1.1 Special forms are required because sometimes we want to make a procedure where we don't necessarily want to evaluate the arguments before the execution of the body (it may cause an error of some kind) but rather evaluate only the right argument.
Quick example:

```
(if (= x 0)
    2
    (/ 1 x))
```

If the 'if' was not special form by the Scheme language, the 'else' statement would be evaluated before the body, leading to a division by zero.

1.2 For the first question, something like this would work:

```
(+ (+ 1 2) 5)
```

One thread can evaluate the first expression, and the second can evaluate the second expression.

For the second question, something that includes 'define':

```
(define x (+ 1 2))
```

Here, we cannot evaluate 'x' and (+ 1 2) simultaneously, because the evaluation of 'x' is a bind procedure on the evaluation of (+ 1 2). So first we have to wait for the evaluation of (+ 1 2) to finish, and only then bind it to 'x'.

1.3 Each program of L1 can be transcribed to L0 language.
If the program of L1 does not include the word 'define', thus the L0 program will be the same.
On the other hand, if 'define' is included, that means we evaluated some expression and bonded it to some variable (for example 'x' holds some value). Then, on other expression, we use 'x' and there is no need to re-write the whole expression again and evaluate it.
So, each time 'x' is written in L1, we must copy the expression that is bonded to 'x' and paste it instead of 'x'.
Quick example:
In L1 we write:

```
(define x
        (+ 1 2))
x                       ; Printing the value bonded to 'x'.
```

In L0 we write:

```
(+ 1 2)
```

1.4 Each program in L2 can be transcribed to L20 language.
The same explanation as above can be applied here.
'define' is no more than evaluation of some expression and binding it to a 'var' name.
Thus, each time the 'var' is used, we can remove it and paste the whole expression, that will be evaluated again.

1.5 Map – This procedure calculates a given function 'f' on each value of a given 'list', then returns a new list where for each value $a \in 'list'$ we have $f(a)$ in the same order.
Each evaluation is independent of the other, thus we can run each evaluation on a different thread.

Reduce – In this procedure the sequence is important. We have the 'accumulator' which holds some information on all members in the list that were already traversed. Because of this we cannot run the evaluations in parallel.

Filter – This procedure takes some predicate and checks if $a \in 'list'$ answers the predicate.
If true, then a will be in the new list returned by 'Filter'. It doesn't really matter how we traverse the given list, it won't affect the returned list. Thus, we conclude that parallel run is possible in this case.

All – In this procedure, it is enough that one member in the given 'list' returns '#f' and the procedure returns '#f'. We can evaluate 'All' on each member alone, receive '#f' or '#t' and then check if all returned values are '#t' leading to 'All' return '#t', or whether there is at least one element that returned '#f', leading to 'All' return '#f'. So, in this method we can indeed work with concurrency.

Compose – We already know that for given two function f, g the sequence of composition is important. $f(g(x))$ is not necessarily $g(f(x))$. Thus, here we cannot work in a parallel manner.

1.6 The value of this program is 9.
In this line: (define p34 (pair 3 4)) a closure is returned, then a bind occurs where $a = 3$ and $b = 4$.
In the next section: ((lambda (c) (p34 'f)) 5) We bind the value 5 to c, then evaluation (p34 'f). The evaluation is simply an evaluation of a closure which has zero parameters. So, the $c = 5$ is overlooked. This is how the evaluation looks like: $(+\ 3\ 4\ 2)$, these are the values of a, b, c respectively.

First procedure:

; Signature: append (lst1, lst2)

; Type: [List(any) * List(any) -> List(any)]

; Purpose: Make a list that represents the concatenation of the second list to the first list.

; Pre-conditions: None

; Tests: (append '(1 2) '(3 4)) -> '(1 2 3 4)

Second procedure:

; Signature: reverse (lst)

; Type: [List(any) -> List(any)]

; Purpose: Reverses the given list.

; Pre-conditions: None

; Tests: (reverse '(1 2 3)) -> '(3 2 1)

Third procedure:

; Signature: duplicate-items(lst, dup-count)

; Type: [List(any) * List(Number) -> List(any)]

; Purpose: Duplicates element indexed 'i' in 'lst' dup-count[i] times. If dup-count's size is less than lst, we act as if it is a circular list.

; Pre-conditions: None

; Tests: (duplicate-items '(1 2 3) '(1 0))→ '(1 3)

(duplicate-items '(1 2 3) '(2 1 0 10 2))→ '(1 1 2)

Fourth procedure:

; Signature: payment(n coins-lst)

; Type: [Number * List(Number) -> Number]

; Purpose: Counts all ways to change given value 'n', with available coins in 'coins-lst'.

; Pre-conditions: None

; Tests: (payment 10 '(5 5 10)) → 2

[1 coin of 10 ,2 coins of 5]

(payment 5 '(1 1 1 2 2 5 10) → 3

[1 coin of 5, 1 coin of 2 and 3 coins of 1, 2 coins of 2 and 1 coin of 1]

Fifth procedure:

; Signature: compose-n(f n)

; Type: [[Number -> Number] * Number -> [Number -> Number]]

; Purpose: Returns the closure of the n-th self-composition of f.

; Pre-conditions: ‘f’ must be a unary function, that gets a Number an returns a Number, and ‘n’ must be a number.

; Tests: (define mul8 (compose-n (lambda (x) (* 2 x)) 3))

(mul8 3) → 24