CS403: Programming Languages

Assignment 1

Version 2a

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 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))
    (println (apply fact (readExpr)))
    )

(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.text
$ scam -r task5.scm task5.text
120
$
```

The filename *task5.text* is the first command-line argument. The `-r` option informs Scam to run the *main* function after the program has been loaded.

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.

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. Do not use the comment-out-the rest-of-the-file comment in your code.

Tasks

1. Define a function, named *zeno*, that computes the price of a ticket for Zeno's Airline. The function is passed the distance d in stadion and the number of hops n. A passenger is charged $c^{\log_2(h)}$ hemibool for each hop where h is the length of the hop. Note that the hop length is halved for each subsequent hop. For example, if the overall distance is 100 stadion, then the first hop is 50 stadion, the second hop is 25 stadion, the third hop is 12.5 stadion, and so on. When there is one hop left, the entire remaining distance is used to compute the fare.

   The other parameter passed to *zeno* is the hop charge factor c.

   Your *zeno* function should implement a recursive process. Expect real or integer numbers as arguments. Return the cost (a real number) in drachma.

   Your *main* function should look like this:

   ```
   (define (main)
       (setPort (open (getElement ScamArgs 1) 'read))
       (println (apply zeno (readExpr)))
       )
   ```

   Example:

   ```
   $ echo "(1 1 1)" > task1.args
   $ scam -r task1.scm task1.args
   0.0833333333
   $
   ```

   Constraints: division should always be real number division.

2. Define a function, named *minMaxSum*, that returns the sum of the minimum value and the maximum value of three unique arguments. Example:

```
$ echo "(1 2 3)" > task2.args
$ scam -r task2.scm task2.args
4
$
```

Constraints: Your implementation should use the minimum number of comparisons, semantically speaking. You are only allowed to call the `<` function to do your comparisons and you must only pass two arguments. You may not use local defines.

3. The Mandelbrot set (for examples, see `http://www.atopon.org/mandel/` 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.

Define a function named *resistance*, that when given an x, a y, and a threshold t, returns the iteration count, returning 0 if the divergence test is immediately true and returning t if the iteration count reaches t.

Points in the mandelbrot set are often colored black. The points *not* in the set can be categorized as to their resistance to divergence. When visualizing the Mandlebrot set, points are often colorized. Supposed one colored a point black if it is in the set, blue if it is very resistant to divergence, red if it immediately diverges, and somewhere in between red and blue for intermediate resistance.

Consider colorizing a point not in the mandelbrot set so that the resistance to divergence corresponds to the spectrum (red, orange, yellow, green, and blue - indigo and violet can only be approximated on a traditional computer screen), with red values indicating low resistance and blue values indicating high resistance. Define a set of functions named *mred*, *mgreen*, and *mblue*, which, when given a iteration number i and a threshold t, return the corresponding red, green, and blue value, respectively, For red, the *mred* function should model a cosine wave that spans a quarter cycle from to the given threshold The call `(mred i t)` should return 255 when i is 0, 0 when i is t-1, and 180 when i is $\frac{t-1}{2}$. The *mblue* function should model a sine wave that spans a quarter cycle, as well. The call `(mblue i t)` should return 0 when i is 0, 255 when i is t-1, and 180 when i is $\frac{t-1}{2}$. The *mgreen* function should model a sine wave that spans a half cycle. The call `(mgreen i t)` should return 0 when i is 0, 0 when i is t-1, and 255 when i is $\frac{t-1}{2}$. The three color functions should return zero when i is equal to t. In calculating color values, convert to an integer at the last moment by rounding.

Example:

```
$ echo "(0.0 0.0 100)" > task3.args
$ scam -r task3.scm task3.args
100 0 0 0
$
```

Constraints: You are only allowed the following top-level functions: *main*, *resistance*, *mred*, *mgreen*, and *mblue*. They must be defined in the order given. The only local definitions allowed are functions definitions. You are only allowed to call each of these functions once. Use a value of 3.14159265358979323846 for π.

4. Everybody is familiar with square roots and their utility (Pythagorean theorem, anyone?) but 12^{th} roots also have their utility. In western music, the twelfth root of two is used for *equal temperament* tuning of instruments.

   Define a function named *root12* which calculates the twelfth root of the given argument. Note that for a number n and a guess y, a better guess for the second root (square root) is $\frac{y+\frac{n}{y}}{2}$ and a better guess for the third root is $\frac{2\times y+\frac{n}{y^2}}{3}$. Extrapolate this pattern to figure out how to compute the twelfth root. The form of your solution should follow that in the text for square root.

   Example:

   ```
   $ echo "(1)" > task4.args
   $ scam -r task4.scm task4.args
   1.0000000000
   $
   ```

   Constraints: Input is a positive number, either a real number or an integer. Test for convergence by comparing consecutive guesses to see if they are *close enough*; do not compare with strict equality. Your function *root12* should return a real number. Your *main* function should not do any special formatting when printing the result. You may only have two top level functions, *main* and *root12*.

5. Define a function, named *pt*, that prints out n levels (given as the sole argument) of Pascal's triangle so that the output looks like a centered pyramid. For example `(pt 1)` should print out:

   ```
   1
   ```

   while `(pt 3)` should print out:

   ```
     1
    1 1
   1 2 1
   ```

   Don't worry if the triangle gets skewed to the right when entries become greater than 9.

   Example:

   ```
   $ echo "(1)" > task4.args
   $ scam -r task4.scm task4.args
   1
   $
   ```

   Constraints: You are only allowed the following top-level functions: *main* and *pt*. The bottom row in your triangle should have no preceding spaces. Higher rows should be indented, as shown, with spaces. Your method for computing a single element in the triangle should implement a recursive process using the traditional recursive method. The *pt* function should return `nil` after printing the rows of the triangle.

6. Partial function application is the process of providing the first k arguments to a function that takes n arguments. The result is a new function that takes the remaining $n - k$ arguments. Define a function, named *pfa*, that takes two arguments, a function to be partially applied and an integer specifying k. To make the task easier, k will limited to the values 1, 2, or 3. The *pfa* function will return a function that takes the k arguments. When called with the k arguments , the function that *pfa* returns will return a function that takes the remaining $n - k$ functions. As an example, the last two expressions should evaluate to the same result:

```
(define (f x y z) (+ x (* y z)))
(f a b c)
(((pfa f 2) a b) c)
```

The function that takes the remaining arguments will need to be variadic. You will also need to use the *cons* function and the *apply* function (the same function as found in your main function). Here are some examples of a variadic function using *cons* and *apply*:

```
(define (plus0 @) (apply + @))
(define (plus1 a @) (apply + (cons a @)))
(define (plus2 a b @) (apply + (cons a (cons b @))))
(plus0 1 2 3 4 5)
(plus1 1 2 3 4 5)
(plus2 1 2 3 4 5)
(+ 1 2 3 4 5)
```

The last four expressions should evaluate to the same result.

Your *main* function will read four items: a function of three arguments, the argument list for *pfa*, the first set of arguments, and the second set of arguments:

```
(define (main)
    (setPort (open (getElement ScamArgs 1) 'read))
    (define f (readExpr))
    (define first (readExpr))
    (define second (readExpr))
    (define third (readExpr))
    (define f1 (apply pfa (cons (eval f this) first)))
    (define f2 (apply f1 second))
    (define f3 (apply f2 third))
    (inspect (length (get 'parameters f1)))
    (inspect (length (get 'parameters f2)))
    (println f3)
    )
```

Example:

```
$ echo "(define (f x y z) (+ x (* y z)))" > task6.args
$ echo "(2)" >> task6.args
$ echo "(1 2)" >> task6.args
$ echo "(3)" >> task6.args
$ scam -r task6.scm task6.args
(length (get (quote parameters) f1)) is 2
(length (get (quote parameters) f2)) is 1
7
$
```

Constraints: You are only allowed the following top-level functions: *main* and *pfa*.

7. Define a function, named *zarp*, that computes the function described by the following recurrence:

$zarp(i) = i$	if $i < 3$
$zarp(i) = zarp(i-1) + 2 \times zarp(i-2) - zarp(i-3)$	otherwise

The function *zarp* is only defined for non-negative integers.

Example:

```
$ echo "(0)" > task7.args
$ scam -r task7.scm task7.args
0
$
```

Constraints: You are only allowed the following top-level functions: *main* and *zarp*. Your *zarp* function must implement an iterative process.

8. The ancient Babylonians were quite sophisticated when it came to astronomy and mathematics, being able to predict lunar and solar eclipses. In fact, they were the ones to come up with dividing a day into 24 hours, an hour into 60 minutes, and a minute into 60 seconds. They also had an interesting method of multiplication based upon a table of squares. Since squaring is multiplication, it seems the Babylonians had a chicken-and-egg problem: to multiply, you need to square, but to square, you need to multiply. However, they managed to build their table of squares without resorting to multiplication. Instead, they used the following recurrence:

$square(i) = 1$	if $i = 1$
$square(i) = square(i-1) + i + i - 1$	otherwise

With a table of squares in hand, the multiplication of two numbers a and b would proceed via the following formula:

$$a \times b = \frac{((a+b)^2 - a^2 - b^2)}{2}$$

Thus, multiplication was performed via one addition, three table look-ups for squares, two subtractions, and a halving (again via a table look-up).

Define a method named *babyl* that performs the aforementioned method of multiplying two *positive* numbers. Your implementation should provide a function for squaring based upon the given recurrence as well as a function for halving based upon an analogous recurrence. Name these functions *halve* and *square*.

Your *main* function should look like this:

```
(define (main)
    (setPort (open (getElement ScamArgs 1) 'read))
    (define args (readExpr))
    (println "half of " (car args) " is " (halve (car args)))
    (println "half of " (cadr args) " is " (halve (cadr args)))
    (println (car args) " squared is " (square (car args)))
    (println (cadr args) " squared is " (square (cadr args)))
    (println (apply babyl args))
    )
```

Example:

```
$ echo "(21 42)" > task8.args
$ scam -r task8.scm task8.args
half of 21 is 10
half of 42 is 21
21 squared is 441
42 squared is 1764
882
```

Constraints: You are only allowed the following top-level functions: *main*, *babyl*, *square*, and *halve*. The only built-in mathematical operations allowed are addition and subtraction. The functions *babyl*, *square* and *halve* should implement iterative processes. The *halve* function must run in sub-linear time.

9. Many transcendental numbers can be represented as a continued fraction:

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

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; 1,2,3]
```

is represented in fraction form as:

$$2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{3}}}$$

Consider this infinite equation:

```
[1; 1,1,1, 5,1,1, 9,1,1, 13,1,1, ...]
```

Define a function called *mystery* that when given an integer argument n, computes the value of this equation to n terms (where each term is a group of three digits in the series above).

For example, if n is 0, the function should return 1. For n equal to 1, it should return the value of the fraction [1; 1,1,1]. For n equal to 2, it should return value of [1; 1,1,1, 5,1,1].

Your function should compute its value using a recursive process. Define a second function named *imystery* with the same semantics but implementing an iterative process.

Your program should give the result of *mystery*, *imystery*, and the symbolic value of the equation with an infinite number of terms. Hint, square the mystery results.

Example:

```
$ echo "(0)" > task9.text
$ scam -r task9.scm task9.text
mystery returns 1
imystery returns 1
?
$
```

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 was 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. Do not share this answer in any way.

Constraints: You are only allowed the following top-level functions: *main*, *mystery*, and *imystery*. The *mystery* function should implement a recursive process, while the *imystery* function should implement an iterative process.

10. The famous Indian mathematician, Ramanujan, asked a question that no one else seemed to be able to answer: what is the value of:

$$\sqrt{1+2\cdot\sqrt{1+3\cdot\sqrt{1+4\cdot\sqrt{1+5\cdot\sqrt{1+...}}}}}$$

carried out to infinity? This was before computers; with them you don't have to be a mathematical genius to compute a good guess at the answer. Define a function, named *ramanujan*, which takes as its sole argument the depth of a rational approximation to the above nested expression. For example, if the depth is 0, *ramanujan* should return 0. If the depth is 1, *ramanujan* should return the value of $\sqrt{1+2}$. If the depth is 2, the return value is the value of $\sqrt{1+2\cdot\sqrt{1+3}}$, and so on. Your function should implement a recursive process. Define a second function, named *iramanujan*, with the same semantics but implementing an iterative process.

Your program should give the result of *ramanujan*, *iramanujan*, and the answer to Ramanujan's question.

Example:

```
$ echo "(0)" > task10.text
$ scam -r task10.scm task10.text ramanujan
ramanujan returns 0.000000e+00
iramanujan returns 0.000000e+00
?
$
```

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 was 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. Do not share this answer in any way.

Constraints: You are only allowed the following top-level functions: *main*, *ramanujan*, and *iramanujan*. The *ramanujan* function should implement a recursive process, while the *iramanujan* function should implement an iterative process.

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.