Q1:

1.1: No a function-body with multiple expressions isn't a requirement in a purely functional programming language. That's because in such a language a function can only return a value and can't change the global state. And any setting of the inner state of a function can be rewritten into a single expression.

It would be useful in languages where it's possible to change the state of the program (e.g., Object Oriented, Procedural, etc.)

1.2: **a)** Special forms are needed to extend functionality of the language to beyond what pure functions are capable of doing (e.g., execution flow control, defining a new function, defining a variable.)

We can't simply define them as primitive operators because they can have operands that don't need to be evaluated, as opposed to primitive operators, which evaluate all of their arguments (e.g., the name of the variable declared with define, or the parameter names in the lambda function.)

b) No. The logical operation "or" implements the shortcut semantics principle, meaning that it evaluates the given operands until one of them results in "#t" value or until all of them evaluate to "#f". This functionality isn't possible with primitive operators, which have to evaluate all of their operands.

1.3: Syntactic abbreviation is creation of new, more convenient and readable, syntax for already existing functionality in the language. Examples: a) the "let" structure as a syntactic abbreviation of an application of a lambda function; b) the "cdddr" abbreviation of 3 consecutive "cdr" commands.

1.4: **a)** The value is 3. At the moment of evaluation of the value of "y", the "x" that gets defined inside the "let" structure isn't appearing yet in the bindings of the closure, since "let" first evaluates all of its declarations, and only then updates them in the bindings. So the value of "y" is 1*3=3.

b) The value is 15. The construct "let*" evaluates and binds it's declarations sequentially, meaning that at the moment of evaluation of "y", the "x" it refers to is the one declared a line prior to it. So the value of "y" is 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)

```
(define x 2)
(define y 5)

(
    let
    ( ;variables of the first "let"
        (x 1)
    )

    ( ;return value expression of the first "let"
        let
        ( ;variables of the second "let"
            (f (lambda (z) (+ x y z)))
        )

        (f x) ;return value expression of the second "let"
    )
)
```

e)

```
(define x 2)
(define y 5)

(
    (lambda (x)
            (
                (lambda (f)
                    (f x)   ;the return value of the whole thing
                )

                (lambda (z) (+ x y z)) ;defining the "f" var
            )
    )

    1 ;defining the "x" var
)
```

Q2: Contracts of implemented functions:

2.1: a)

```
;; Signature: make-ok(val)
;; Type: [T1 -> OK(T1)]
;; Purpose: construct an OK Result object containing the value val
;; Pre-conditions: None
;; Tests:   (make-ok 3) -> '(ok . 3);
;;                (make-ok 3.5) -> '(ok . 3.5)


;; Signature: make-error(msg)
;; Type: [String -> Error]
;; Purpose: construct an Error Result object containing the error message
;; Pre-conditions: Message is a string
;; Tests: (make-error "abc") -> '(error . "abc")

;; Signature: ok?(res)
;; Type: [T1 -> Boolean]
;; Purpose: determine if the given object is of type OK
;; Pre-conditions: None
;; Tests:   (ok? (make-ok 1)) -> #t;
;;                (ok? (make-error "1")) -> #f;
;;                (ok? 123) -> #f;

;; Signature: error?(res)
;; Type: [T1 -> Boolean]
;; Purpose: determine if the given object is of type Error
;; Pre-conditions: None
;; Tests:   (error? (make-ok 1)) -> #f;
;;                (error? (make-error "1")) -> #t;
;;                (error? 123) -> #f;
```

```
;; Signature: result?(res)
;; Type: [T1 -> Boolean]
;; Purpose: determine if the given object is of type Result
;; Pre-conditions: None
;; Tests:   (result? (make-ok 1)) -> #t;
;;                (result? (make-error "1")) -> #t;
;;                (result? 123) -> #f;



;; Signature: result->val(res)
;; Type: [Result -> T1]
;; Purpose: Return the value or error message of the given Result object
;; Pre-conditions: res is of type Result
;; Tests:   (result->val (make-ok 1)) -> 1;
;;                (result->val (make-error "1")) -> "1";
```

b)

```
;; Signature: bind(f)
;; Type: [(T1->Result) -> (Result->Result)]
;; Purpose: return a function that gets a Result object res and if res is of
type OK, returns f(res.val), and
;;          otherwise – an Error
;; Pre-conditions: f is a function that returns a Result
```

2.2:

```
;; Signature: make-dict()

;; Type: [Void -> Dictionary]

;; Purpose: construct an empty dictionary

;; Pre-conditions: None

;; Tests: (make-dict) -> '();



;; Signature: dict?(e)

;; Type: [T1 -> Boolean]

;; Purpose: type predicate for dictionaries

;; Pre-conditions: None

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

;;                  (dict? 123) -> #f;



;; Signature: put(dict k v)

;; Type: [T1*T2*T3 -> Result]

;; Purpose: if dict is a dictionary - returns an OK of it with the value v associated
with key k, otherwise – returns an Error

;; Pre-conditions: None

;; Tests:    (put (result->val (put (make-dict) 1 2)) 2 3) -> '(ok (1 . 2) (2 . 3));

;;                  (put 1 1 2) -> '(error . "Error: not a dictionary");



;; Signature: get(dict k)

;; Type: [T1*T2 -> Result]

;; Purpose: if dict is a Dictionary and the key k is associated with a value in it -
return an OK of the value,

;;           otherwise - return an Error

;; Pre-conditions: None

;; Tests:    (get (result->val (put (make-dict) "1" 2)) "1") -> '(ok . 2);

;;                  (get (result->val (put (make-dict) 1 2)) "1") -> '(error . "Key
not found");

;;                  (get 123 123) -> '(error . "Error: not a dictionary");
```

```
;; Signature: put-iter(dict k v)

;; Type: [Dictionary*T1*T2 -> Dictionary]

;; Purpose: return the given dictionary dict with the value v associated with key k

;; Pre-conditions: dict is a dictionary

;; Tests:    (put-iter (make-dict) 1 2) -> '((1 . 2));

;;                  (put-iter (put-iter (make-dict) 1 2) 2 3) -> '((1 . 2) (2 . 3));



;; Signature: get-iter(dict k)

;; Type: [Disctionary*T1 -> Result]

;; Purpose: if the key k is associated with a value in dict - return an OK of the
value,

;;           otherwise - return an Error

;; Pre-conditions: dict is a Dictionary

;; Tests:    (get-iter (put-iter (make-dict) "1" 2) 1) -> '(ok . 2);

;;                  (get-iter (put-iter (make-dict) 1 2) 3) -> '(error . "Key not
found");



;; Signature: caar(lst)

;; Type: [List(Pair(T1,T2)) -> T1]

;; Purpose: return the first element of the first pair in lst

;; Pre-conditions: lst is a List of Pairs

;; Tests: (caar '((1 . 2) (2 . 3))) -> 1;



;; Signature: cadr(lst)

;; Type: [List(Pair(T1,T2)) -> T2]

;; Purpose: return the second element of the first pair in lst

;; Pre-conditions: lst is a List of Pairs

;; Tests: (cadr '((1 . 2) (2 . 3))) -> 2;
```

2.3: a)

```
;; Signature: map-dict(dict f)

;; Type: [T1*(T2->T3) -> Result]

;; Purpose: if dict is a Dictionary return an OK with it with it's values
updated to the results of running f on them,

;;                otherwise - return an Error

;; Pre-conditions: f is an unary function

;; Tests:   (map-dict (result->val (put (make-dict) "abc" #t)) (lambda (x)
(not x ))) -> '(ok ("abc" . #f));

;;                (map-dict 123 (lambda (x) (not x ))) -> '(error . "Error:
not a dictionary");


;; Signature: map-dict-iter(dict f)

;; Type: [Dictionary(Any,T1)*(T1->T2) -> Dictionary(Any,T2)]

;; Purpose: return dict with it's values updated to the results of running f
on them

;; Pre-conditions: dict is a Dictionary, f is an unary funciton that's
applicable to the values of dict

;; Tests: (map-dict-iter (result->val (put (make-dict) "abc" #t)) (lambda (x)
(not x ))) -> '(("abc" . #f));
```

b)

```
;; Signature: filter-dict(dict pred)

;; Type: [T3*(T1*T2->Boolean) -> Result]

;; Purpose: if dict is a Dictionaary - return an OK of a Dictionary containing only
the (key, value) pairs

;;           from dict for which f(key, value) returns #t, otherwise - return an Error

;; Pre-conditions: pred is a function taking (key, value) pairs

;; Tests:    (define even-key (lambda (k v) (even? k)));

;;                (filter-dict (result->val (put (make-dict) 2 3)) even-key) -> '(ok
(2 . 3));

;;                (filter-dict (result->val (put (make-dict) 1 3)) even-key) ->
'(ok);

;;                (filter-dict 123 even-key) -> '(error . "Error: not a
dictionary");
```

```
;; Signature: filter-dict-iter(dict pred)

;; Type: [Dictionary(T1,T2)*(T1*T2->Boolean) -> Dictionary(T1,T2)]

;; Purpose: return a Dictionary containing only the (key, value) pairs for
which f(key, value) returns #t

;; Pre-conditions: dict is a Dictionary, pred is a function taking (key,
value) pairs, type-compatible

;;          with the (key, value) pairs in dict

;; Tests:   (define even-key (lambda (k v) (even? k)));

;;                (filter-dict-iter (result->val (put (make-dict) 2 3)) even-
key) -> '((2 . 3));

;;                (filter-dict-iter (result->val (put (make-dict) 1 3)) even-
key) -> '();
```