entry* (unless the final value itself overflows).

Example:

```
$ echo "1 1 5" > task4.args
$ scam -r task4.scm task4.args
((crazyTriangle 1 1) 5)
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
$
```

5. Currying is the process of providing the arguments to a function at different points in time. The result of currying a function is a new function that accepts the first of the remaining, unspecified arguments. Define a function, named *curry*, that curries a four-argument function. As an example, the last two expressions should evaluate to the same result:

```
scam> (define (f a b c d) (+ a b c d))
<function f(a b c d)>
scam> (f 1 2 3 4)
10
scam> (((((curry f) 1) 2) 3) 4))
10
```

Example:

```
$ echo "(define (f a b c d) (+ a b c d))" > task5.args
$ echo "1 2 3 4" >> task5.args
$ scam -r task5.scm task5.args
(((((curry <function f(a b c d)>) 1) 2) 3) 4) is 10
$
```

Your main will have to do something a little bit different for the first argument. Instead of:

```
(define arg1 (readExpr))
```

it will need to evaluate the expression to turn it into a functioning function:

```
(define arg1 (eval (readExpre) this))
```

6. The function w, described below, implements *Shank's* transform:

$w(f,i) = f(i)$	if i is zero
$w(f,i) = \frac{S(f,i+1) \times S(f,i-1) - S(f,i)^2}{S(f,i+1) - 2 \times S(f,i) + S(f,i-1)}$	otherwise

where the function S implements summation:

$$S(f,n) = \sum_{i=0}^{n} f(i)$$

Implement w and S using iterative processes with *no* redundant computations. Report results to 15 decimal places.
Example:

```
$ echo "(lambda (x) (/ (if (= (% x 2) 0) 1.0 -1.0) (+ x 1)))" > task6.args
$ echo "0" >> task6.args
$ scam -r task6.scm task6.args
(S <function anonymous(x)> 0) is 1.000000000000000
(w <function anonymous(x)> 0) is 1.000000000000000
$
```

Note: the division in the transform is real number division.

7. The ancient Ethiopians were able to multiply numbers together without the use of a times table. Curiously, certain numbers were considered ‘unlucky’, so whenever those numbers came up in their calculations, that part of the calculation was thrown out. Yet, their method worked flawlessly for any two positive numbers. The method is easy to learn: Start a two-column table with the multiplicand heading the left column and the multiplier heading the right. For example, to multiply 1960 by 56, we generate a table, initialized with the following:

a	*b*
1960	56

At this point, we add new rows to the table by doubling the left-hand value and halving the right-hand value of the previous row. We do this until we get to a one in the right-hand column:

a	*b*
1960	56
3920	28
7840	14
15680	7
31360	3
62720	1

Note the halving throws away the fractional bits. Even numbers in the right-hand column are considered very unlucky, so we remove them:

a	*b*
15680	7
31360	3
62720	1

Finally, we add up the remaining numbers in the left-hand column. Thus, an ancient Ethiopian would tell you that 1960×56 is 109760, a value you can verify on your new-fangled computers.

Define a function named *ethiop*. Your method should implement the above algorithm using an iterative process. You may only use addition and subtraction but neither multiplication nor division in your solution. Define three functions named *double*, *halve*, and *div2?* which do their calculations using just addition and/or subtraction. The *double*, *halve*, and *div2?* functions must run in sub-linear time. The *halve* function discards any remainder.

Example:

```
$ echo "1960 56" > task7.args
$ scam -r task7.scm task7.args
(halve 1960) is 980
(double 1960) is 3920
(div2? 1960) is #t
(ethiop 1960 56) is 109760
$
```

8. The transcendental number, *e*, can be represented as the continued fraction:

   ```
   e = [2; 1,2,1, 1,4,1, 1,6,1, 1,8,1, 1,10,1, ...]
   ```

   Note the embedded series 2,4,6,8,10... In this notation, 2 is the augend and the remaining numbers represent the continued fraction addend. The numbers specify the denominators in the continued fraction (the numerators are all assumed to be one). For example, the list:

   ```
   [2; 1,2,1]
   ```

   is represented in fraction form as:

   $$2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1}}}$$

   Define a function, named *ecfi*, that, given the number of terms returns an approximation of *e* based upon the above continued fraction. For example, `(ecfi 0)` should return 2 and `(ecfi 1)` should return the value of:

   ```
   [2; 1,2,1]
   ```

   or 2.75, while `(ecfi 2)` should return the value of:

   ```
   [2; 1,2,1, 1,4,1]
   ```

   or 2.717948717948718. A *term*, in this case, has three denominators: the even number and the ones flanking it.

   The *ecfi* function should implement an iterative process. Place your *ecfi* function (with *main*) in the file *task8i.scm* Also, define a function *ecfr* that computes the same values but uses a recursive process. Place this function (with *main*) in *task8r.scm*. Your *main* functions should report the answers with 25 decimal places.

   Example:

   ```
   $ echo "0" > task8.args
   $ scam -r task8i.scm task8.args
   (ecfi 0) is 2.0000000000000000000000000
   $ scam -r task8r.scm task8.args
   (ecfr 0) is 2.0000000000000000000000000
   $
   ```

9. The famous Indian mathematician, Ramanujan, asked a question that stumped a number of people. What is the value of:

$$1 \cdot \sqrt{6 + 2 \cdot \sqrt{7 + 3 \cdot \sqrt{8 + 4 \cdot \sqrt{9 + 5 \cdot \sqrt{10 + ...}}}}}$$

   carried out to infinity?

   Define a function, named *ramanujanr*, which takes, as its single argument, the depth of a rational approximation to the above nested expression. For example, if the depth is 0, *ramanujanr* should return one times the square root of 6. If the depth is 1, *ramanujanr* should return the value of $1 \cdot \sqrt{6 + 2 \cdot \sqrt{7}}$ If the depth is 2, the return value should be the value of $1 \cdot \sqrt{6 + 2 \cdot \sqrt{7 + 3 \cdot \sqrt{8}}}$ Your function should implement a recursive process. Place your *ramanuganr* function (with *main*) in the file *task9r.scm* Also, define a function *ramanugani* that computes the same values but uses an iterative process. Place this function (with *main*) in *task9i.scm.* Your *main* functions should report the answers with 25 decimal places.

   Finally, your main functions should give the value of the nested square-root expression when carried out to infinity, using a LaTeX math expression.

   Example:

```
$ echo "0" > task9.args
$ scam -r task9r.scm task9.args
(ramanujanr 0) is 2.4494897427831778813356323)
?
$ scam -r task9i.scm task9.args
(ramanujani 0) is 2.4494897427831778813356323)
?
$
```

   The question mark in the last line of output, of course, should be replaced with the LaTeX math equation that represents the answer. For example, if the answer is π, then the question mark would be replaced by `$\pi$`. If the answer is $\log \pi$, then the question mark would be replaced by `$\log\pi$`. 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 are mandatory. Do not share this answer in any way.

Compliance

Output format has to match exactly, spacing and all. There can be no whitespace other than a newline after the last printable character of each line in any output. No lines of output are indented, unless explicitly specified.

Handing in the tasks

To submit assignments, you need to install the *submit system*:

- *Linux or Windows Bash instructions*
- *Mac instructions*

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

```
submit proglan lusth test1
```

For your final submission, use the command:

```
submit proglan lusth assign1
```

The *submit* program will bundle up all the files in your current directory and ship them to me. Thus it is very important that only the files related to the assignment are in your directory (you may submit test cases and test scripts). This includes subdirectories as well since all the files in any subdirectories will also be shipped to me, so be careful. You may submit as many times as you want before the deadline; new submissions replace old submissions.

Change log

- Version 2c - Sun Mar 3 10:21:42 - fixed example output for task 9
- Version 2b - Thu Feb 28 09:29:21 - fixed example output for task 2
- Version 2a - Thu Feb 28 07:36:36 - fixed example output for task 1 (again!)
- Version 2 - Wed Feb 27 09:27:52 - correct example output for task 1
- Version 1a - Thu Feb 21 09:48:57 - corrected number of decimal places in examples 8 and 9
- Version 1 - Fri Jan 11 09:50:20 - updated number conversion in task 1