Preliminary information

This is your second Scam assignment. 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.

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 (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).

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.

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.

Tasks

1. Define a function named loop that takes a procedure and a list containing a lower bound (inclusive) and an upper bound (exclusive). The loop function should repeatedly execute the procedure, supplying a single argument from the list of integers derived from the given bounds.

For example, the call:

```
(loop (lambda (x) (inspect x)) '(0 5))
```

should produce the following output:

- x is 0
- x is 1
- x is 2
- x is 3
- x is 4

2. Currying is the process of providing one or more of the arguments to a function. The result of currying is a new function that accepts some or all of the remaining, unspecified arguments. Define a function, named curry, that curries the given function. As an example, the following pairs of expressions should evaluate to the same result:
(f a b c)
((curry f a) b c)

(g v w x y z)
(((curry g v w ) x) y z)

Note that curry is variadic and that the syntax of variadic functions in Scam is different than that of Scheme.

An easy way to implement curry is to use the apply function, which, when given a function and a list of arguments, calls the given function with the given arguments. The two calls below are equivalent:

(+ 1 2 3 4)
(apply + (list 1 2 3 4))

You can determine the number of arguments a function expects by asking for the length of its formal parameter list:

(length (get 'parameters f))

Your curry function will only be tested with functions taking a fixed number of arguments.

3. Define a function named infix->prefix that takes a quoted arithmetic infix expression involving numbers, variables, and operators and transforms the expression into a prefix expression. The operators are +, -, *, /, and ^ where ^ represents the exponentiation operator. The precedence of the operators increases in the order given. Thus + has the lowest precedence while ^ has the highest precedence. As an example,

(infix->prefix '(2 + 3 * x ^ 5 + a))

would return the list

(+ 2 (+ (* 3 (\^ x 5)) a))

or

(+ (* 2 (* 3 (\^ x 5))) a)

or something mathematically equivalent. Implement all operators as left-associative, except exponentiation, which should be right associative. You do not need to handle parenthesized expressions in the infix form.

You must implement the classic solution to this problem, an approach which utilizes a stack and a queue (or two stacks). Your queue, if you need one, is allowed to use linear time for enqueuing.

4. Define a function named no-locals which takes a source code list representing a function definition and returns a source code list representing an equivalent definition with local defines replaced by a lambda. For example,

(no-locals
 (quote
   (define (nsq a)
     (define x (+ a 1))
     (define y (- a 1))
     (* x y)
   )
 )

should evaluate to the list:

(define (nsq a)
  ((lambda (x y) (* x y))
   (+ a 1)
   (- a 1)
  )
)

You may assume that all local defines are at the beginning of the function body and that they do not refer to each other.
5. In the lambda calculus, functions are restricted to having a single argument. Define a function named `convert` that takes a lambda of possibly many arguments, as a source code list, and returns a source code list of a nested function that takes the arguments one at a time.

For example,

```
(convert (quote (lambda (a b) (+ a b))))
```

should evaluate to the list:

```
(lambda (a) (lambda (b) (+ a b)))
```

If one were to instantiate the source code, as in:

```
(define plus
  (eval
   (convert (quote (lambda (a b) (+ a b))))
   this)
)
```

then the expression:

```
((plus 3) 4)
```

should return 7.

Remember, there may be any number of formal parameters and any number of body expressions in the source code lambda.

6. Define a function, named `reverse*`, that reverses the top level of a list as well as any nested sublists, recursively. For example,

```
(reverse* '(1 (2 (3 (4 5))) 6))
```

should evaluate to:

```
(6 (((5 4) 3) 2) 1)
```

Your solution should implement iterative recursive loops.

To make this task easier, you only need to handle lists with numbers. Any nested lists will be right-associative (as shown in the example). You may assume any library function is iterative; they may not be, but they could be rewritten to be iterative.

7. Do exercise 2.37, defining `matrix-*vector`, `transpose`, and `matrix-*matrix`, along with any other support functions. Unlike the exercise, store the matrix in column-major format (as is done in Fortran). In column-major format, the example matrix would be represented as (1 4 6) (2 5 7) (3 6 8) (4 6 9).

You must follow the style of the book and you must minimize your use of the `transpose` operation.
8. Consider this node constructor and tree displayer:

```scheme
(define (node value left right)
  (define (display) (print value))
  this)

(define (displayTree root indent)
  (if (valid? root)
      (begin
        (displayTree (root'right) (string+ indent " "))
        (print indent)
        ((root'display))
        (println)
        (displayTree (root'left) (string+ indent " "))
      )
    ))
)
```

Define an `insertInTree` function that takes a binary search tree and a value and returns a new binary search tree that includes that value. Example:

```
(define t0 (node 5 nil nil))
(define t1 (insertInTree t0 2))
(define t2 (insertInTree t1 8))
(displayTree t2 " ")
```

The last command would display the tree sideways:

```
8
5
2
```

9. Define a series of functions, named `big+`, `big-`, and `big*`, to support the addition, subtraction, and multiplication of arbitrary sized integers, respectively. Each integer will be expressed as a list of digits. For example, the integer 4918039 would be represented as the list `'(4 9 1 8 0 3 9)`. Use Karatsuba's divide and conquer method for your multiplication function. Your functions need not be variadic; binary operation is sufficient.

Hint: it will make your life easier if you reverse your big integers while doing your computations. Consider this addition:

```
(big+ ''(4 5 8 9) ''(3 0 2))
```

The `big+` function would internally reverse the augend and addend to `'(9 8 5 4)` and `'(2 0 3)`, respectively, before doing the addition. After computing the sum `'(1 9 8 4)`, it would return the reverse of the result, yielding `'(4 8 9 1)`. Negative numbers should be represented by a leading minus sign. For example, the number -4918039 would be represented by the list `'(− 4 9 1 8 0 3 9)`. Note: the minus sign is the symbol generated by `(quote -)`, not the built-in subtraction function.
10. Integrate your big+, big-, and big* functions into Scam's normal arithmetic functions. Do this by saving the old functions, as in:

\[
\text{(define old+ +)}
\]

and then redefining new ones, as in:

\[
\text{(define (+ a b)
\quad \ldots)
\]

The new functions should call the old functions, if appropriate. Furthermore, if the operation processing two regular integers will cause an overflow or underflow, the regular integers should be first converted to big integers. Conversely, if a big integer result fits into a regular integer, the big integer should be converted to a regular integer. For ease of testing, assume an regular integer ranges from \(-2^{15}\) to \(2^{15} - 1\). Examples:

\[
\begin{align*}
(+ 234 5) & \quad ; \text{yields a regular int} \\
(+ '(2 3 4) 5) & \quad ; \text{yields a regular int} \\
(+ 234 '(5)) & \quad ; \text{yields a regular int} \\
(+ 32767 32767) & \quad ; \text{yields a big int} \\
(* 10000 4) & \quad ; \text{yields a big int} \\
(- '(4 0 0 0 0) '(1 0 0 0 0)) & \quad ; \text{yields a regular int}
\end{align*}
\]

Your functions cannot refer to any regular integers outside the assumed range.

Assignment submission

Submitting the assignment The entire assignment should be contained in a single file named assign2.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:

\[
\text{(println "assignment 2 loaded!")}
\]

If you do not see the message "assignment 2 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 your assignment, delete extraneous files from your working directory. Then, while in your working directory, type the command:

\[
\text{submit proglan luth assign2}
\]

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.