# Scheming in Scheme

GitLab Register and submit here.

## Prefix Expressions in Java : 25 Points

### Description

You must write a Java program named Prefix that is executable directly from the command-line. This program must accept a command-line string denoting the name of an **ExP** file. An **ExP** file contains a list of **E-Expressions**, one per line. The Prefix program must read every **E-expression** in the file, evaluate the **E-expression**, and print the results in the order they appear in the **ExP** file.

### Syntax of an E-Expression

1. An **E-expression** is an algebraic, parenthesized, prefix expression involving the binary operators +, -, ^, %, *, and / along with integer literals, single-character variables, and environments. More precisely, we define the syntax of an **E-expression** below:
   a. An integer literal (one-or-more consecutive digits optionally preceded by a dash) is an **E-expression**.
   b. The word `undefined` is an **E-expression**.
   c. A single alphabetic character is an **E-expression**.
   d. If `E1` and `E2` are **E-expressions** and `ENV` is an environment (defined below), then each of the following is an **E-expression**.
      i. (+ `E1` `E2`)
      ii. (* `E1` `E2`)
      iii. (/ `E1` `E2`)
      iv. (^ `E1` `E2`)
      v. (% `E1` `E2`)
      vi. (BLOCK `ENV` `E1`)
2. If `E1` is an **E-expression**,
   - A `DECL` is of the form `( ID E1 )` where ID is a single alphabetical character denoting a variable and `E1`, when evaluated, gives the value of that variable.
   - An `ENV` is an environment given as a list of `DECL`s of the form ( `DECL`$_1$ `DECL`$_2$ ... `DECL`$_n$ )
3. Additionally, for ease of processing, we impose the constraint that there is at least one space between any two consecutive language elements.

### Semantics of an E-expression

The following ad-hoc rules describe the semantics of an **E-expression**:

1. The value of a binary expression (any **E-Expression** involving +, -, *, /, ^, %) is obtained by application of the specified binary operator to the value of it's two operands. You must support exponentiation (^) in a reasonable fashion; all of the arithmetic operators otherwise behave precisely like the integer-valued arithmetic operators in Java with the exception of 'undefined' as described in item 2. The result of division by zero should be undefined.
2. The value of an identifier is the value of the identifier as it exists in the closest containing environments list of definitions. Any variable whose value is not defined in any containing environment has a value of 'undefined'.
3. The value of an integer literal is itself.
4. The value of a block is the value of the blocks single **E-expression** where the blocks environment declares variables and associates a value with each declared variable. The declared variables are only accessible in the **E-expression** of the block and nowhere else. Note that this implies that declared variables are not accessible within the environment itself and that variables declared in the blocks environment take precedence over (they hide) variables declared outside of the block.

### Examples

```
( + ( - 10 2 ) 3 ) ---> 11
( + a 3 ) ---> undefined
( / 3 5 ) ---> 0
( block ( ( a 3 ) ( b 10 ) ) ( * a ( + b 1 ) ) ) ---> 33
( block ( ( a 3 ) ( b a ) ) ( + a b ) ) ---> undefined
( block ( ( x 3 ) ( y ( + 1 2) ) ( z ( block ( ( x 4 ) ) ( * x 2 ) ) ) ) ( + x ( + y z ) ) ) ----> 14
( block ( ( x 3 ) ) ( + ( block ( ( y x ) ) y ) 2 ) ) ---> 5
```

### ExP Files

An ExP file is a text file having one **E-expression** on each line. There may be whitespace before and after the **E-expression** of a line.

### Testing

I've created a test file and corresponding expected output.

## Scheme functions [75 Points]

You must write each of the following scheme functions. You must use only basic scheme functions (those described in class), do not use third-party libraries to support any of your work. Do not use any function with side effects.

1. [5 Points] Write a Scheme function (partition L N) that accepts a list L and an integer N. The function partitions L into lists of length N and returns these partitions as a list. The last element in the returned list may have a length that is less than N. If N is non-positive, it is treated as the value 1. For example:

```
(partition '(1 2 3 4 5) 2) ---> ((1 2) (3 4) (5))
(partition '(1 2 3 4) 1) ---> ((1) (2) (3) (4))
(partition '(1 (2 3) (4 5) 6 7 8) 2) ---> ((1 (2 3)) ((4 5) 6) (7 8))
(partition '(1 2 3 4) -12) ---> ((1) (2) (3) (4))
```

2. [5 Points] Write a function named (cycle ALIST N) that accepts a list of elements ALIST and an integer N. This function returns a list containing N repetitions of the elements of ALIST. If N is non-positive, this function returns the empty list. For example:

```
(cycle '(a b c) 3) ---> (a b c a b c a b c)
(cycle '(a (a b c) c) 1) ---> (a (a b c) c)
(cycle '(a b c) 0) ---> ()
(cycle '(a (a (a)) (a)) 3) --> (a (a (a)) (a) a (a (a)) (a) a (a (a)) (a))
```

3. [5 Points] Write a function named (list-replace ALIST SYM VAL) that accepts a list of elements and returns that list where all SYM's (a single symbol) have been replaced by the VAL (some scheme value). The replacement must occur even within nested lists. For example:

```
(list-replace '(a b c) 'a 3) ---> (3 b c)
(list-replace '(a (a b c) c) 'a 3) ---> (3 (3 b c) c)
(list-replace '() 'a 3) ---> ()
(list-replace '(a (a (a))) 'a '(3)) --> ((3) ((3) ((3))))
```

4. [5 Points] Write a function named (repeat VAL COUNT) that creates a list of length count where each element is given by VAL. COUNT is a non-negative integer and VAL is any scheme value.

```
(repeat 3 3) ---> (3 3 3)
(repeat 'a 9) ---> (a a a a a a a a a)
(repeat '(a) 5) ---> ((a) (a) (a) (a) (a))
```

5. [10 Points] Write a function (summer L) that takes a list of numbers L and generates a list of the runnining sums. See the following examples for clarification.

```
(summer '(1 2 3)) ---> (1 3 6)
(summer '()) ---> ()
(summer '(3 0 -2 3)) ---> (3 3 1 4)
```

6. [10 Points] Write a function (counts XS) that takes a list of items XS and generates a counting of the elements in XS. The returned object is a list of lists. Each element of the returned object is a list of length two containing an element X of XS and an integer denoting the number of occurrences of X in XS. The order of the elements in the computed list is not specified. For example:

```
(counts '(a b c c b b)) ---> '((a 1) (b 3) (c 2))
(counts '()) ---> '()
(counts '(1 3 c c #f)) ---> '((1 1) (3 1) (c 2) (#f 1))
```

7. [10 Points] Write a function (prefix L N) that takes a list L and integer number N. If the length of L is greater-than or equal-to N, the method returns the first N elements of L as a list. If N is negative, the method returns the empty list. If the length of L is less-than N, the method returns L. For example:

```
(prefix '(1 2 3 4 5 6) 2) ---> (1 2)
(prefix '(1 2 3 4 5 6) -1) ---> ()
(prefix '(1 2 3 4 5 6 7 8 9) 3) ---> (1 2 3)
(prefix '(1 2 3) 12) ---> (1 2 3)
```

8. [10 Points] Consider two techniques for representing a graph as Scheme lists. We can represent a directed graph as a list of edges. We call this representation an **el-graph** (i.e. edge-list graph). An edge is itself a list of length two such that the first element is a symbol denoting the source of the edge and the second element is a symbol denoting the target of the edge. Note that an edge is a list (not just a pair). For example, the following is a graph: '((x y) (y z) (x z)). We can also represent a graph similar to an adjacency matrix. We call this representation an **x-graph** (i.e. matrix-graph). In this case, a graph is a list of adjacencies where an adjacency is a list of length two such that the first element is a node (a symbol) and the second element is a list of the targets of that node. For example, the following is a graph: '((x (y z)) (y (z)) (z ())).
   - Write function (el-graph->x-graph g), that accepts an el-graph g and returns an x-graph of g.
   - Write function (x-graph->el-graph g), that accepts an x-graph g and returns an el-graph of g.

```
(el-graph->x-graph '((x y) (y z) (x z)) ) ---> ((x (y z)) (y (z)) (z ())).
(x-graph->el-graph '((x (y z)) (y (z)) (z ())) ---> ((x y) (y z) (x z)).
```

9. [15 Points] Write a function (evaluate exp) that takes an **E-expression** and evaluates that expression. In this problem, an the **E-expression** is a list rather than a String and we define the arithmetic operators differently than the Java version in that the operators are in all other respects precisely like the corresponding Scheme operators. See the following examples for clarification.

```
(evaluate '(block ((z 3) (q 2)) (+ z (/ 4 2)))) ---> 5
(evaluate '(block ((z 3) (q 2)) (+ z (/ (block ((q 3)) (+ q 1)) 2))) ---> 5
(evaluate '(block ((z 3) (a -3)) (+ p a))) ---> undefined
```

## Additional Requirements

1. You must submit all your work using GitLab using a project named `CS421` with a folder named **hw1**. Within this folder you must have a file named `scheme.rkt` along with your Java project (this can be packaged as a sub-folder containing your code). Here is a GitLab tutorial.
2. You must have the statmement `#lang racket` as the first line in your `scheme.rkt` file.
3. You must have the statement `(provide evaluate el-graph->x-graph x-graph->el-graph prefix rle rld repeat list-replace cycle partition)` as the last line in your `scheme.rkt` file.
4. The Scheme functions you submit must not use any imperative features. **Do not use set, while, display, and begin**! You will receive 0 points for any problem that uses these constructs. You may only use **define** to define functions; any other use will incur a loss of all points for that problem.
5. Do not write code that breaks a problem into more cases than necessary. Points will be deducted for including un-necessary cases.
6. You must follow good SE practices. The following apply to Java (and several apply to all code).
   - Associate a block with every conditional branch
   - A left-curly will never have a character after it on any line
   - All variable names begin with lower-case letters
   - All class names begin with upper-case letters
   - Use methods. If you write a method longer than about 20 lines of code, break it into other methods
   - Never copy-and-paste
   - No empty lines unless they separate things like blocks of code, methods, or variable declarations
   - Delete all spurious comments
   - Never print (unless you are writing a command-linen driven **program**)
   - Use spaces in expressions. A single space should surround every variable, number, and operator.
   - Never have a line longer than about 80 characters (this is flexible and I'm not going to get overly picky, but 80 is a good rule of thumb)
   - Don't create objects that you don't use
   - Don't have instance variable that should be local