1.1: No a function-body with multiple expressions isn't a requirement in a purely functional programming language. That's because in such a language a function can only return a value and can't change the global state. And any setting of the inner state of a function can be rewritten into a single expression.

It would be useful in languages where it's possible to change the state of the program (e.g., Object Oriented, Procedural, etc.)

1.2: **a)** Special forms are needed to extend functionality of the language to beyond what pure functions are capable of doing (e.g., execution flow control, defining a new function, defining a variable.)

We can't simply define them as primitive operators because they can have operands that don't need to be evaluated, as opposed to primitive operators, which evaluate all of their arguments (e.g., the name of the variable declared with define, or the parameter names in the lambda function.)

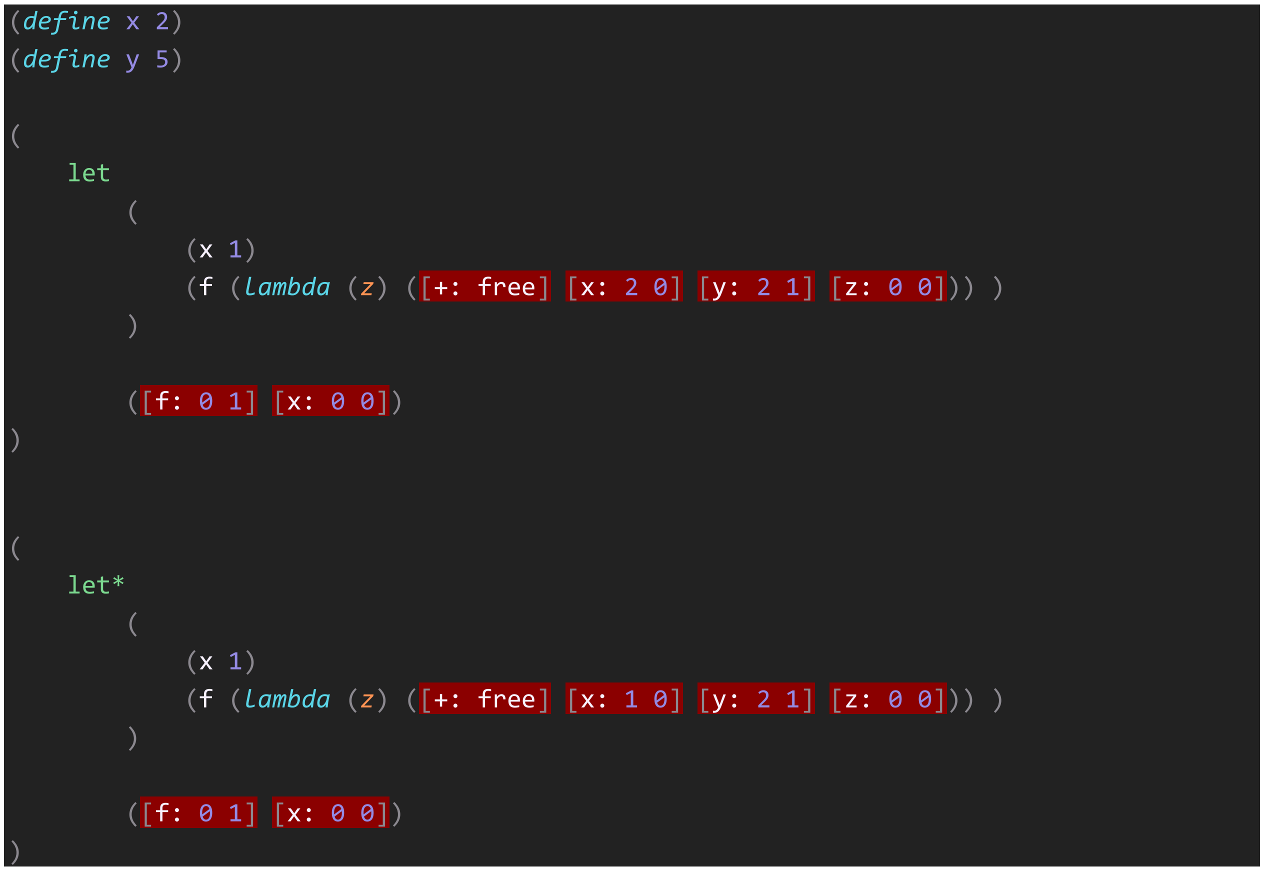
b) No. The logical operation "or" implements the shortcut semantics principle, meaning that it evaluates the given operands until one of them results in "#t" value or until all of them evaluate to "#f". This functionality isn't possible with primitive operators, which have to evaluate all of their operands.

1.3: Syntactic abbreviation is creation of new, more convenient and readable, syntax for already existing functionality in the language. Examples: a) the "let" structure as a syntactic abbreviation of an application of a lambda function; b) the "cdddr" abbreviation of 3 consecutive "cdr" commands.

1.4: **a)** The value is 3. At the moment of evaluation of the value of "y", the "x" that gets defined inside the "let" structure isn't appearing yet in the bindings of the closure, since "let" first evaluates all of its declarations, and only then updates them in the bindings. So the value of "y" is 1*3=3.

b) The value is 15. The construct "let*" evaluates and binds it's declarations sequentially, meaning that at the moment of evaluation of "y", the "x" it refers to is the one declared a line prior to it. So the value of "y" is 5*3=15.

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)

```scheme
(define x 2)
(define y 5)

(
    let
    ( ;variables of the first "let"
        (x 1)
    )

    ( ;return value expression of the first "let"
        let
        ( ;variables of the second "let"
            (f (lambda (z) (+ x y z)))
        )

        (f x) ;return value expression of the second "let"
    )
)
```

e)

```scheme
(define x 2)
(define y 5)

(
    (lambda (x)
            (
                (lambda (f)
                    (f x)   ;the return value of the whole thing
                )

                (lambda (z) (+ x y z)) ;defining the "f" var
            )
    )

    1 ;defining the "x" var
)
```