Q1: Is a function-body with multiple expressions required in a pure functional programming? In which type of languages is it useful?

No, multiple expressions aren't required in pure functional programming. Its useful in functional languages because its possible to write expression with side effects without changing the purety of the function

Q2:

a. A special form is a primitive function specially marked so that its arguments are not all evaluated. Most special forms define control structures or perform variable bindings—things which functions cannot do.

   Each special form has its own rules for which arguments are evaluated and which are used without evaluation. Whether a particular argument is evaluated may depend on the results of evaluating other arguments.

   We can't define them as primitive operators because each special form has its own rule to be calculated, like the or operator, where we have a shortcut semantic.

b. The or special form tests whether at least one of the conditions is true. It works by evaluating all the conditions one by one in the order written.

   If any of the conditions evaluates to a non-nil value, then the result of the or must be non-nil; so, or returns right away, ignoring the remaining conditions. The value it returns is the non-nil value of the condition just evaluated.

   If it was defined as a primitive operator, then all the conditions would've been calculated first, which is not efficient, that's a shortcut semantic, where when of the conditions is satisfied, the or operation returns immediately without the need to evaluate the rest of the conditions, this won't be possible as a primitive operator, where every arguments gets calculated first.

Q3:

syntactic abbreviation is 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.

Example 1:

combinations of car and cdr: the "cdd…dr" procedures:
(caddr c) ;; equivalent to (car (cdr (cdr c)))

Example 2:

Abstractly, an array reference is a procedure of two arguments: an array and a subscript vector, which could be expressed as get_array(Array, vector(i,j)). Instead, many languages provide syntax such as Array[i,j]

Q4:

a. What is the value of the following L3 program? Explain?

The value: 3

x was defined with the value 1 first, then when we entered the let expression, x was used inside the let binding (* x 3) to assign it as the value for y. the usage of a variable inside the binding and not the body of the expression refers to the x that was defined first and not the x that is still in the binding process. Therefore y = (* 1 3) = 3

in other means, the initial values are computed before any of the variables become bound.

b. What is the value of the following program? Explain?

The value: 15

Let* is similar to let, but the bindings are performed sequentially from left to right, and the region of a binding indicated by (<variable> <init>) is that part of the let* expression to the right of the binding. Thus the second binding is done in an environment in which the first binding is visible, and so on.

Based on that, after entering the let*, a new variable x was assigned with value 5 , and was visible to the next binding , in which y was assigned with the value x*3 = 5*3 = 15.

c. Annotate lexical addresses in the given expression

```
(define x 2) <0>
(define y 5) <1>

(let
  ((x <0> 1)
    (f <1> (lambda (z <0>) (+ x < 1,0 > y <1,1> z <0,0>))))
  (f<0,1>  x <0,0>))

(let*
  ((x<0> 1)
    (f<1> (lambda (z<0>) (+ x<0,0> y<1,1> z<0,0>))))
(f<0,1> x<0,0>))
```

d. Define the let* expression in section c above as an equivalent let expression

```
(let ((x1))
     (let ((f (lambda (z) (+ 1 y z))))
(f x)))
```

e. **Define the let* expression in section c above as an equivalent application expression (with no let)**

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

Programming in L3

Contracts:

; Signature: make-ok (val)
; Type: [T -> Result<T>]
; Purpose: return pair which encapsulates val as an ok structure type result.
; Pre-conditions: True
; Tests: (make-ok 5) => (“ok”. 5)

; Signature: make-error (msg)
; Type: [String -> Result<String>]
; Purpose: return pair which encapsulates the msg as an ‘error’ structure type result.
; Pre-conditions: True
; Tests: (make-error “sad face”) => (“error”. “sad face”)

; Signature: ok? (res)
; Type: [any ->Boolean]
; Purpose: return whether res as an ok structure or not.
; Pre-conditions: True
; Tests: (ok? (“ok”. 5)) => #t
(ok? (“error”. “sad face”))=> #f

; Signature: error? (res)
; Type: [any ->Boolean]
; Purpose: return whether res as an error structure or not.

```
; Pre-conditions: True
; Tests:  (error?  (“ok”. 5)) => #f
            (error? (“error”. “sad face”))=> #t
; Signature: result? (res)
; Type: [any ->Boolean]
; Purpose: return true if res as an result type.
; Pre-conditions: True
; Tests:  (result?  (“ok”. 5)) => #t
            (result? (“error”. “sad face”))=> #t

; Signature: result->val (res)
; Type: [Result<T> ->T]
; Purpose: return the value of the result type structure.
; Pre-conditions: res is pair from type result
; Tests:  (result->val  (“ok”. 5)) => 5
            (result->val (“error”. “sad face”))=> “sad face”

; Signature: bind (f)
; Type: [(T1 -> Result<T2>) => Result<T1> -> Result<T2>]
; Purpose: return function from non-result parameter to result and
returns this function from result to                result.
; Pre-conditions: f Is a function
; Tests:  (bind (lambda (x) (make-ok (* x x)) 5) => (“ok”. 25)

; Signature: make-dict ()
; Type: [void -> Dict]
; Purpose: returns empty dictionary.
; Pre-conditions: True
; Tests:  (dict) => ‘()
; Signature: dict? (e)
; Type: [any -> Boolean]
; Purpose: return true if res as an dictionary.
; Pre-conditions: True
; Tests:  (dict? (dict)) => #t

; Signature: get(dict, k)
; Type: [Dict(T1, T2)*T1 -> Result<T2>]
```

; Purpose: find a value in an dictionary given its key.
; Pre-conditions: True
; Tests: (get (put (dict) 3 4) 3) => 4

; Signature: put (dict, k, v)
; Type: [Dict(T1, T2)*T1*T2 -> Result<Dict(T1, T2)>]
; Purpose: returns a result of a dictionary with the addition of the given key and value
; Pre-conditions: True
; Tests: (get (put (dict) 3 4) 3) => 4

; Signature: map-dict (dict, f)
; Type: [Dict(any, T1)*T2 -> Result<Dict(any, T2)>]
; Purpose: returns a result of a new dictionary with the resulting values from the function
; Pre-conditions: True
; Tests: (result->val (get (result->val (map-dict (result->val (put (result->val (put (make-dict) 1 #t)) 2 #f)) (lambda (x) (not x)))) 1)) => #f

; Signature: filter-dict (dict, pred)
; Type: [Dict(T1, T2)*(T1*T2) -> Boolean => Result<Dict(T1, T2)>]
; Purpose: returns a result of a filtered dictionary that contains only the key-values that satisfy the predicate
; Pre-conditions: True
; Tests: (result->val (get (result->val (filter-dict (result->val (put (result->val (put (make-dict) 2 3)) 3 4)) lambda (k v) (and (even? k) (odd? v))) 2)) => 3