עבודה 2 - עקרונות שפות תכנות

חלק 1

שאלה 1.1

לא, גוף פונקציה שמכילה מספר ביטויים איננה הכרחית בתכנות פונקציונאלי.
זה כיוון שבתכנות פונקציונאלי רק השורה האחרונה שנכתבת חוזרת מהפונקציה וכיוון שבשיטת תכנות זו אין side affects אז ניתן לצמצם מספר ביטויים לכדי ביטוי יחיד.
להשלים

שאלה 1.2

א) צורות מיוחדות הכרחיות בשפות תכנות כיון שבהפעלת פרוצדורות מסוימות אנו רוצים לעשות אבלואציה לביטוי לפי כללים יחודיים. למשל בצורה המיוחדת if, הפרוצדורה כתובה בפורמט:

if (test) (then) (else)

צורה זו מאפשרת לנו לבצע אבלואציה של הביטוי בהתאם לקיום ביטוי הtest, כלומר האם לבצע את הthen או את הelse

ב) האופרטור or הינו מהצורה or conditionA conditionB
כאשר הצורה המיוחדת or מממשת מתודה הנקראת 'some' הממשת קונספט שנקרא 'shortcut semantics'. המשמעות של 'some' הוא שהאופרטור לא בודק את conditionB במידה ו-conditionA מתקיים.
ניתן להגדיר את האופרטור or בתור אופרטור פרימיטיבי, אך במימוש זה נאלץ לבדוק בכל מקרה את תנאי ב׳, למרות שלעתים אין בכך צורך.

שאלה 1.3

קיצור תחבירי הינו צורה שקולה לכתיבה של ביטוי, אך בצורה מקוצרת.
קיצורים תחביריים נועדו בכדי לצמצם את אורך הקוד, לשפר את יכולת ההבנה של הקוד ע״י שיפור הקריאות ולמנוע חזרתיות. שתי דוגמאות לקיצורים תחביריים הינם:

1. הצורה let היא צורה מקוצרת להפעלה של lambda, למשל הביטויים הבאים שקולים:

```
(let ((x 1)) (+ x 1))
```

```
((lambda(x) (+ x 1)) 1)
```

2. הצורה cond היא צורה מקוצרת לביטוי if אשר שימושית בעיקר כאשר ביטוי הif מכיל מס׳ תנאים, למשל הביטויים הבאים שקולים:

```
(define x 2)
(cond ((= x 0) 0)
      ((= x 1) 1)
      (else 2))
```

```
(define x 2)
(if (= x 0) 0
    (if (= x 1) 1
        2))
```

עבודה 2 - עקרונות שפות תכנות

חלק 1

<u>שאלה 1.4</u>

א) הערך המתקבל מהתוכנית הוא 3, הסבר:
בשלב בו מתבצעת ההשמה y = x*3 הערך הנתון של המשתנה x הינו 1. זה קורה כיוון שבשימוש בצורה let הbinding הנעשה במסגרת הסוגריים הראשוניים גלויים רק בגוף הlet (הסוגריים הבאים), כלומר ההשמה של x=5 רלוונטית רק בגוף הlet.

ב) הערך המתקבל מהתוכנית הוא 15, הסבר:
הצורה let* דומה לצורה let, אולם בצורה זו הbinding הנעשה במסגרת הסוגריים הראשונים נעשה במקביל, כלומר בשלב בו מתבצעת ההשמה y = x*3 הערך הנתון של x הינו 5.

ג)

```
(define x 2) ;[x: 0 0]
(define y 5) ;[y: 0 1]

(let
((x 1) ;[x: 0 0]
 (f (lambda (z) (+ x y z)))) ;[x: 2 0], [y: 2 1], [z: 0 0]
  (f x)) ;[x: 1 0]

(let*
    ((x 1) ;[x: 0 0]
     (f (lambda (z) (+ x y z)))) ;[x: 1 0], [y: 2 1], [z: 0 0]
  (f x)) ;[x: 1 0]
```

ד)

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

ה)

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

עבודה 2 - עקרונות שפות תכנות

חלק 1

חוזים

; Signature: caar(p) ; Type: pair->object ; Purpose: return the left most element in (car p) ; Pre-conditions: (car p) is a pair ; Tests: (caar ((1 . 2) (3 . 4))) 1	; Signature: cadr(p) ; Type: pair->object ; Purpose: return the left most element in (cdr p) ; Pre-conditions: (cdr p) is a pair ; Tests: (cadr ((1 . 2) (3 . 4))) 3
; Signature: cdar(p) ; Type: pair->any ; Purpose: return the right most element in (car p) ; Pre-conditions: (car p) is a pair ; Tests: (cadr ((1 . 2) (3 . 4))) 2	; Signature: cddr(p) ; Type: pair->any ; Purpose: return the right most element in (cdr p) ; Pre-conditions: (cdr p) is a pair ; Tests: (cddr ((1 . 2) (3 . 4))) 4
; Signature: (make-empty-list) ; Type: ()->'() ; Purpose: return an empty list ; Pre-conditions: None ; Tests: (make-empty-list) '()	; Signature: make-ok(val) ; Type: any -> <'ok,any> ; Purpose: return an implementation of an ok object that contains val ; Pre-conditions: None ; Tests: (make-ok 3) ('ok . 3)
; Signature: make-error(msg) ; Type: string -> <'error,string> ; Purpose: return an implementation of an error object that contains msg ; Pre-conditions: None ; Tests: (make-error "err") ('error . "err")	; Signature: ok?(x) ; Type: any -> boolean ; Purpose: check whether a given object is an ok object ; Pre-conditions: None ; Tests: (ok? ('ok . 3)) #t
; Signature: error?(x) ; Type: any -> boolean ; Purpose: check whether a given object is an error object ; Pre-conditions: None ; Tests: (error? ('ok . 3)) #f	; Signature: result?(x) ; Type: any -> boolean ; Purpose: check whether a given object is a result object(ok or error) ; Pre-conditions: None ; Tests: (result? ('error . "err")) #t
; Signature: result->val(x) ; Type: result->any ; Purpose: return the nested val/msg in x	; Signature: bind(f) ; Type: (T->ok) -> (result->result) ; Purpose: gets a function from non-result to result ans returns this

עבודה 2 - עקרונות שפות תכנות

חלק 1

; Pre-conditions: x is a result object ; Tests: (result->val ('ok . 3)) 3	function from result to result ; Pre-conditions: T is a non result object ; Tests: (define g (bind((lambda(x) (make-ok x*x))))) (g ('ok . 3)) ('ok . 9)
; Signature: make-dict() ; Type: () -> <'dict,'()> ; Purpose: return an implementation of a dictionary ; Pre-conditions: None ; Tests: (make-dict) ('dict . '())	; Signature: dict?(x) ; Type: any -> boolean ; Purpose: check whether a given object is a dict object ; Pre-conditions: None ; Tests: (dict? ('dict . '())) #t
; Signature: get(d,k) ; Type: dict*any -> result ; Purpose: get the 'val' assigned to the 'key' k in d ; Pre-conditions: None ; Tests: (get ('dict (3 . 4) (5 . 6) 5) ('ok . 6)	; Signature: getRec(d,k) ; Type: pair*any -> result ; Purpose: get the 'val' assigned to the 'key' k in d. This function is called from get without the 'dict tag ; Pre-conditions: None ; Tests: (getRec ((3 . 4) (5 . 6) 5) ('ok . 6)
; Signature: put(d k v) ; Type: dict*any*any -> result ; Purpose: add (k . v) to dict ; Pre-conditions: None ; Tests: (put ('dict (3 . 4)) (1 . 2)) ('ok . ('dict (1 . 2) (3 . 4)))	; Signature: putRec(d k v) ; Type: pair*any*any -> result ; Purpose: add (k . v) to d ; Pre-conditions: None ; Tests: (putRec ((3 . 4)) (1 . 2)) ('ok . ('dict (1 . 2) (3 . 4)))
; Signature: map-dict(d , f) ; Type: dict*(T1->T2) -> result ; 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: d's values are of type T1 ; Tests: (map-dict-rec ('dict (1.2)) (lambda(x) x*x)) ('ok . ('dict (1.4)))	; Signature: map-dict-rec(d , f) ; Type: pair*(T1->T2) -> pair ; Purpose: gets a pair and an unary function, applies the function of the right element in the pair, and returns the pair after applying f to the right element in each nested pair in d ; Pre-conditions: None ; Tests: (map-dict-rec ((1.2)) (lambda(x) x*x)) ((1.4))
; Signature: filter-dict(d, pred) ; Type: dict*(T->boolean) -> result ; Purpose: gets a dictionary and a predicate that takes (key value) as arguments, and returns a result of a	; Signature: filter-dict-rec(d, pred) ; Type: pair*(T->boolean) -> pair ; Purpose: an auxiliary function used by filter-dict that retruns the given pair(without the 'dict tag) after applying

עבודה 2 - עקרונות שפות תכנות

חלק 1

new dictionary that contains only the key-values that satisfy the predicate ; Pre-conditions: d's values are of type T ; Tests: (filter-dict ('dict (1.2) (4.6)) isEven?)) ('ok . ('dict (4.6)))	the pred function ; Pre-conditions: None ; Tests: (filter-dict-rec ((1.2) (4.6)) isEven?) ((4.6))