CS403: Programming Languages

Assignment 2

Version 2a

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:

```
(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. On any line of output, there should be no leading whitespace and no trailing whitespace other than a newline (except when otherwise directed).

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:

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

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

```
(throw 'MALFORMED_FUNCTION_CALL "too many arguments")
(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,

```
(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 +)
$
```

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 *if*s and nested *cond*s, respectively You may assume that all *if*s have both a true expression and a false expression and that all *cond*s 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 num1 (lambda (f) (lambda (x) x)))" > task5.args
   $ echo "(define num2 (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 num1 num2) inc) base)
   (((multiply num1 num2) 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.

7. This exercise is similar to Exercise 2.42 in the text. Unlike the textbook version, the return value of the *queens* function should be a list of locations with the row numbers in decreasing order. Columns need to be filled in a greedy manner, For example, here is an example return value for an eight by eight board size with columns filled greedily from low to high:

   ```
   ((7 6) (6 7) (5 5) (4 3) (3 1) (2 4) (1 2) (0 0))
   ```

   Note that the row numbers and column numbers start with 0 (unlike the textbook version). Note also that the above solution is incorrect.

   Your *queens* function should take three arguments, the number of rows, the number of columns, and a flag that says columns should be filled greedily from low to high (if the flag is `#t`) or high to low (if the flag is `#f`). If there is no solution, *queens* should return the empty list.

8. Define a function, named *extract*, which when given a symbol composed of `'h`s and `'t`s, generates a list function that extracts the appropriate element(s). An `'h` implies taking the *car* while a `'t` implies taking the *cdr*. Actions should be performed from left to right. For example:

```
(extract 't '(1 2 3 4))
```

should return `(2 3 4)`. Also,

```
(extract 'th '(1 2 3 4))
```

should return 2.

```
(extract 'ht '((1) 2 3 4))
```

should return `nil`.

To convert a symbol to a string, one uses the *string* function. Like lists, one can take the head and tail of a string:

```
(car (string 'htt))
(cdr (string 'htt))
```

The two expressions above evaluate to `"h"` and `"tt"`.

To compare two strings, one should use the *equal?* function, which tests for structural equality, not the *eq?* function, which tests for pointer equality. Finally, the empty string `""` is equivalent to nil.

Example:

```
$ echo hht > task8.args
$ echo (((1 2)) 3) >> task8.args
$ scam -r task8.scm task8.args
(2)
$
```

9. Define a series of functions, named *big+*, *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)`.

   Example:

```
$ echo (1 2) > task9.args
$ echo (3 4) >> task9.args
$ scam -r task9.scm task9.args
(1 2) plus (3 4) is (4 6)
(1 2) minus (3 4) is (- 2 2)
(1 2) times (3 4) is (4 0 8)
(1 2) divided by (3 4) is (0)
$
```

   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.

   For efficiency reasons, you should define (at the top level) a set of analogous functions that perform the above operations with the digits reversed. These function names should have the prefix *r*. For example, the function *big+* should simply call *rbig+* with the digits of its arguments reversed. Try to make your operations relatively efficient.

10. Integrate your *big+*, *big-*, *big**, and *big/* functions into Scam's normal arithmetic functions. Do this by saving the old functions, as in:

```
(define old+ +)
```

and then redefining new ones, as in:

```
(define (+ a b)
    ...
    )
```

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 easy of testing, assume an regular integer ranges from -2^{15} to $2^{15} - 1$. Examples:

```
(+ 234 5)                       ;yields a regular int
(+ '(2 3 4) 5)                  ;yields a regular int
(+ 234 '(5))                    ;yields a regular int
(+ 32767 32767)                 ;yields a big int
(* 10000 4)                     ;yields a big int
(- '(4 0 0 0 0) '(1 0 0 0 0))   ;yields a regular int
```

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

Your *main* function should read two expressions and output the four results of adding, subtracting, multiplying, and dividing the two expressions, respectively, one result per line. An expression can either be a normal integer (within the restricted range given above) or a big integer.

Handing in the tasks

For preliminary testing, send me all the files in your directory by running the command:

```
submit proglan lusth test2
```

For your final submission, use the command:

```
submit proglan lusth assign2
```