Question 1: Theoretical Questions

1.1) חישוב צורות מיוחדות שונה מחישוב אופרטורים פרימיטיביים כיוון שמחשבים את האופרנדים והאופרטורים ואחר כך מפעילים האופרטור על מה שחישבנו (האופרנדים והאופרטורים).
למשל: if מחשבים רק את שניים מ- then else test ולא את שלושתם.

1.2) לא, אין קריאות רקורסיה ב-L1 לכן ניתן להחליף כל משתנה לערך שלו.

1.3) כן, משום שיש פרוצדורות אז אפשר להשתמש בקריאות רקורסיביות ב-L2. עם משתנים מוגדרים כ- lambda אז אי אפשר לשנות var references שלהם עם הערכים המוגדרים שלהם.

1.4) יתרונות:

- PrimOp:
  1. הוא ייצוג פשוט ויעיל שניתן ליישם בקלות בחומרה או בתוכנה ברמה נמוכה.
  2. ניתן לייעל את הפעולות לביצועים מכיוון שהן מיושמות בדרך כלל כהוראות מכונה ברמה נמוכה.
  3. פעולות יכולות לשמש כאבני בניין כדי להגדיר פעולות מורכבות יותר, כגון פעולות אריתמטיות ולוגיות.,

- Closure:
  1. מאפשר ייצוג של מבני תכנות ברמה גבוהה יותר כגון פונקציות וסגירות.
  2. הוא מנגנון רב עוצמה ליצירת קוד שניתן לשימוש חוזר ולחיבור, שכן ניתן להגדיר פונקציות ולהעביר כארגומנטים לפונקציות אחרות.
  3. מספק דרך גמישה ליישום היקף דינמי והיקף מילוני, הנחוץ עבור שפות תכנות רבות.

1.5) map - ניתן להשתמש בו זמנית. יישום הפונקציה על פריט אחד מוערך ללא תלות בביצוע הפונקציה על פריט אחר.

reduce - ניתן להשתמש בו במקביל, אם תהליך ההפחתה הוא קומוטטיבי ואסוציאטיבי בהתבסס על הפחתת הפריטים הקודמים.

filter - ניתן להשתמש בו זמנית. יישום הפונקציה על פריט אחד מוערך ללא תלות בביצוע הפונקציה על פריט אחר.

all - ניתן להשתמש בו במקביל, יישום הפונקציה על פריט אחד מוערך ללא תלות בביצוע הפונקציה על פריט אחר.

compose - ניתן להשתמש בו במקביל, בתנאי שהפרוצדורה אסוציאטיבית.

1.6) lexical address הכתובת המילונית קובעת באופן חד משמעי את הצהרת המשתנה שאליה קשורה הפניה למשתנה. למשל:

$$int\ main(){\ int\ x = 0;\ printf(\%\mathrm{d}, x);\ \ return\ 0;\}$$

המשתנה x יש לו lexical address בפונקצית main, בזמן קומפילציה מקום x בזיכרון תלוי ב- lexical address שלו. בזמן ריצה ערך x יודפס למסך, אך ה- lexical address לא השתנה ונשאר קבוע.

1.7)

```
|( cond <CClause>+ <EClause> ) / cond(cclauses:CClause[], eClause:EClause)
;;<CClause> ::= (<cexp> <cexp>+) / CClause(test: CExp, then: CExp[])
;;<EClause> ::= ( <cexp> ) / EClause(then: CExp)
```

Question 2: Contracts

; Signature: take(list)
; Type: (list(any)) => list(any)
; Purpose: gets a $list$ and a number pos and returns a new list whose elements are the first pos elements of the $list$.
; Pre-conditions:

- List of any values
- pos: non negative number.

; Tests:

- (take (list 1 2 3) 2) → '(1 2)
- (take '() 2) → '()

; Signature: take-map(list, func, pos)
; Type: (list(any), (any) => any, number) => list(any)
; Purpose: gets a $list$, a function $func$ and a number pos and returns a new list whose elements are the first pos elements mapped by $func$. If the $list$ is shorter then pos- return the mapped $list$.
; Pre-conditions:

- List of any values
- Func takes a list of any values and returns a new value of any values
- Pos: non negative number

; Tests:

- (take-map (list 1 2 3) (lambda (x) (* x x)) 2) → '(1 4)
- (take-map (list 1 2 3) (lambda (x) (* x x)) 4) → '(1 4 9)

; Signature: take-filter(list, pred, pos)
; Type: (list(any), (any) => Boolean, number) => list(any)
; Purpose: - gets a $list$, a predicate $pred$ and a number pos and returns a new list whose elements are the first pos elements of the $list$ that satisfy $pred$.
; Pre-conditions:

- List of any values
- Pred a function takes element from list and returns boolean
- Pos: non negative number

; Tests:

- (take-filter (list 1 2 3 4) (lambda (x) (> x 1)) 2) → '(2 3)
- (take-filter (list 1 2 3) (lambda (x) (> x 3)) 2) → '()

; Signature: sub-size(list, size)
; Type: (list(any), number) => list(any)
; Purpose: gets a $list$ and a number $size$ and returns a new list of all the sublists of $list$ of length $size$.
; Pre-conditions:

- List of any values
- Size non negative number

; Tests:

- (sub-size '() 0) → '(())
- (sub-size (list 1 2 3) 3) → '((1 2 3))
- (sub-size (list 1 2 3) 2) → '((1 2) (2 3))
- (sub-size (list 1 2 3) 1) → '((1) (2) (3))

; Signature: sub-size-map(list, func, size)
; Type: (list(T), (T) => T, number) => list(T)
; Purpose: gets a $list$, a function $func$ and a number $size$ and returns a new list of all the sublists of $list$ of length $size$ that all their elements are mapped by $func$.
; Pre-conditions:

- List of T values
- Func takes an element from list and returns a new value of T values
- Size non negative number

; Tests:

- (sub-size-map '() (lambda (x) (+ x 1)) 0) → '(())
- (sub-size-map (list 1 2 3) (lambda (x) (+ x 1)) 3) → '((2 3 4))
- (sub-size-map (list 1 2 3) (lambda (x) (+ x 1)) 2) → '((2 3) (3 4))
- (sub-size-map (list 1 2 3) (lambda (x) (+ x 1)) 1) → '((2) (3) (4))

; Signature: root(tree)
; Type: (list(T)) => T
; Purpose: gets a list representing a tree and returns the value of the root.
; Pre-conditions:

- List: a tree represented as a list of T values

; Tests:

- (root '(1 (#t 3 #t) 2)) → 1
- (root '(#t (#t 3 #t) 2)) → #t

; Signature: left(tree)
; Type: (list(T)) => list(T)
; Purpose: gets a list representing a tree and returns the subtree of the left son, or an empty list if there is no left son.
; Pre-conditions:

- List: a tree represented as a list of T values

; Tests:

- (root '(1 (#t 3 #t) 2)) → '(#t 3 #t)
- (root '(#t (#t 3) 2)) → '(#t 3)

; Signature: right(tree)
; Type: (list(T)) => list(T)
; Purpose: gets a list representing a tree and returns the subtree of the right son, or an empty list if there is no right son.
; Pre-conditions:

- List: a tree represented as a list of T values

; Tests:

- (right '(1 (#t 3 #t) 2)) → '(2)
- (right '(#t (#t 3) 4)) → '(4)

; Signature: count-node(tree, val)
; Type: (list(T), T) => number
; Purpose: returns the number of nodes whose value is equal to val.
; Pre-conditions:

- List: list of T values
- Val: value of T

; Tests:

- (count-node '(1 (#t 3 #t) 2) #t) → 2
- (count-node '(1 (#t 3 #t) 2) 4) → 0

; Signature: mirror-tree(tree)
; Type: (list(T)) => list(T)
; Purpose: given a list representing a $tree$, returns the mirrored tree.
; Pre-conditions:

- List: contains T values

; Tests:

- (mirror-tree '(1 (#t 3 4) 2)) → '(1 2 (#t 4 3))

; Signature: make-ok(value)
; Type: (T)=> result
; Purpose: - gets a $value$ and returns an ok structure for the $value$ of type result.
; Pre-conditions:

- Value: value of T values

; Tests:

- (define ok (make-ok 1))

; Signature: make-error(T)
; Type: (T) => result
; Purpose: gets an error $message$ and returns an error structure for the $message$ of type result.
; Pre-conditions:

- Message: message of T

; Tests:

- (define error (make-error “some error message”))

; Signature: ok?(ok)
; Type: (T) => boolean
; Purpose: type predicate for ok
; Pre-conditions:

- Ok of type T

; Tests:

- (ok? ok) → #t

; Signature: error?(error)
; Type: (T) => boolean
; Purpose: type predicate for $error$.
; Pre-conditions:

- Error of type T

; Tests:

- (error? ok) → #f

; Signature: result?(res)
; Type: (T) => boolean
; Purpose: type predicate for $result$.
; Pre-conditions:

- Res of type T

; Tests:

- (result? ok) → #t

; Signature: result->val(res)
; Type: (result) => T
; Purpose: gets a $result$ structure and returns the value it represents, or the error message for error. If the given result is not a result, return an error structure with the message "Error: not a result"
; Pre-conditions:

- Res: $result$ structure

; Tests:

- (result->val ok) → 1

; Signature: bind(func)
; Type: ((T)=>result) => (result)=>result
; Purpose: given a function $func$ from a non-result to result, returns a new function which given a result, returns the activation of $func$ on its value or an error structure accordingly. If the given result is not a result, return an error structure.
; Pre-conditions:

- Func: $func$ from a non-result to result

; Tests:

- (define inc-result (bind (lambda (x) (make-ok (+ x 1)))))
- (result->val (inc-result ok)) → 2