Question 1

**1.1**

In functional programming, as defined in class; there are no side effects and every function is an expression (result of the returned value). Therefore, in a functional program we shouldn’t have a function with multiple expressions in it, we might as well use a single function for each expression instead and since there are no side effects the evaluation of the expression besides the one that is returned from the function are meaningless. It might be useful for procedural and imperative languages because each function is a sequence of commands which might have to evaluate expressions and create side-effects to complete a task.

**1.2**

a.

Each special form might be evaluated differently, which allows behavior that wouldn’t be possible otherwise. Let’s say we need to evaluate the expression

(**if** (**eq?** x 0) 0 x/2)

This is possible because the special form ‘if’ evaluates the ‘else’ statement only when x is **not** 0. Without special forms this expression would result in error (division by 0)

b.

logical ‘or’ can be defined as a primitive operator because it is assembled from primitive operators (each evaluated left to right) but it will be wasteful since it is suffices that only one of the expression in the ‘or’ will be evaluated as true. Accounting for shortcut semantics, we can optimize the evaluation process by terminating the evaluation process once we found ‘true’ statement. We can see that ‘or’ operator in TypeScript is following the shortcut semantics principles by executing the following code:

**const** throwsError = () : boolean =>{  
 **throw** **new** Error("this line was executed!");  
 **return** false;  
}  
console.log( (true || throwsError()) ); // true

the fact that the execution of this code resulted in true without throwing error proves that the ‘throwsError’ function was never executed.

**1.3**

syntactic abbreviation is an operation which can be equivalently defined with the language primitives, meaning it does not require a new computation rule. It results in an easier way to write more readable code.

In L3 we have the abbreviations ‘let’ and ‘cond’:

(**let** ( (**<var1>** <exp1>)  
 (**<var2>** <exp2>)  
 ...  
 (**<varn>** <expn>) )  
 <body>)

( (**lambda** (**<var1>** ... <varn>) <body>)  
 <exp1> ... <expn> )

(**cond** (**<test1>** <then1>)  
 (**<test2>** <then2>)  
 (**else** <then>))

(**if** (**<test1>**) (**<then1>**)  
 (**if** (**<test2>**) (**<then2>**)  
 (**<then>**)))

**1.4**

a.

The value will be ; first line binds to the value. Second and third line binds to and to but since the values are computed before any bindings, is still 1. Therefore, is

b.

The value will be ; The only difference from a is the usage of let\* instead of let, hence this time the bindings are performed sequentially. By the time the expression is evaluated, is already, so

c.

(define x 2)

(define y 5)

(let

((x 1)

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

(f x)) [x 1 0] [y 1 1] [z 1 0]

(let\*

((x 1)

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

(f x))

)

d.

(let ((x 1))

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

(f x)

)

)

e.

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

**Contracts:**

; Signature: make-ok(val)

; Type: T -> pair ("Ok" T)

; Purpose: return Ok Result wrapper for given val

; Pre-conditions: none

; Tests: (ok? (make-ok "test")) ; #t

; Signature: make-ok(msg)

; Type: string -> pair ("Failure" string )

; Purpose: return error Result wrapper with given msg

; Pre-conditions: none

; Tests: (error? (make-error "test")) ; #t

; Signature: ok?(result)

; Type: Result -> boolean

; Purpose: return true if given Result is ok

; Pre-conditions: none

; Tests: (ok? (make-ok "test")) ; #t

; Signature: error?(result)

; Type: Result -> boolean

; Purpose: return true if given Result is Fauilure

; Pre-conditions: none

; Tests: (error? (make-error "test")) ; #t

; Signature: result?(result)

; Type: any -> boolean

; Purpose: return true if given argument is result (Ok or Error)

; Pre-conditions: none

; Tests: (result? (make-error "test")) ; #t

; Signature: result->val(result)

; Type: result<T> -> T

; Purpose: return the value encapsulated in result

; Pre-conditions: (result? result) ; #t

; Tests: (result->val (inverse-square-inverse 2)) ; 4

; Signature: bind(f)

; Type: (function<U> -> result<U>) -> ((function(result<T>) -> result<T2>)

; Purpose: return function based ont the given function that maps result to result

; Pre-conditions: f: <T> -> result

; Tests: (res->val ((bind make-ok) 4)) ; 4

; Signature: make-dict()

; Type: () -> list

; Purpose: return an empty dictionary

; Pre-conditions: none

; Tests: (dict? make-dict) ; #t

; Signature: dict?()

; Type: T -> boolean

; Purpose: return true if T is dictionary

; Pre-conditions: none

; Tests: (dict? make-dict) ; #t

; Signature: get?(dict k)

; Type: (dict, T) -> result<U>

; Purpose: return an error result in cast the given key is not defined in the dictionary, else returns Ok of value assigned to tje given key

; Pre-conditions: none

; Tests: (get (put (make-dict) 4 2 ) 4) ; 2

; Signature: put(dict k v)

; Type: (dict, T, U) -> result<dictionary>

; Purpose: return dictionary with the given value assigned to given key

; Pre-conditions: none

; Tests: (get (put (make-dict) 4 2 ) 4) ; 2

; Signature: remove(dict k )

; Type: (dict, T, U) -> dictionary

; Purpose: return dictionary without the given key

; Pre-conditions: (dict? dict) ; #t

; Tests: (eq? (result->val (remove (put (make-dict) 4 2 ) 4)) (make-dict) ;#t

; Signature: map-dict(dict f )

; Type: (dict, function<T> -> U) -> result<dictionary>

; Purpose: return result of dictionary with f(value) as values

; Pre-conditions: none

; 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-non-ok(dict f )

; Type: (dict, function<T> -> U) -> dictionary

; Purpose: return dictionary with f(value) as values

; Pre-conditions: (dict? dict) ; #t

; Tests: (get (result->val (map-dict-non-ok (result->val (put (result->val (put (make-dict) 1 #t)) 2 #f)) (lambda (x) (not x )))) 1)) ; #f

; Signature: filter-dict(dict f )

; Type: (dict, function<T> -> boolean) -> result<dictionary>

; Purpose: return result of dictionary only with values such that that f(value) == #t

; Pre-conditions: none

; 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)) ; Ok<3>

; Signature: filter-remove(dict f )

; Type: (dict, function<T> -> boolean) -> result<dictionary>

; Purpose: return dictionary only with values such that that f(value) == #t

; Pre-conditions: (dict? dict) ;t

; Tests: (get (result->val (filter-remove (result->val (put (result->val (put (make-dict) 2 3)) 3 4)) (lambda (x y) (< (+ x y) 6)))) 2) ; 3