Practice Exam

I. Suppose you had a language that was designed for programs that plotted data on the screen. Suppose this language allowed you to specify a position along the X axis by writing something like any of the following:

X: 3.2
X: +3.2
X:-0.5
X: 005.2

and similarly for the Y axis, but **not**:

Z: 12.0 (only X and Y axes are allowed, not Z, not anything else)
X: + 3.2 (no spaces after a sign are allowed)
X: 3. (no trailing decimal points are allowed)
X: .2 (no leading decimal points are allowed)

In other words, a position is:

- an axis specifier, either 'X:' or 'Y:' (with no space between the X or Y and the :)
- zero or more spaces after the :
- an optional sign, '+' or '-', with no spaces after a sign
- one or more digits (leading 0s are ok)
- decimal point
- one or more digits

Write a Regular Expression for the set of string that are legal positions. You may write a space character as ␣. You may use the terms letter, non0, and digit to stand for lower case letters. non-0 digits (1 through 9), and all the digits (0 thought 9), respectively.

For the Scheme questions below, all repetition must be done by recursion. (The recursion implicit in functions like map and assoc is ok to use.). You **may** write and use additional functions if you wish. You may **not** use `do` or any built-in function whose name ends in '!', e.g. you may not use `set!`. You **may** use any other function in R5RS Scheme including the following:

Expression	Value		Expression	Value
(map sqrt '(9 1 4))	(3 1 2)		(reverse '(a (b c) d))	(d (b c) a)
(member 'a '(b c a d a))	(a d a)		(list (+ 2 3) '(a))	(5 (a))
(member 'x '(b c a d))	#f		(cons (+ 2 3) '(a))	(5 a)
(assoc 'x '((a b) (x y)(q r)))	(x y)		(append '(a b) '(c d))	(a b c d)

(null? x) is true if x is the empty list ( ), (eq? x y) is like Java's x = = y,

II

The function (foo fn-pair) takes one argument, a list of two functions, and returns a function as its result. This returned function should take a numeric argument and apply to it either the first element of fn-pair (if this numeric argument is less than 0) or the second element of fn-pair (if the numeric argument is 0 or greater). E.g., ((foo (list (lambda (x) (+ x 10))
(lambda (x) (* x x))))
-4)
should return 6, since (+ -4 10) is 6.

III

Define the macro (chain2 init (x bodyx) (y bodyy)) where x and y are variables and bodyx and bodyy are expressions that refer to x and to y. E.g.,
(chain2 3 (a (* a a))
(b (+ b 1)))
returns 16 because 3 + 1 is 4 and 4 * 4 is 16. Chain2 may translate into a call to chain.

IV

The function (count x lst) takes arguments x, a symbol, and lst, a list, and returns the number of times x appears at the top level in lst. E.g., (count 'a '(a (b a) c a)) returns 2, since the a in (b a) is not at the top level. **count and any helper functions must be tail-recursive.**
(define (count x lst)

V

Define the function (all-true fn-list value) which takes a list of single-argument functions, and a single value. It applies the functions one by one to the value. If all of the functions return true, all-true returns #t. If any function returns #f, all-true returns #f. If fn-list is ( ), all-true returns #t. The call
(all-true (list (lambda (x)(>= x 5))
(lambda (x)(<= (sqrt x) 10)))
4)
returns #f, since applying the first function in the list to the value 4 returns #f.

Solution

I

```
( X | Y ): Ø/* ( + | - | ε ) digit + . digit +
```

II

```
(define (foo fn-pair)
  (lambda (x)
    (if  (< x 0)
         ((car fn-pair) x)
        ((cadr fn-pair) x))))
```

III

```
(define-syntax chain2
  (syntax-rules ( )
    ((_   init (x bodyx)(y bodyy))
     (chain init (list (lambda (x) bodyx)
                       (lambda (y) bodyy))))))
```

IV

```
(define (count x lst)
  (count-helper x lst 0))
(define (count-helper x lst accum)
   (cond ((null? lst) accum)
         ((eq? (car lst) x)   (count-helper x (cdr lst) (+ accum 1)))
         (else (count-helper x (cdr lst) accum))))
```

V

```
(define (all-true fn-list value)
    (if (null? fn-list) #t
        (and ((car fn-list) value)
             (all-true (cdr fn-list) value))))
```