Preliminary information

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

```sh
scam assign1.scm
```

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

```scm
(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:

```scm
(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:

```scm
(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).

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. If your program passes the provided test script, it will be graded with a more thorough set of tests.

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

```scm
(println "The quick brown fox
    m
    u
    j
        e
d
    over the lazy dog")
```
which will print out as:

The quick brown fox

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.

Tasks

1. Explain why if and my-if (defined below) can behave differently for equivalent, legal inputs. Here is an example of equivalent calls to if and my-if that behave exactly the same, in terms of output:

   (define x 2)
   (define a (readInt))
   (inspect (if (= a 0) x (/ a x)))
   (inspect (my-if (= a 0) x (/ a x)))

   where my-if is defined as

   (define (my-if a b c)
      (if (true? a)
          b
          c)
   )

   In particular, give a concrete example in which the behavior is different. You will need to come up with specific cases that show this difference.

   Your run function should present this example and should explain precisely why the behavior is different.

2. Define a function, named zeno_cost, that computes the price of a ticket for Zeno’s Airlines given the distance d of the flight in stadia, the cost c of the first half of the trip in drachma, and a factor f for computing the cost of the rest of the trip. The total cost of a ticket is computed as follows: c drachma for the first half of the trip and \( c \times f \) drachma for the first half of the remaining half. Of the part of the trip that still remains, the first half of that is \( c \times f \times f \) drachma. Indeed, the first half of the remaining portion is always \( f \) times the cost of the previous portion.

   If the distance to be traveled is less than or equal to a daktylos (there are 9,600 daktylos to 1 stadion), then the cost of traveling that distance a fixed cost of 7 drachma. Otherwise, if the cost of traveling half the distance to be traveled is less than or equal to a hemiobol (one-twelfth of a drachma), the cost of traveling the two halves is one hemibool.

   Your zeno_cost function should implement a recursive process. Expect real or integer numbers as arguments. Return the cost (a real number) in drachma.
3. The Mandelbrot set (for examples, see [link](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:

\[
\begin{align*}
r &= r^2 - s^2 + x \\
\text{and} \quad s &= 2 \times r \times s + y
\end{align*}
\]

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, a point colored black if it is in the set, red if it is very resistant to divergence, blue if it immediately diverges, and somewhere in between red and blue for intermediate resistance.

Define a function, named `mandelbrot-iter`, 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 usage:

```scheme
(define mandelbrot-tester (mandelbrot-iter 100))
(if (= (mandelbrot-tester 2 3) 0)
  (print "point (2,3) is in the Mandelbrot set!\n")
  (print "point (2,3) is not in the Mandelbrot set.\n")
)
```

In the above example, the threshold for determining whether or not a number is in the Mandelbrot set is 100.

4. Define a function named `root3` which uses a binary search algorithm to find the cube root a given number. Your function should return the current approximation when the current approximation is indistinguishable from the previous approximation. Your function need only work for non-negative numbers.

5. Define a function, named `crazyTriangle`, that prints 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. The third argument is the number of levels to be printed. The output produced by `(crazyTriangle 1 1 6)` would be 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 function must print one level to a line with lower levels above upper levels. Your levels need to be centered around the apex (but don’t worry if the triangle skews rightward with multi-digit numbers). Your function must also minimize any redundant computations and should not overflow an integer while computing a triangle entry (unless the entry itself overflows).
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 last of the remaining, unspecified arguments. Define a function, named oppy, that curries a mathematical expression of two binary operators. As an example, these two expressions should evaluate to the same result:

\[
(+ \ x \ (\ast \ y \ z))
\]
\[
(((\text{oppy} \ (+)) \ x \ (\ast)) \ y) \ z)
\]

Note that \(y\) and \(z\) could be instantiated far later than \(x\). Your implementation will only be tested on expressions of the form given above.

7. The function \(w\), described below, implements Shank’s transform:

\[
\begin{align*}
\text{if } i \text{ is zero} & \quad w(f, i) = f(i) \\
\text{otherwise} & \quad w(f, i) = \frac{S(f, 2i) \times S(f, i-1) - S(f, i)^2}{2 \times S(f, i) + S(f, i-1)}
\end{align*}
\]

where the function \(S\) implements summation:

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

Implement \(w\) and \(S\) using an iterative process with no redundant computations.

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 multiplication. The Egyptian Multiplication method is a tabular calculation that lends itself to a straightforward computer implementation. The table starts out with a 1 in column \(a\), the multiplicand in column \(b\) and the multiplier in column \(c\). Columns \(a\) and \(c\) are successively doubled until the value in column \(a\) is greater than the value in column \(b\). For example, to multiply 1960 by 56, we generate the following table:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>56</td>
<td>1960</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>56</td>
<td>3920</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>56</td>
<td>7840</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>56</td>
<td>15680</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>56</td>
<td>31360</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>56</td>
<td>62720</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>56</td>
<td>125440</td>
<td></td>
</tr>
</tbody>
</table>

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

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>56</td>
<td>125440</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>56</td>
<td>62720</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>24</td>
<td>31360</td>
<td>62720</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>15680</td>
<td>94080</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>7840</td>
<td>109760</td>
</tr>
</tbody>
</table>
```

Define a function named \(egypt^*\) that takes two arguments, the multiplicand and the multiplier and returns the product. Example call:

\[
(egypt^* \ 56 \ 1960) ;\text{multiply 56 by 1960 (with no multiplication)}
\]
\[
(halve 56) ;\text{divide 56 by 2 (with no division)}
\]

Your method should implement an iterative process for both \(egypt^*\) and \(halve\). You may not use either multiplication or division in your solution. The \(halve\) function must run in sub-linear time.
9. Consider this infinite fraction:

\[
1 + \frac{1}{1 + \frac{1}{2 + \frac{1}{1 + \frac{1}{2 + \cdots}}}}
\]

Define a function called \textit{mystery} that when given an integer argument \(n\), computes the value of this equation to \(n\) terms.

For example, if \(n\) is 0, the function should return 1. For \(n\) equal to 1, it should return \(1 + \frac{1}{1}\) or 2. For \(n\) equal to 2, it should return \(1 + \frac{1}{1 + \frac{1}{2}}\) or \(\frac{5}{3}\). The return value should be cast to a real number. Your function should compute its value using a recursive process.

Your \textit{run} function should give the value of the equation with an infinite number of terms.

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

\[
\sqrt{1 + 2 \cdot \sqrt{1 + 3 \cdot \sqrt{1 + 4 \cdot \sqrt{1 + 5 \cdot \sqrt{1 + \cdots}}}}}
\]

carried out to infinity? Instead of answering this question, Ramanujan, gave a solution to the more general problem, the value of:

\[
\sqrt{1 + x \cdot \sqrt{1 + (x + 1) \cdot \sqrt{1 + (x + 2) \cdot \sqrt{1 + (x + 3) \cdot \sqrt{1 + \cdots}}}}}
\]

carried out to infinity. Define a function, named \textit{ramanujan}, which takes as its two arguments the depth of a rational approximation to the above nested expression (as before) and the value of \(x\). For example, if the depth is 0 and \(x\) is 3, \textit{ramanujan} should return 0. If the depth is 1 and \(x\) is 3, \textit{ramanujan} should return the value of \(\sqrt{1 + 3}\). If the depth is 2 and \(x\) is 3, the return value should be the value of \(\sqrt{1 + 3 \cdot \sqrt{1 + 4}}\). Your function should implement a recursive process. Define a second function, named \textit{iramanujan}, with the same semantics but implementing an iterative process.

Your \textit{run} function should give the value of the above expression in terms of \(x\).

\textbf{Assignment submission}

The entire assignment should be contained in a single file named \texttt{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:

\begin{verbatim}
(println "assignment 1 loaded!"
\end{verbatim}

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 \textit{submit} system:

- \texttt{linux and cygwin instructions}
- \texttt{mac instructions}

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

```
submit proglan lusth assign1
```

The \textit{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.