Assignment 2: PPL

Question 1:

1.1:

Function-body with multiple expressions is not required in pure functional programming languages. In functional programming there are no mutations(side-effects), as a result all the expressions that were in the middle will not change other variables nor return values. Only the last expression affects our program. Therefore, function with the last expression only and function with a multiple expression will affect our program the same. We can use functions with only one expression.

On the other hand, in non-pure functional programming languages, we have mutations and the expressions in the middle of the function can take effect on our program and even make our code more efficient.

1.2:

a.

Special forms are required in programming languages, and we cannot define them as primitive operators because the special form has a different structure/logic than primitive ones. For example, in L2 we have the primitive operator + that this structure is to compute all the other expressions in the compound expression and then sum the results. All the primitive operators share this kind of structure. In contrast, the define operator has a different structure, we compute the other expression and then bind the result with the name of the variable. We do not compute the value of the variable.

b.

Logical operation 'or' must be defined as special form because we don't compute all the expressions and then we compute the operation 'or' with the results. Logical operation 'or' is shortcut semantics.

1.3:

Syntactic abbreviation refers to a expression that can be defined by other syntactic constructs. when we define the operational semantics of the language, we don't need to define a new computation rule to compute for this expression type, instead we indicate that this expression is equivalent to a combination of other syntactic constructs that mean the same thing.

For example, Let is a syntactic abbreviation- It can be defined by lambda in L3. Another example is "switch case" in java script which can be defined as "if-else" so "switch case" can be syntactic abbreviation if we wrote java script.

1.4:

a.

The value of this L3 program is the number 3 because in Let evaluation process we first compute the value of each Val in each "binding" in the Let expression. By this time the variable "x" is not associated with the value 5. Therefore, x is "bounded" with 1 in the environment. So, y=1*3=3 and we return y as value.

b.

The value of this L3 program is the number 15 because in Let* expression evaluation process we compute the value of Val and then we bind the value to Val to the Var for each "binding" sequentially. So, by the time we bind a value to y, x is already associated with 5. Y=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.

```
(let

        ((x 1))

        (let

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

                (f x)

        )

)
```

e.

```
(define x 2)

(define y 5)

((lambda (x) ((lambda (m f) (f m))

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

) 1)
```

```scheme
; Signature: make-ok(val)
; Type: [T -> Pair(String * T)]
; Purpose: return a result of type ok with the value of val
; Pre-conditions: true
; Tests: (make-ok 5) => '(ok . 5)

; Signature: make-error(msg)
; Type: [String -> Pair(String * String)]
; Purpose: return a result of type error with the string of msg
; Pre-conditions: true
; Tests: (make-error 'failed) => '(error . failed)

; Signature: ok?(res)
; Type: [Pair(T1 * T2) -> Boolean]
; Purpose: find out if the result is of type ok
; Pre-conditions: res is a result pair
; Tests: (ok? (make-ok 5)) => #t  (ok? (make-error 'failed)) => #f

; Signature: error?(res)
; Type: [Pair(T1 * String) -> Boolean]
; Purpose: find out if the result is of type error
; Pre-conditions: res is a result pair
; Tests: (error? (make-ok 5)) => #f  (error? (make-error 'failed)) => #t

; Signature: result?(res)
; Type: [Pair(T1 * T2) -> Boolean]
; Purpose: find out if the result is of type ok or error
; Pre-conditions: res is a result pair
; Tests: (result? (make-ok 5)) => #t  (result? (make-error 'failed)) => #t

; Signature: result->val(res)
; Type: [Pair(T1 * T2) -> T2]
; Purpose: given a result, return it's value
; Pre-conditions: res is either ok or error
; Tests: (result->val (make-ok 5)) => 5

; Signature: bind(f)
; Type: [(T1 -> Pair(String * T2)) -> (Pair(String * T3) -> Pair(String * T4))]
; Purpose: returns a function that calls f if res is ok?
; Pre-conditions: res is of type ok
; Tests: (result->val ((bind (lambda (x) (make-ok (* x x)))) (make-ok 2))) => 4
; Signature: make-dict()
; Type: [ -> '()]
; Purpose: creates an empty list/dictionary
; Pre-conditions: true
; Tests: (make-dict) => '()

; Signature: dict?(e)
; Type: [List(Pair(T1 * T2)) -> Boolean]
; Purpose: checks if e is a dictionary
; Pre-conditions: e is a list of pairs
; Tests: (dict? (make-dict)) => #t  (dict? '(1 2)) => #f

; Signature: get(dict, k)
; Type: [List(Pair(T1 * T2)) * T3 -> T4]
; Purpose: looks for a value in the dictionary by it's key
; Pre-conditions: dict is a list of pairs, k is a possible key
; Tests: (result->val (get (result->val (put (make-dict) 3 4)) 3)) => 4

; Signature: put(dict, k, v)
; Type: [List(Pair(T1 * T2)) * T3 * T4-> T5]
; Purpose: adds the pair (k, v) to our dictionary, if the key exists -> override
; Pre-conditions: dict is a list of pairs
; Tests: (put (make-dict) 1 2) => '(ok (1 . 2))

; Signature: map-dict(dict, f)
; Type: [List(Pair(T1 * T2)) * (T2 -> T3) -> Pair(String * List(Pair(T4 * T5)))]
; Purpose: goes through the dictionary and applies f on every value. returns a result of the new dictionary.
; Pre-conditions: dict is a list of pairs, f is a function that accepts (cdr (car dict)) as a param.
; 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: map-dict(dict, pred)
; Type: [List(Pair(T1 * T2)) * (T2 -> Boolean) -> Pair(String * List(Pair(T4 * T5)))]
; Purpose: goes through the dictionary and applies f on every value, if the returned value from f is true -> add
the input to our new dictionary.
; Pre-conditions: dict is a list of pairs, pred is a function that accepts (cdr (car dict)) as a param.
; Tests: (result->val (get (result->val (filter-dict (result->val (put (result->val (put (make-dict) 2 3)) 3 4)) (lambda
(x y) (< (+ x y) 6)))) 2)) => 3
```