part 1

1.1) A function-body with multiple expressions is **not** necessarily required in pure functional programming. Of the languages we've learned so far (L1-L3), it is useful in languages L2 and L3.

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) 'or' can be defined as primitive operator by calculating each term and operating the 'or' operator between each couple. A more efficient way is to define 'or' as a special form, one that calculates one condition at a time and returns True for the first true condition, if found.

1.3) Syntactic abbreviation is a shortcut that was meant for easing programmers by defining an easier way of writing common expressions.

Examples: a) in 'NumberLink', as shown in practical session 2 : <next>? = <type> instead of <next> = type | undefined.

b) using the 'let' exp instead of using the 'lambda' exp.

1.4) a) value is 3. Because the values in the bindings are evaluated in the current environment.

b) value is 15. Because but the bindings are performed sequentially from left to right.

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

Question 2 – Contract:

make-ok:

Signature: make-ok(val)
Type: (T → pair('ok, T))
Purpose: create a pair, it's "car" field is 'ok, while it's "cdr" is a given value
Test: (make-ok 1) → '('ok . 3)

make-error:

Signature: make-error(val)
Type: (T → pair('error, T))
Purpose: create a pair, it's "car" field is 'error, while it's "cdr" is a given value
Test: (make-error 1) → '('error . 3)

ok?:

Signature: ok?(res)
Type: (T → (#t | #f))
Purpose: checks whether res is of an 'ok' object or not
Test: (ok? (make-ok 1)) → #t

error?:

Signature: error?(res)
Type: (T → (#t | #f))
Purpose: checks whether res is of an 'error' object or not
Test: (error? (make-error 1)) → #t

result?:

Signature: result?(res)
Type: (T → (#t | #f))
Purpose: checks whether res is of an ('error | 'ok) object or not
Test: (result? (make-error 1)) → #t

result->val:

Signature: result->val (res)
Type: (T1 → T2)
Purpose: returns the value of a Result type object, or Error

otherwise.

Test: (result->val (make-ok 1)) → 1

bind:

Signature: bind (f)

Type: (Function → (#t | #f))

pre-conditions: f is a function

Purpose: Create a function that gets a Result-type 'res' and either executes the original function-param f with res value, if the Res-param is of an ok type, or return res as is otherwise

Test: *assuming* half is: (lambda (a) (/ a 2)),
(bind half) → (lambda (res) (make-ok (/ res 2))

make-dict:

Signature: make-dict()

Type: (none → dict)

Purpose: create an empty list

Test: (list? (make-dict)) → #t

dict?:

Signature: dict?(lst)

Type: (T → (#t | #f))

Purpose: checks whether lst is of an dict Type object or not

Test: (dict? (make-dict)) → #t

get:

Signature: get(dict key)

Type: (T1 T2 → result)

Purpose: returns the value of a key – wrapped inside an ok, if found in dict, or error otherwise

Test: (get (result->val (put (make dict) 1 10)) 1) → '('ok . 10)

put:

Signature: put (dict key val)

Type: (T1 T2 T3→ result)

Purpose: returns dict with a new (k,v) pair wrapped inside an ok, if dict is of an Dict-Type, or error otherwise. Exchange the original

value for a new value for an existing key in dict.
Test: (get (result->val (put (make dict) 1 10)) 1) → '('ok . 10)

map-dict:

Signature: map-dict(dict f)
Type: (T Function→ result)
Pre-condition: f is of a Function-Type.
Purpose: returns a new dictionary-type object, s.t his keys are the keys of the original dict, and their values are return value of executing the function-type f using the original values.
Test: (map-dict(
(result->val (put(
result->val (put(
make dict) 1 10)) 2 20)) ,
(lambda (x) (/ x 2)))
→ '('(1 . 5) . '('(2 . 10) . '()))

filter-dict:

Signature: map-dict(dict f)
Type: (T Function→ T)
Pre-condition: f is of a boolean Function-Type.
Purpose: returns a new dictionary-type object, s.t his keys are the keys of the original dict which their values return #t when executing the function-type f using each key and value.
Test: (filter-dict(
(result->val (put(
result->val (put(
make dict) 1 5)) 2 3)) ,
(lambda (k v) (and (even? k) (odd? v))))
→ '('(2 . 3) . '())) *(odd?, even? are pre-defined)*