Q1.1 – Special forms are required in programming languages to make special evaluations in the programming language to answer for specific needs of the programmer such abstraction to improve readability of the code and more. If we would define the special forms as primitives, the evaluation will be the same evaluation as the normal form, and we would not get the output we are expecting. For example, there is special form (if <test-part> <then-part> <else-part>) where then-part or else-part are evaluated only when the test is true or false, respectively. If the special form "if" was a primitive, then the program:

```
(if
 (= x 0) ;; test-part
 1 ;; then-part
 (/ 1 x)) ;; else-part
```

Will not consider the "if" and might evaluate division by zero.

Q1.2 – The following program's expressions can be run in parallel:

```
(+ 2 3)

(+ 1 4)
```

This is because there is no dependency between them.

On the other hand, the following program's expressions cannot be run in parallel:

```
(define a 2)

(+ 1 a)
```

This is because when evaluating the second expression, the value of a is needed, but the value is only accessible after the first expression has been evaluated.

Q1.3 – A program in the L0 language can always be transformed to an equivalent program in L1. Because L1 is using only numbers, boolean expressions, primitive operations, and variables. Thus, every variable defined in L1 is either number or boolean expressions and we can replace with the number/boolean we defined with the congruent number/boolean expressions.

Q1.4 – A program in the L20 language can not always be transformed to an equivalent program in L2. For example, suppose the following program in L2:

```
(define a (lambda (b)
          (if (< b 10)
               (a (+ b 2))
               b)))
```

Notice that the program contains a recursive call for the function a, and we cannot call the recursive call in L20 (using a lambda expression instead of ‘a’).

Q1.5

map - the order of procedure can be applied in parallel. This is because the function is applied to each element independently of the other elements. If the same element appears in two different lists, the function will return the same value after being applied to it.

reduce – the order of procedure must be sequential, because when applying the function on a certain element, the acc is affected by previous calculations. That is, if the same element appears into different lists, the function might return a different value after being applied to it in each case. Therefore, we cannot apply the function to the second element before the first element has finished its computation.

filter – the order of procedure can be applied in parallel. Each element is checked against the predicate, and there is no dependency on previous calculations.

all – can be applied in parallel. Each element is checked against the predicate, and only if all of them return true then the function will return true. We can check whether each element upholds the condition, and then apply ‘and’ to all results. There is no dependency between the calculations.

Compose – must be sequential. This is because each function in the list (except the first) should be composed with both the main function and the previous functions in the list. So, the results of composing the previous functions are needed for the current evaluation. Thus each element must be applied to the function in order.

Q1.6 – The value of the following program will be 9. When creating the Pair object, the program will look for the value of ‘c’ in the f function. It will look outwards from the f function until it finds a declaration for ‘c’. The first declaration it finds appears in line 2, so ‘c’ will be 2.
When calling the f function from the lambda in line 12, ‘c’ is already assigned a value, thus the value that the lambda receives is ignored.

Part 2 Contracts

```
; Signature: append(lst1, lst2)
; Type: [List(any) * List(any) -> List(any)]
; Purpose: concatenate 'lst1' and 'lst2' to one list
; Pre-conditions: lst1 and lst2 are lists
; Tests: (append '(1 2) '(3 4)) ==> '(1 2 3 4)


; Signature: reverse(lst)
; Type: [List(any)->List(any)]
; Purpose: To return a list whose elements are in reverse
;          order to the ones in 'lst'
; Pre-conditions: lst is a List
; Tests: (reverse '(1 2 3)) ==> '(3 2 1)
; Signature: duplicate-items
; Type: [List(T) * List(Number) -> List(T)]
; Purpose: duplicate each item of the first list according to the number in
               the second list.
; Pre-conditions: dup-count (second list) contains numbers and is not empty.
; Tests: (duplicate-items '(1 2 3) '(1 0)) ==>0 2)) ==> '(1 1 2)


; Signature: payment(n, coins-lst)
; Type: [Number*List(Number) -> Number]
; Purpose: To count how many ways there are to reach 'n' with
;           the coins in 'coins-lst'
; Pre-conditions: n is a whole Number, coins-
lst is a list of positive whole numbers
; Tests: (payment 10 '(5 5 10)) ==> 2
;        (payment 5 '(1 1 1 2 2 5 10)) ==> 3


; Signature: compose-n
; Type: [(T -> T) * Number -> (T->T)]
; Purpose: composing a function n times with itself.
; Pre-conditions: n is bigger then 0
; Tests: (define mul8 (compose-n (lambda (x) (* 2 x)) 3))
;        (mul8 3) ==> 24
```