1.1. צורות מיוחדות נדרשות בשפות התכנות מכיוון שישנן פעולות האפשריות למימוש עבור צורות מיוחדות אך אינן אפשריות למימוש באותה דרך עבור אופרטורים פרימיטיביים וזאת משום שלצורות מיוחדות ישנן כללי חישוב שונים מכללי החישוב של האופרטורים הפרימיטיביים.

דוגמה : "if" הינה צורה מיוחדת המחשבת את התנאי הבוליאני קודם ורק לאחר מכן מחשבת את קטע הקוד העונה על התנאי (קטע הקוד השני אינו מחושב כלל). בעוד שהאופרטורים הפרימיטיביים מחשבים את שני קטעי הקוד.

1.2. לא. מכיוון שב- L1, הצורה המיוחדת "define" מסייעת במניעת שכפול של קטעי קוד בלבד.

1.3. כן.

דוגמה נגדית :

```
(define sqrt (lambda (x) (sqrt-iter 1 x)))

(define sqrt-iter
  (lambda (guess x)
     (if (good-enough? guess x)
         guess
         (sqrt-iter (improve guess x) x))))

(define abs (lambda (x) (if (< x 0) (- x) x)))
(define square (lambda (x) (* x x)))
(define epsilon 0.0001)

(define good-enough?
  (lambda (guess x)
     (< (abs (- (square guess) x)) epsilon)))

(define average
  (lambda (x y) (/ (+ x y) 2.0)))

(define improve
  (lambda (guess x)
    (average guess (/ x guess))))
```

1.4. primOp – המימוש מתבצע באופן ישיר, דבר המאפשר הרצה מהירה וייעול השימוש במשאבים.
closure – ניתן לגשת לסביבה שקיימת בזמן החישוב של הפונקציה ובכך מאפשר כתיבת קוד גמישה ומודולרית.

1.5. Map – סדר יישום הפונקציה על כל אחד מאיברי הרשימה יכול להיות מקבילי מכיוון שלהפעלת הפונקציה על כל איבר מהרשימה אין כל השפעה על שאר איברי הרשימה.

Reduce – סדר יישום הפונקציה על כל אחד מאיברי הרשימה יכול להיות מקבילי עבור פונקציות המאפשרות קומוטטיביות ואסוציאטיביות בלבד. הסיבה לכך היא שבכל שלב, הפונקציה מהסדר הגבוה Reduce מבצעת את הפונקציה הנתונה על איבר מהרשימה ועל הפלט שהתקבל מהפעלות הפונקציה הנתונה על האיברים שעבר עליהם עד כך ברשימה, וכן מעבירה לשלב הבא את הפלט שהתקבל מהפונקציה הנתונה (רקורסיבי).

Filter – סדר יישום הפונקציה על כל אחד מאיברי הרשימה יכול להיות מקבילי מכיוון שלהפעלת הפונקציה על כל איבר מהרשימה אין כל השפעה על שאר איברי הרשימה.

All – סדר יישום הפונקציה על כל אחד מאיברי הרשימה יכול להיות מקבילי מכיוון שלהפעלת הפונקציה על כל איבר מהרשימה אין כל השפעה על שאר איברי הרשימה.

Compose – סדר יישום הפונקציה על כל אחד מאיברי הרשימה יכול להיות מקבילי עבור פונקציות המאפשרות קומוטטיביות ואסוציאטיביות בלבד. הפונקציה מהסדר הגבוה Compose מקבלת מספר פונקציות שונות שתפעלנה על רשימת האיברים. לכן, במידה שהפונקציות הינן קומוטטיביות ואסוציאטיביות אז לא תהיינה כל משמעות לסדר הפעלתן של הפונקציות השונות על רשימת האיברים.

1.6. כתובת מילונית של משתנה היא הצבעה למיקום המשתנה. הכתובת המילונית בנויה מ3 מרכיבים – שם המשתנה, העומק של המשתנה והמיקום של המשתנה באותו העומק.

לדוגמא: ((lambda (x) (* [x : 0 0] [x : 0 0]).

1.7.

(cond <cond-clause>+ <else-clause>) /

CondExp(cond-clauses: List(cond-clause), else-clause: cond-clause)

<cond-clause> ::= (<exp> <exp>+) / cond-cluase(test:cexp, then:List(cexp))

<else-clause> ::= (else <cexp>+) / cond-cluase(test:true, then:List(cexp))

2.1.

take

; Signature: take(lst,pos)

; Type: [List(T1) * number -> List(T1)]

; Purpose: return the pos first number of elements in the list.

; Pre-conditions: true

; Tests: (list 1 2 3) 2) -> ‘(1 2);

take-map

; Signature: take-map(list,func,pos)

; Type: [List(T1) * (T1->T2) * number -> List(T2)]

; Purpose: return a new list whose elements are the first pos elements mapped by func.

; Pre-conditions: true

; Tests: (take-map (list 1 2 3) (lambda (x) (* x x)) 2) -> ‘(1 4);

take-filter

; Signature: take-filter(lst,pred,pos)

; Type: [List(T1) * (T1->boolean) * number -> List(T1)]

; Purpose: return a new list whose elements are the first pos elements of the list that satisfy pred.

; Pre-conditions: true

; Tests: (take-filter (list 1 2 3 4) (lambda (x) (> x 1)) 2) -> ‘(2 3);

sub-size

; Signature: sub-size(lst,size)

; Type: [List(T1) * number -> List(List(T1))]

; Purpose: return a new list of all the sublists of list and length size.

; Pre-conditions: true

; Tests: (sub-list (list 1 2 3) 2) -> ((1 2) (2 3));

sub-size-map

; Signature: sub-size-map(lst,func,size)

; Type: [List(T1) * (T1->T2) * number -> List(List(T2))]

; Purpose: return a new list of all the sublists of length size when the elements are mapped by func.

; Pre-conditions: size <= length list , if the list is not empty – size >= 1.

; Tests: (sub-size-map (list 1 2 3) 2 (lambda (x) (+ x 1))) -> ‘((2 3) (3 4));

.2.2

root

; Signature: root(tree)

; Type: [List((List(T)) -> T]

; Purpose: return the value of the root.

; Pre-conditions: true

; Tests: (root ‘(1 (#t 3 4) 2)) -> 1;

left

; Signature: left(tree)

; Type: [List(List(T)) -> T]

; Purpose: return the subtree of the left son.

; Pre-conditions: true

; Tests: (left ‘(1 (#t 3 4) 2)) -> #t;

right

; Signature: right(tree)

; Type: [List(List(T)) -> T]

; Purpose: return the subtree of the right son.

; Pre-conditions: true

; Tests: (right '(1 (#t 3 4) 2)) -> 2;

count-node

; Signature: count-node(tree,val)

; Type: [List(List(T)) * T -> number]

; Purpose: return the number of nodes whose value is equal to val.

; Pre-conditions: true

; Tests: (count-node '(1 (#t 3 #t) 2) #t) -> 2;

mirror-tree

; Signature: mirror-tree(tree)

; Type: [List(List(T)) -> (List(List(T))]

; Purpose: return the mirrored given tree (represented by a list).

; Pre-conditions: true

; Tests: (mirror-tree '(1 (#t 3 4) 2))) -> '(1 2 (#t 4 3));

.2.3

make-ok

; Signature: make-ok(val)

; Type: [T -> pair(string T)]

; Purpose: create an ok result with the given value.

; Pre-conditions: true

; Tests: (make-ok 1) -> '(ok 1);

make-error

; Signature: make-error(msg)

; Type: [string -> pair(string, string)]

; Purpose: create an error result with the given message.

; Pre-conditions: true

; Tests: (make-error "some error") -> '(error "some error");

ok?

; Signature: ok?(res)

; Type: [result[T] -> Boolean]

; Purpose: return true if the given result is ok, or false if it is not.

; Pre-conditions: res is result type.

; Tests: (ok? (make-ok 1)) -> #t; (ok? (make-error “msg”)) -> #f;

error?

; Signature: error?(res)

; Type: [result[T] -> Boolean]

; Purpose: return true if the given result is error, or false if it is not.

; Pre-conditions: res is result type.

; Tests: (error? (make-ok 1)) -> #f; (error? (make-error “msg”)) -> #t;

result?

; Signature: result?(res)

; Type: [T -> Boolean]

; Purpose: return true if the given result is result type, or false if it is not.

; Pre-conditions: res is result type.

; Tests: (result? (make-ok 1)) -> #t; (result? ‘(1,2,3)) -> #f;

result->val

; Signature: result->val(res)

; Type: [result[T] -> T]

; Purpose: return the value of res.

; Pre-conditions: res is result type.

; Tests: (result-val (make-ok 1)) -> 1; (result-val ‘(1,2,3)) -> “Error: not a result”;

bind

; Signature: bind(f(result))

; Type: [(result[T] -> T) -> T]

; Purpose: return a new function which returns the activation of f on the given res value, or an error structure accordingly.

; Pre-conditions:

; Tests: (define inc-result (bind (lambda (x) (make-ok (+ x 1)))))

(define ok (make-ok 1))

(result->val (inc-result ok)) -> 2 ;