שאלה 1 – תיאורטיות

1.1

תשובה: בתכנות פונקציונלי טהור אין הכרח בריבוי ביטויים בגוף פונקציה. זאת מכיוון שערך פונקציה הוא מה שמעניין אותנו, ותמיד ערך הפונקציה הוא ערך הביטוי האחרון שגוף הפונקציה. בנוסף, אין צורך בביטויים נוספים לפני, כיוון שבכל מקרה בתכנות פונקציונלי נמנעים ממוטציות ותופעות לוואי למשתנים ומבנים בסביבה. ריבוי ביטויים בגוף פונקציה זה שימושי בשפות שאינן פונקציונליות טהורות.

1.2

a. ללא צורות מיוחדות, שיטת החישוב היחידה של ביטויים היא הדיפולטיבית של הפעלת אופרטור על אופרנדים. שיטות מיוחדות כמו if לדוגמה, מהווה מבנה בקרה אשר לא ניתן לממש כאופרטור פרימיטיבי, משום שחישוב של ביטוי if עשוי להוביל לשגיאה ע"פ השיטה הדיפולטיבית, למשל בקוד הבא:

```
(define inverse (lambda (x)
 (if (= x 0)
      (make-error "div by 0")
      (make-ok (/ 1 x)))))
```

.b

אם רוצים להגדיר or מבלי להשתמש ב-shortcut semantics, אז מספיק להשתמש באופרטור פרימיטיבי, ואז כל הערכים של האופרנדים מחושבים קודם, ואז האופרטור or מופעל עליהם.

אם רוצים להגדיר or תוך שימוש ב-shortcut semantics, חייב להשתמש בצורה מיוחדת, כדי לאפשר עצירה ברגע שנתקלנו בביטוי שערכו true.

1.3

סוכר תחבירי – מבנה תחבירי בשפה אשר מבטא פעולה של תחביר קיים בשפה בצורה יותר קריאה וברורה עבור המתכנת.

דוגמאות:

let - מהווה סוכר תחבירי מקביל לlambda. הצורות הבאות שקולות.

```
(let ((a 2)
      (b 3))
  (+ a b))


((lambda (a b)
  (+ a b))
  2 3)
```

caar – מהווה סוכר תחבירי מקביל לשימוש חוזר ב-car. הצורות הבאות שקולות.

```
(define sugar (list '(1 2) '(3 4) '(5 6)))
(car (car sugar))
(caar sugar)
```

1.4

.a

הערך המוחזר הוא 3.

הסבר: let עובד כך שהביינדינג בין המשתנים לערכים בבלוק ה-let נעשה **לאחר** חישוב הערכים. כלומר, הערכים 5 ו-$(* \ x \ 3)$ מחושבים לפני ההשמה. $x = 1$ ברגע שהחישוב הזה מבוצע (בגלל ה-define), אז y מקבל השמה לערך 3 המתקבל מחישוב זה.

.b

הערך המוחזר הוא 15.

הסבר: *let עובד כך שהביינדינג בין המשתנים לערכים נעשית בסדרתיות משמאל לימין, בצורה כזו שההשמה הראשונה תהיה visible ברגע שההשמה השנייה מתבצעת. לכן קודם כל נקבל $x = 5$, ואז $y = (* \ x \ 3) \ = \ (* \ 5 \ 3) \ = \ 15$.

.c

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

(let
    ((x 1)
     (f (lambda (z) ([+ free] [x : free] [y : free] [z : 0 0]))))
  ([f : 0 1] [x : 0 0]))

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

.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)
```

שאלה 2 – DBC

make-ok

Signature: (make-ok item)

Type: [T -> Result<T>]

Purpose: encapsulate a given value within an ok structure of type Result

Pre-conditions: none

Tests:

```
> (define x (make-ok 5))
> x
'("ok" . 5)
```

make-error

Signature: make-error(message)

Type: [T -> Result<T>]

Purpose: encapsulate a given string message within an error structure of type Result

Pre-conditions: none

Tests:

```
> (define x (make-error "failure"))
> x
'("error" . "failure")
```

?ok

Signature: ok? x

Type: [T -> boolean]

Purpose: type predicate for ok

Pre-conditions: none

Tests:

(ok? (make-ok 5)) --> #t ;

(ok? (make-error "fail") -> #f

error?

Signature: error? x

Type: [T -> boolean]

Purpose: type predicate for error

Pre-conditions: none

Tests:

(error? (make-ok 5)) --> #f ;

(error? (make-error "fail")) --> #t

?result

Signature: result? x

Type: [T -> boolean]

Purpose: type predicate for result

Pre-conditions: none

Tests:

(result? (make-ok 5)) --> #t ;

(result? "some string") --> #f

result->val

```
; Signature: (result->val res)
; Type: (result<T>) -> T
; Purpose: returns the encapsulated value of a given result: value for ok result, and the error string for error result.
; Pre-conditions: res is of type Result<T>
; Tests: (define r1 (make-ok 5))
         (eq? (result->val r1) 5)
         -> #t
```

bind

```
; Signature: (bind f)
; Type: [(T1 -> Result<T2>) -> (Result<T1> -> Result<T2>)]
; Purpose: gets a function from non-result parameter to result and returns this function from result to result.
; Pre-conditions: none
; Tests:
(define pipe
    (lambda (fs)
        (if (empty? fs)
            (lambda (x) x)
            (compose (pipe (cdr fs)) (car fs)))))


(define square (lambda (x) (make-ok (* x x))))

(define inverse (lambda (x)
    (if (= x 0)
        (make-error "div by 0")
        (make-ok (/ 1 x)))))

(define inverse-square-inverse
    (pipe (list inverse (bind square) (bind inverse))))

(result->val (inverse-square-inverse 2))
→ 4

(result->val (inverse-square-inverse 0))
→ "div by 0"
```

make-dict

```
; Signature: (make-dict)
; Type: [() -> '()]
; Purpose: returns a new empty dictionary
; Pre-conditions: none
; Tests: (define dict (make-dict))
         (eq? dict '())  --> #t
```

?dict

```
; Signature: (dict? e)
; Type: [any -> Boolean]
; Purpose: type predicate for dictionaries
; Pre-conditions: none
; Tests: (define dict (make-dict))
         (dict? dict) -> #t
         (ditc? '(4 5)) -> #f
```

put

```
; Signature: (put dict key value)
; Type: (T1 * T2 * T3 -> Result<T4> )
; Purpose: gets a dictionary, a key and a value,
           and returns a result of a dictionary with the addition of the given key-value.
           In case the given key already exists in the given dict, the returned dict should contain the new value for this key.
; Pre-conditions: none
; Tests: (define dict1 (make-dict))
         (get (result->val (put dict1 2 3), 2)) -> 3
```

get

```
; Signature: (get dict key)
; Type: (T1) -> result<T2>
; Purpose: gets a dictionary and a key, and 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 should be returned.
; Pre-conditions: none
; Tests: (result-> val (get (result->val (put dict 3 4)) 3)) -> 4
```

map-dict

```
; Signature: (map-dict dict foo)
; Type: [ Dict * (T1->T2) -> Result<Dict | String>]
; Purpose: gets a dictionary and an unary function, applies the function of the values in the dictionary, and returns a result of a new dictionary with the resulting 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
```

filter

```
; Signature: (filter dict P)
; Type: [ Dict * (T1 -> Boolean) -> Result<Dict | String>]
; Purpose: gets a dictionary and a predicate that takes (key value) as arguments,
           and returns a result of a new dictionary that contains only the key-values that satisfy the predicate.
; Pre-conditions: none
; Tests: (define even-key-odd-value? (lambda (k v) (and (even? k) (odd? v))))
         (result->val (get (result->val (filter-dict (result->val (put (result->val (put (make-dict) 2 3)) 3 4)) even-key-odd-value?)) 2))
         --> 3
```