1. Special forms are required in programming languages because usage of only normal forms is limiting. For example, we may want to use variable x to divide a number given by the user, but there's a chance x=0, and dividing by 0 is an arithmetic error. By using spacial form we can write the following code:
   (if
   (= x 0)
   num
   (/ num x))
   The definition of x and num are also done by using special forms.
   the reason we can't define the special forms as primitive is that there's a need for a different computation procedure for them, as the evaluation of non special forms is the same for all non special forms.
2. <u>can be done in parallel:</u>
   (+ 4 5)
   (* 7 8)
   (* (/ 6 2) (+ 2 3))

   <u>cannot be done in parallel:</u>
   (define x 10)
   (define y 20)
   (* x y)
3. Considering L0 which has no "define" statement, each program in L1 can be translated to L0. That's because define is simply having the option to name an expression, and the expression itself is made of primitive types- also included in L0. So if a program in L1 has a (define "name" <expression>), we just need to translate each "name" in the program to <expression>.
4. Let's look at the following program wrote in L2 that cannot be transferred to an equivalent program in L20:
   ```
   (define fact
       (lambda (a)
           (if (= a 0)
               1
               (* a (fact (- a 1))))))
   ```
   the code above cannot be translated to a L20 program because without "define" we can't create a recursive method.
5. **<u>Map-</u>** can be done in parallel because the function given to map is operating on each item in the item list separately, with no connection between different items.
   <u>Reduce-</u> must be done sequentially because the acc variable may change over the iterations, and it could have different effects on the items in the list.
   <u>Filter-</u> can be done in parallel because each item need to be #t in the filter function in order to be in the returned list, with no relation to other items on the list.
   <u>All-</u> can be done in parallel. each thread can return true or false according to the item it checked with the given function, and another thread checking whether all the threads returned true or not.
   <u>Compose-</u> must be done sequentially because when composing different functions, the order in which those functions are composed can be important, e.g if you want to add 5 to a variable and then calculate its square root, you shouldn't calculate the variable square root and then add 5.

6. The value of the program given will be 9, that's because a and b the global variables aren't the a and b defined in pair class. so after the creation of variable p34, a will be 3 and b=4. C how ever is a global variable, and since there's no new defenition of c inside the pair class, the calculation done by f is: 3+4+2=9.

Q.2.1
Signature: append(lst1 lst2)
Type:[list<any>*list<any> => list<any>]
Purpose:concat two lists into one
Pre-conditions:true
Tests: (append '(1) '(2) => '(1 2))

Q.2.2
Signature:reverse (lst1)
Type:[list<any> => list<any>]
Purpose:reversing a list
Pre-conditions:true
Tests:(reverse '(1 2) => '(2 1))

Q.2.3
Signature:duplicate-items (lst1 lst2)
Type:[list<any>*list<number> => list<any>]
Purpose:duplicating items from lst1 according to the matching index value in lst2
Pre-conditions: eq? (lst2) = false
Tests:(duplicate-items('(1 2 3) '(2)) => '(1 1 2 2 3 3))

Q.2.4
Signature:payment (sum lst)
Type:[number*list<number> => number]
Purpose:determine how many payment methods there's with the coins in lst in order to pay sum
Pre-conditions: sum of type number, lst not empty
Tests:payment (3 '(1 2)) => 1

Q.2.5
Signature:compose-n (fun num)
Type:[function*number => any]
Purpose:composing the function given num times and return it's closure
Pre-conditions:true
Tests: compose-n(lambda(x) (+ x 1)) 3)(2) => 5