Question 1:

1.1 Is a function-body with multiple expressions required in a pure functional programming? In which type of languages is it useful?
No it is not, for example:

RandomString === “messi” ? “messi” : “ randomString is not equal to messi”

RandomString is a variable outside the scope of the function, and we can see that we don’t change its value therefore there is no mutation or assignment.

On the other hand, having multiple expressions is useful in procedural programming.

1.2

a. Special forms are required in case the semantics of the evaluation does not follow the default ‘procedure application’ semantics, *i.e.,* evaluation of the operator and the operands + application. The ‘if’ special form, for example, does not required pre-evaluation of the ‘then’ and the ‘else’ sub-expressions. In case ‘if’ was a primitive operator rather than a special form, the evaluation of the following expression would cause an error: (if (> 3 4) 5 (\ 6 0))

b. It must be defined as a special form, because the expressions will be evaluated from left to right, and its in our interest to stop the calculation once we get a true statement, to save time and continue the procedure.

1.3

is a syntax designed to make programming easier to read or articulate and easier to use. example:

1. let is a syntactic abbreviation of lambda:
   (let ((g 15) (k 16) (*s y)), this will be replaced by the equivalent syntactic form:

   ((lambda (g k) (* g k) 15 16)

2. cond is a syntactic abbreviation of if:

   ```
   (cond (((> g 15) 16)
    ((> k 3) 4)
   (else 6))
   ```

   this will be replaced by the equivalent syntactic form:

   ```
      if(>g 15)
            16
   ```

```
            (if(> k 3)
               4 6))
```

1.4

a. 3

Explanation: the initial values are computed before any of the variables become bound Therefore it is * 1 3 = 3

b . 5

Explanation: in a `let*` expression, the bindings and evaluations are performed sequentially therefore it is * 5 3 = 15

c.

```
["define" "free"] [ "x" "free"] "2"
        ["define" "free"] [ "y" "free"] "5"
[ [ "let" "free" ] [ [ [ "x" "free" ] "1" ] [ [ "f" "free" ] [ "lambda" [ "z" ] [ [ "+" "free" ] [
"x" "free"]  [ "y" "free"] [ "z" ":" "0" "0" ] ]]]]
[ [ "f" "free" ] [ "x" "free" ] ] ]
[ [ "let*" "free" ] [ [ [ "x" "free" ] "1" ] [ [ "f" "free" ] [ "lambda" [ "z" ] [ [ "+" "free" ]
[ "x" "free" ] [ "y" "free"] [ "z" ":" "0" "0" ] ] ] ] ]
[ "f" "free" ] [ "x" "free" ] ] ] ]
```

d. (define x 2) (define y 5) (let ((x 1) (f (lambda (z) (+ x y z)))) (f x)) (let ((x 1)) (let ((f (lambda (z) (+ x y z)))) (f x)))

```
e. (define x 2)
  (define y 5)
((lambda ( x f) (f x) )
1
(lambda (z) (+ x y z)))
((lambda (x) ((lambda (f) (f x))
        (lambda (z) (+ x y z)))
1)
```

```
; Signature: make-ok(val)
 ; Type: result
 ; Purpose: gets a value and encapsulates it as an ok structure of type result
 ; Pre-conditions: none
 ; Tests: make-ok(6) => (ok,6)

; Signature: make-error(msg)
 ; Type: result
 ; Purpose: gets an error string and encapsulates it as an ‘error’ structure of type result
 ; Pre-conditions: none
 ; Tests: make-error(“result is error”) => (Error, result is error)

; Signature: ok?(res)
 ; Type: boolean
 ; Purpose: - type predicate for ok
 ; Pre-conditions: none
 ; Tests: ok?(make-ok(6) => true



; Signature: error?(res)
 ; Type: boolean
 ; Purpose: type predicate for error
 ; Pre-conditions: none
 ; Tests: error?(make-error(“error”)) => true



; Signature: result?(res)
 ; Type: boolean
 ; Purpose: type predicate for result
 ; Pre-conditions: none
 ; Tests: result?(ok?(res)) => true

; Signature: result->val(res)
 ; Type: result
```

; Purpose: returns the encapsulated value of a given result: value for ok result, and the error string for error result
; Pre-conditions: res is of type result
; Tests: result->val(res) => cdr(res)

; Signature: bind(f)
; Type: lambda
; Purpose: gets a function from non-result parameter to result and returns this function from result to result.
; Pre-conditions: f is function
; Tests: bind(f) => F

; Signature: make-dict()
; Type: Dictionary
; Purpose: returns a new empty dictionary
; Pre-conditions: none
; Tests: make-dict() => ‘()

; Signature: dict?(param)
; Type: boolean
; Purpose: type predicate for dictionaries
; Pre-conditions: none
; Tests: dict?(‘((1,2))) => true

; Signature: get(dict, k)
; Type: the type of k
; Purpose: gets a dictionary and a key, and returns the value in dict assigned to the given key as an ok result. In case the given key is not defined in dict, an error result should be returned
; Pre-conditions: dict?(dict) => true, k is in dict
; Tests: dict = ‘((1.5)), get(dict 1) => 5

; Signature: put(dict, k, v)
; Type: dictionary
; Purpose: - gets a dictionary, a key and a value, and returns a result of a dictionary with the addition of the given key-value. In case the given key already exists in the given dict, the returned dict should contain the new value for this key.
; Pre-conditions: dict?(dict) => true
; Tests: dict = ‘(), put(dict 1 5) => ‘((1.5))

; Signature: map-dict(dict, f)

; Type: dictionary
; Purpose: gets a dictionary and an unary function, applies the function of the values in the dictionary, and returns a result of a new dictionary with the resulting values
; Pre-conditions: dict?(dict) => true, f is valid function
; Tests: f multiples by 2, dict =’((1.2)), map-dict(dict f) => ‘((1.4))

; Signature: filter-dict(dict, pred)
; Type: Dictionary
; Purpose: gets a dictionary and a predicate that takes (key value) as arguments, and returns a result of a new dictionary that contains only the key-values that satisfy the predicate
; Pre-conditions: : dict?(dict) => true, pred is valid function
; Tests: pred removes odd numbers, dict = ‘((1.1)), filter-dict(dict, pred) => ‘()