Preliminary information

This is your second 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:

```scheme
(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).

You may not use assignment in any of the code you write. Nor may you use any looping function such as while or for. Do not use the comment-out-the rest-of-the-file comment in your code.

Tasks

1. Define a variadic function named n-loop that takes a procedure and some number of lists, each containing a lower bound (inclusive) and an upper bound (exclusive). The loop function should repeatedly execute the procedure, supplying as many arguments as there are bounding lists with the arguments derived from the given bounds. NOTE: the syntax of variadic functions in Scam differs from that of Scheme.

For example, the call:

```
(n-loop (lambda (x y) (inspect (list x y))) (0 2) (0 3))
```

should produce the following output:

```
(list x y) is (0 0)
(list x y) is (0 1)
(list x y) is (0 2)
(list x y) is (1 0)
(list x y) is (1 1)
(list x y) is (1 2)
```

Example:

```
$ # (n-loop (lambda (x) (inspect x)) (0 1))
$ echo "((lambda (x) (inspect x))" > task1.args
$ echo "((0 1))" >> task1.args
$ scam -r task1.scm task1.args
x is 0
$ 
```
Define your main such that it evaluates the first expression read, as in pfa task in the previous assignment.

2. Partial function application is the process of breaking up the arguments to a function into two groups. When the first group of arguments and the function itself is passed to a partial-evaluator, a function that accepts the remaining arguments is returned. Define a variadic function, named pfa, that partially evaluates a given function and the first \( k \) arguments. As an example, the last five expressions in the following list should evaluate to the same result:

\[
\begin{align*}
  &((\text{define } (f \ x \ y \ z) (+ \ x \ y \ z)) \\
  &\quad (f \ a \ b \ c) \\
  &\quad ((\text{pfa } f) \ a \ b \ c) \\
  &\quad ((\text{pfa } f \ a) \ b \ c) \\
  &\quad ((\text{pfa } f \ a \ b) \ c) \\
  &\quad ((\text{pfa } f \ a \ b \ c))
\end{align*}
\]

If the wrong number of arguments are supplied, you should throw one of the following exceptions:

\[
\text{(throw 'MALFORMED_FUNCTION_CALL "too many arguments")}
\]
\[
\text{(throw 'MALFORMED_FUNCTION_CALL "too few arguments")}
\]

Example:

```
$ # ((pfa f 1) 1)
$ echo "((define (f a b) (+ a b))" > task2.args
$ echo "(1)" >> task2.args
$ echo "(1)" >> task2.args
$ scam -r task2.scm task2.args
2
```

Your main will need to evaluate the first expression.

3. Define a function named infix->postfix that takes a quoted arithmetic infix expression involving numbers, variables, and operators and transforms the expression into a postfix 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,

\[
\text{(infix->postfix '(2 + 3 * x ^ 5 + a))}
\]

would return the list:

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

Note that all operators are left associative.

In your main, do not apply infix->postfix to the read-in expression. Just call infix->postfix with the expression as an argument. Example:

```
$ # (infix->postfix '(2 + 3))
$ echo "((2 + 3))" > task3.args
$ scam -r task3.scm task3.args
(2 3 +)
```

2
4. Define two functions, `if2cond` and `cond2if` that convert source code in the first form to that of the second. For example,

```
(if2cond '(if (< a b) a b))
```

should return the list:

```
(cond ((< a b) a) (else b))
```

Both functions should work recursively. That is, both should handle nested `ifs` and nested `conds`, respectively. You may assume that all `ifs` have both a true expression and a false expression and that all `conds` have an else. You may also assume that each action of a `cond` clause and both clauses of an `if` are single expressions and are neither begin blocks and nor lambda expressions.

In your `main`, call `if2cond` with the first read-in expression and `cond2if` with the second expression, printing each result. Example:

```
# (if2cond '(if #t 0 1))
# (cond2if '(cond (#t 0) (else 1)))
$ echo "(if #t 0 1)" > task4.args
$ echo "(cond (#t 0) (else 1))" >> task4.args
$ scam -r task4.scm task4.args
(cond (#t 0) (else 1))
(if #t 0 1)
$ 
```
5. It turns out that a programming language need not have numbers as a core part of the language; they can be programmed in!

Suppose we define zero as a function rather than a number. Let zero be the function that, regardless of its single argument, returns the identity function. The identity function is:

```
(define (identity x) x)
```

Understand that we are not defining zero as the identity function but as a function that returns the identity function.

Next, suppose we define increment as a function that takes one of these funny numbers (like zero) and returns a function representing the next higher number. We can define increment as:

```
(define (increment number)
  (lambda (incrementer)
    (define (resolver base)
      (incrementer ((number incrementer) base))
    )
    resolver
  )
)
```

We can see that incrementing zero is equivalent to defining one as:

```
(lambda (incrementer)
  (define (resolver base)
    (incrementer base)
  )
  resolver
)
```

Next, define two functions named add and multiply that add and multiply two of these functional numbers, respectively. Your add routine should add numbers directly (i.e. without using increment and the like).

Example:

```
$ echo "(define zero (lambda (f) (lambda x) x))" > task5.args
$ echo "(define one (lambda (f) (lambda x) (f x)))" >> task5.args
$ echo "(define (inc x) (+ x 1))" >> task5.args
$ echo "(define base 0)" >> task5.args
$ scam -r task5.scm task5.args
1
0
$
```

Your main function should evaluate all four expressions and the print the following expressions;

```
(((add zero one) inc) base)
(((multiply zero one) inc) base)
```

Constraints: You are only allowed the following top-level functions: main, increment, add, and multiply. Your add function should not use the increment function. Your multiply function should not use the add function. You will likely need to do some research on Church numerals.
6. Define a function named `map+` that has the same functionality as the native map in Scheme (the version of `map` that can take one or more lists to map over). You may call `map` from `map+` but you may only send one list to `map` on any given invocation. You will need to make `map+` a variadic function. Note: Scam does not use the dotted tail notation of Scheme to implement variadic functions. See The Scam Reference Manual on implementing variadic functions. Hint: a list of lists is a single list that can be passed to `map`.

Example usage:

```
$ # (map+ + (1 2 3) (4 5 6) (7 8 9))
$ echo "+" > task6.args
$ echo "((1 2 3) (4 5 6) (7 8 9))" >> task6.args
   (12 15 18)
$  
```

Your `main` function should evaluate the first expression and should apply `map+` to the cons of the evaluated first expression and the second expression.

Constraints: You are only allowed the following top-level functions: `main` and `map+`. You may only call the built-in `map` with a single list.