CS403: Programming Languages

Assignment 1

Preliminary information

This is your first Scam assignment. To run your code, use the following command:

```
scam assign1.scm
```

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

```
(define (author)
    (println "AUTHOR: Rita Recursion rrita@crimson.ua.edu")
    )
```

with the name and email replaced by your own name and email. Define this function as well, to help with testing your code:

```
(define (exprTest # $expr target)
    (define result (catch (eval $expr #)))
    (if (error? result)
        (println $expr " is EXCEPTION: " (result'value)
            " (it should be " target ")")
        (println $expr " is " result
            " (it should be " target ")")
        )
    )
```

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, starting at one (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)
    (exprTest (fact 0) 1)
    (exprTest (fact 3) 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).

There is 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. If your program passes the provided test script, it will be graded with a more thorough set of tests. You can get the test script with the command:

```
wget beastie.cs.ua.edu/proglan/test1.scm
```

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
```

Another useful function for your *run* function is *inspect*. Here is an example usage:

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

which produces the output:

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

You may not use assignment (*assign* or *set!*) in any of the code you write. Nor may you use any looping function such as *while* or *for*. You may not use list or arrays, unless otherwise specified.

Tasks

1. Explain how and why *and* and *my-and* (defined below) can behave differently when given two identical boolean expressions. Here is an example of equivalent calls to *and* and *my-and* that behave exactly the same, in terms of output:

   ```
   (define x 10)
   (define a (readInt))
   (inspect (and (< a x) (= (% a 2) 1)))
   (inspect (my-and (< a 10) (= (% a 2) 1)))
   ```

   where *my-and* is defined as

   ```
   (define (my-and a b)
       (if (true? a)
           b
           #f
           )
       )
   ```

   Your *run* function should display a concrete example in which the behavior is different. You will need to come up with specific cases that show this difference.

   Note: your *run* function should not execute your example; it should simply explain why the behavior is different for your example.

2. Define a function named *min5* that returns the minimum of its five numeric arguments. Full credit is reserved for those implementations that use a minimum of darkspace.

   Do not redefine any builtin functions. Any helper functions are to be nested.

   In addition to placing your implementation of your program in *assign1.scm*, place a copy of your function (and only your function) in a file named *min5*. We will use your *min5* file to determine the amount of darkspace.

3. Consider colorizing a value between 0 and 100 so that each integer value corresponds to an CYM color. To determine the CYM color, the cyan, yellow, and magenta colors are determined individually. To compute the *cyan* color, 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.

   Use a value of 3.14159265358979323846 for π. You may use a more accurate value if you want.

   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. In my solution, I also used the *array* and *getElement* functions, but these are not strictly necessary.

   Note: all computed color values should be truncated. For example, if the actual value computed by a color function is 138.87645673, the function should report 138.

4. Define a function named *root5* which calculates the fifth root of the given argument. Note that for a number n and a guess y, a better guess for the second root of n is $\frac{y+\frac{n}{y}}{2}$ and a better guess for the third root of n is $\frac{2\times y+\frac{n}{y^2}}{3}$. Extrapolate this pattern to figure out how to compute the fifth root. The form of your solution should follow that in the text for square root.

5. Define a function, named *bico*, that returns , from the i^{th} expansion of the quantity $(x+y)$, the j^{th} coefficient. For example *(bico 4 2)* should return 6 since $(x+y)^4$ is equal to $x^4+4x^3y+6x^2y^2+4xy^3+y^4$. Note that j ranges from 0 to i.

   Your function should be recursive and implement a recursive process.

6. 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:

   ```
   (define (f a b c d) (+ a b c d))
   (f a b c d)
   ((((curry f a) b) c) d)
   ```

7. Define a function, named *zorp*, that computes (via an iterative process) the function described by the following recurrence:

$zorp(i,f) = f(i)$	if $i < 3$
$zorp(i,f) = zorp(i-1,f) + \frac{(zorp(i-1,f)-zorp(i-2,f))^2}{zorp(i-3,f)-2\times zorp(i-2,f)+zorp(i-1,f)}$	otherwise

   The functions *zorp* is only defined for non-negative integers. The first argument passed to *zorp* is i, the second is the function *f*. Note that not all functions are compatible with *zorp*; some will cause *zorp* to fail with a divide-by-zero error. Use integer division in the recursive call.

   Do not make *any* recursive calls that are not iterative.

8. The ancient Egyptians were perhaps the first people on earth to come up with the idea of binary arithmetic when they developed their method of division. The Egyptian Division method is a tabular calculation that lends itself to a straightforward computer implementation. The table starts out with a 1 in column a, the divisor in column b and the dividend in column c. Columns a and b are successively doubled until the value in column b is greater than the value in column c. For example, to divide 1960 by 56, we generate the following table:

a	b	c
1	56	1960
2	112	1960
4	224	1960
8	448	1960
16	896	1960
32	1792	1960
64	3584	1960

At this point, we add a fourth column initialized to zero and apply the following algorithm. If the number in column b is less than or equal to that of column c, we add column a to column d and subtract column b from column c. Otherwise, we leave the values in c and d unchanged. In either case, we halve (integer division) the values in both columns a and b. We stop when column a becomes less than 1. At this point, the answer resides in column d.

a	b	c	d
64	3584	1960	0
32	1792	1960	0
16	896	168	32
8	448	168	32
4	224	168	32
2	112	168	32
1	56	56	34
0	28	0	35

Define a function named *egypt/* that takes two arguments, the divisor and the dividend and returns the quotient. Example call:

```
(egypt/ 1960 56) ;divide 1960 by 56
```

Your method should implement an iterative process. You may use neither multiplication nor division in your solution. Hint: define two functions named *double* and *halve* which do their calculations using just addition and/or subtraction. The *halve* function must run in sub-linear time. Both *double* and *halve* should be visible, not nested.

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:

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

Implement a function, named *mystery*, that implements an iterative process. Your function should take the number of terms plus the augend and two functions for generating the numerator and the denominator of the continued fraction. For example, this call:

```
    (mystery 3 2 (lambda (n) 1) (lambda (n) n))
```

should produce the value of this truncated infinite fraction:

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

Your *run* function should specify the mysterious number *z*, generated when taking the continued fraction out to infinity.

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

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

carried out to infinity?

Define a function, named *ramanujan*, which takes, as its single argument, the depth of a rational approximation to the above nested expression. For example, if the depth is 0, *ramanujan* should return the square root of 6. If the depth is 1, *ramanujan* should return the value of $\sqrt{6+2\cdot\sqrt{7}}$ If the depth is 2, the return value should be the value of $\sqrt{6+2\cdot\sqrt{7+3\cdot\sqrt{8}}}$ Your function should implement a recursive process. Define a second function, named *iramanujan*, with the same semantics but implementing an iterative process.

Your *run* function should give the value of the above expression when carried out to infinity.

Assignment submission

The entire assignment should be contained in a single file named *assign1.scm*. Any explanatory text should be in the form of Scam comments, which begin with a semicolon. The file should load into the Scam interpreter cleanly. The last line of your file should be:

```
    (println "assignment 1 loaded!")
```

If you do not see the message ”assignment 1 loaded” when executing your file with the Scam interpreter, then there is an error somewhere that needs to be fixed. If your file does not load properly (i.e. I do not see the message), you will receive no credit for your work.

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

- *linux and cygwin instructions*
- *mac instructions*

Now delete extraneous files from your working directory. Finally, while in your working directory, type 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.