Assignment 2

1. A function-body with multiple expressions is allowed but not required. If there are multiple expressions in function only the value of the last expression is returned, the other expressions will be evaluated only for some side effect. It is useful in imperative or procedural languages.
2.
   a. The difference between special form and primitive operator is the a primitive operator has all its arguments evaluated before its applied. We cannot define them as a primitive operators because in primitive operator we evaluate all the variables in the expression, while in a special operator we won't necessary would like to evaluate all the variables. For exe: in an if exp, we would like to evaluate the 'test' and only according to the outcome we evaluate the 'then' or the 'alt'.
   b. The 'or' operation can be defined as a primitive operator. Although referring to shortcut semantics we won't necessarily would like to evaluate all the given conditions of the 'or' , only some of them can be sufficient.
3. A syntactic abbreviation is a special from that can be defined with the primitives that already exists in the language . for example: let, list.
   a. <u>Let:</u>

```
  (
    (v (/ (sqrt 2) (sqrt 3)))
  )
  (+ v v)
)
```

<u>Equals to:</u>

```
(
  (lambda (v) (+ v v))
   (/ (sqrt 2) (sqrt 3))
)
```

   b. list: (list 1 2 3)

   <u>equals to</u> : (cons 1(cons 2 3))

4.
   a. The value of the program is 3. In let the initial values are computed before any of the variables become bound ,so in the process of biding y, (* x 3) = 3, because the local x haven't been bound yet. so in the binding of y, x=1.
   b. The value of the program is 15. The bindings in let* are performed sequentially from left to right, Thus the second binding is

done in an environment in which the first binding is visible, so in the process of biding y, (* x 3) = 15, because x=5 is visible to to y.

c.

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

d.

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

e.

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

Function contracts:

make-ok:

Signature: make-ok(val)

Purpose: gets a value and encapsulates it as an ok structure of type result returned pair

Type: Any -> Ok

Pre-conditions:

Test: (make-ok 3) => ("ok". 3)

make-error:

Signature: make-error(msg)

Purpose: - gets an error string and encapsulates it as an ‘error’ structure of type

result

Type: String -> EROOR

Pre-conditions:

Test: (make-error "this is an error msg") => ("error". "this is an error msg")

ok?:

Signature: ok?(x)

Purpose: To find if x is an object of type Ok

Type: Any -> Boolean

Pre-conditions:

Test: (ok? (make-ok 3)) => #t

error?:

Signature: error?(x)

Purpose: To find if x is an object of type Error

Type: Any -> Boolean

Pre-conditions:

Test: (error? (make-error "this is an error msg")) => #t result?:

Signature: result?(x)

Purpose: To find if x is an object of type Result(Ok|Error)

Type: Any -> Boolean

Pre-conditions:

Test: (result? (make-error "this is an error msg")) => #t

        (result? (make-ok 3)) => #t

result->val:

signature: result->val(res)

Purpose: returns the encapsulated value of a given result: value for ok result,

and the error string for error result

Type: Result -> Any

Pre-conditions: (result? res) => #t

Test: (result->val(make-ok 3)) => 3

bind:

signature: bind(f, res)

Purpose: Apply the function f on the value of res if res is an Ok, otherwise do nothing

Type: (Lambda, Result) -> Any || Error

Pre-conditions: (result? res) => #t, f.Type = typeof(res.val) -> Any

Test: ((bind (lambda (x) (* x x))) (make-ok 3)) => 9

make-dict:

signature: make-dict()

Purpose: Create new empty dictionary

Type: Any -> Dict

Pre-conditions:

Test: (make-dict) => (("dict" . "dict"))

dict?

signature: dict?(dict)

Purpose: Find if dict is an object of type Dict

Type: Any -> Boolean

Pre-conditions:

Test: (dict? (make-dict)) => #t

put:

signature: put(dict, k, v)

Purpose: Add the pair (k, v) to the dictionary dict or change the value of the key k to v if k allready in dict

Type:(Dict, Any, Any) -> Ok<Dict> || Error<"Error: not a dictionary">

Pre-conditions: (dict? dict) => #t

Test:(put (make-dict) "grade" 100) => "(ok" . (("dict" . dict)("grade" . 100)))

get:

signature: get(dict, k)

Purpose: 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 wii be returned

Type: (Dict, Any) -> Ok<Any> || Ok<"Key not found"> || Error<"Error: not a dictionary">

Pre-conditions: (dict? dict) => #t

Test: (get (result->val(put (make-dict) "grade" 100)) "grade") => ("ok" . 100) map-dict:

signature: map-dict(dict, f)

Purpose: Apply the function f to any value v in dictionary dict and return a result with the new dict with the same keys and the new values.

Type: (Dict, Lambda) -> Ok<Dict> || Error<"Error: not a dictionary">

Pre-conditions: (dict? dict) => #t, f.Type == typeof((get dict k)) for all k in dict

Test: (map-dict (result->val (put(result->val (put (make-dict) "a" 2)) "b" 4)) (lambda (x) (* x x))) => ("ok" ("dict" . "dict") ("a" . 4) ("b" . 16))

filter-dict:

signature: filter-dict(dict, pred)

Purpose: Returns a result with a new dictionary from dict with the pairs (k v) that satisfies pred.

Type: (Dict, Lambda) -> Ok<Dict> || Error<"Error: not a dictionary">

Pre-conditions: (dict? dict) => #t, pred.Type == (typeof(k), typeof(v)) -> Boolean for all (k v) in dict

Test: (filter-dict (result->val (put(result->val (put (make-dict) "a" 2)) "b" 4)) (lambda (x y) (> y 3)))=> ("ok" ("dict" . "dict") ("b" . 4))