Assignment 2

1.1 Functional programming doesn't require few expressions in the body of the functions, because it requires only when those expressions has "side effects".
We should remember that a functional programming shouldn't has "side effects".
It might be that we will have few expressions but it will occur as while as in any running of the program the same input will cause the same output.
In a procedural programming the usage of few expressions in the body of the functions is more useful.

1.2 a. Special forms required in programming languages because it allows us to use our own evaluation rules and therefore they can't be defined as primitive operators. That is, Scheme (or any other programming language) doesn't evaluate all the subexpressions, instead, each special form has its own evaluation rules. For example: 'if' special form, (if #t 1 0).

b. The "Or" operator can be computed as a primitive operator and then it will be computed as a default operator, meaning that in case of an "or" operator and afterwards 2 boolean expressions, we will compute each expression and we will apply the or operation only after.

If we take the shortcuts semantics in mind, we will always rather set the "or" operator as a special form and in this way we will first check the truth of the first expression and will continue the second just in case of a positive result.
If the first expression is false, we won't continue to the second one, and this is why in this case it is more efficient.

1.3 In computer science, syntactic abbreviation is a syntax within a programming language that is designed to make things easier to read or to express. It makes the language "sweeter" for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer.

i.e a. instead of writing : "x = x+y ", one can write x +=y .

b. instead of writing : "x = x+1 ", one can write x++ .

1.4 a. The value is 3. First, we defined the variable x to be 1, then we evaluated both expressions, thus giving us y = 3*1=3 .

b. The value is 15. First, we defined the variable x to be 1, but then with our "Let*" special form we re-defined it to be 5. Then, we evaluated the expression and put it into y.

c.

```
(define x  2)

(define y  5)

(let

  ( (x  1)

   (f (lambda (z) ([+free] [x : 2 0] [y : 2 1] [z : 0 0] ) ) ) )
```

([f : 0 1] [x : 0 0]))

(let*

((x 1)

(f (lambda (z) **([+ free] [x : 1 0] [y : 2 1] [z : 0 0]**))))

([**f : 0 1] [x : 0 0])**)

d.

(let (x 1)

(let (f (lambda (z) (+ x y z))

(f x))

e.

(lambda (x f)

(f,x))

(1 (lambda (z) (+ x y z))))