PPL- ASSIGNMENT 2

חלק 1 שאלות תאורטיות

1.1 Special forms נחוצים בשפות תכנות כיוון שהם מאפשרים לנו לכתוב קוד מורכב יותר בעזרת סינקס וסמנטיקה שלא ניתנים לביטוי ע"י פונקציות רגילות.
לא ניתן להגדיר special forms כאופרטור פרימיטיבי מאחר ול-special forms יש פונקציונליות מסוימת שלא ניתן ליישם עם אופרטורים פרימיטיביים.
לדוגמא: cif – מחשב ביטוי לפי תנאי מסוים לעומת אופרטור פרימיטיבי שיחשב את כל ערכי הביטויים שיקבל לפני החזרת ערך.

1.2 אין תוכנית ב-L1 שלא ניתן להמיר לתוכנית שוות ערך ב-L0.
special form "define" מאפשר לתת שם לביטוי, דבר ההופך את הקוד לקריא וברור יותר, אך אינו הכרחי לנכונות הקוד ולכן ניתן לכתוב כל תוכנית ב-L1 גם ב-L0 (בה "define" לא קיים).

1.3 יש תוכנית ב-L2 שלא ניתן להמיר לתוכנית שוות ערך ב-L20.
נשים לב. שב-L2 ניתן לבנות פונקציות רקורסיביות בעוד שב-L20 ה-special form "define" לא קיים, דבר אשר אינו מאפשר בניית פונק' רקורסיבית ב-L20 מאחר ובפונק' רקורסיבית נחזיק משתנה זוכר, דבר שאינו מתאפשר ללא "define".

1.4 יתרון של primOp: ניתן להשתמש בו על מנת לממש מבני נתונים מורכבים בצורה יעילה.
יתרון של closure: ניתן לגשת למשתנים דרך ה- lexical environment אחרי שנסגר ה-scope של פונקציית האב.

1.5
- **map**- ניתן ליישם במקביל את map על איברי הרשימה כיוון שהפעלת map על כל איבר לא משפיעה על האיברים האחרים.
- **reduce**- הפעלת reduce על איברי הרשימה יכולה לקרות במקביל כאשר אין חשיבות לסדר הפעלת הפונק' על האיברים (למשל בחיבור כל איברי המערך) וכאשר יש חשיבות לסדר היא תהיה חייבת לעבוד בצורה סדרתית.
- **filter**- ניתן ליישם במקביל את filter על איברי הרשימה כיוון שהפעלת filter על כל איבר לא משפיעה על האיברים האחרים.
- **all**- ניתן ליישם במקביל את all על איברי הרשימה כיוון שהפעלת all על כל איבר לא משפיעה על האיברים האחרים.
- **Compose**- הפעלת compose על איברי הרשימה יכולה לקרות במקביל כאשר אין חשיבות לסדר הפעלת הפונק' על האיברים וכאשר יש חשיבות לסדר היא תהיה חייבת לעבוד בצורה סדרתית.

1.6 lexical address זה המקום בו המשתנה מוגדר בתוכנית. בכל scope בו מגדירים את המשתנה הוא מקבל כתובת אחרת.
לדוגמא-

```
(let  ((x 3) (y 4))            x=x1
  (+ let  ((x (+ y 5)))        x=x2
     (* x y)  <----------      x מתייחס ל-x2
  x))  <----------             x מתייחס ל-x1
```

1.7

```
<program> ::= (L31 <exp>+)                  / Program(exps:List(exp))
<exp> ::= <define> | <cexp>                 / DefExp | CExp
<define> ::= ( define <var> <cexp> )        / DefExp(var:VarDecl,
val:CExp)
<var> ::= <identifier>                      / VarRef(var:string)
<cexp> ::= <number>                         / NumExp(val:number)
        |  <boolean>                        / BoolExp(val:boolean)
        |  <string>                         / StrExp(val:string)
        |  ( lambda ( <var>* ) <cexp>+ )    / ProcExp(args:VarDecl[],
                                             /         body:CExp[]))
        |  ( if <cexp> <cexp> <cexp> )      / IfExp(test: CExp,
                                                    then: CExp,
                                                    alt: CExp)
        |  ( let ( <binding>* ) <cexp>+ )   /
LetExp(bindings:Binding[],
                                                    body:CExp[]))
        |  (cond <cond-clauses>)+(<else>)   / cond exp (cond clauses :
                                               List(cond-clause), else: cond-
                                                                     clause)
        |  ( quote <sexp> )                 / LitExp(val:SExp)
        |  ( <cexp> <cexp>* )               / AppExp(operator:CExp,
        |                                              operands:CExp[]))
<binding>  ::= ( <var> <cexp> )             / Binding(var:VarDecl,
                                                      val:Cexp)
```

```
<cond-clause> ::= (<cexp> <cexp>+) | (else <cexp>+)
<prim-op>  ::= + | - | * | / | < | > | = | not |  eq? | string=?
             | cons | car | cdr | list | pair? | list? | number?
             | boolean? | symbol? | string?
<num-exp>  ::= a number token
<bool-exp> ::= #t | #f
<str-exp>  ::= "tokens*"
<var-ref>  ::= an identifier token
<var-decl> ::= an identifier token
<sexp>     ::= symbol | number | bool | string | ( <sexp>* )
```

- **take**

  ;signature: take(lst, pos)

  ;Type: [List<Number> * Number → List<Number>]

  ;Purpose: Extract the first 'pos' elements of a list, returning a new list containing those elements.

  ;Pre-conditions: pos is a natural number.

  ;Tests: (take (list 1 2 3) 2) → '(1 2)

- **take-map**

  ;signature: take-map(lst, func, pos)

  ;Type: [List<Number> * (func: Number → Number) * Number → List<Number>]

  ;Purpose: Apply a function to the first pos elements of a list and return a list with the altered elements.

  ;Pre-conditions: pos is a natural number, func takes 1 argument.

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

- **take-filter**

  ;signature: take-filter(list, pred, pos)

  ;Type: [List<Number> * (pred: Number → Boolean) * Number → List<Number>]

  ;Purpose: Return a new list containing the first pos elements of lst that satisfy pred.

  ;Pre-conditions: pos is a natural number, pred takes 1 arguments and returns a boolean.

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

- **sub-size**

  ;signature: sub-size(list, size)

  ;Type: [List<Number> * Number → List<Number>]

  ;Purpose: Returns a list of all the contiguous sublists of 'lst' that have 'size' elements.

  ;Pre-conditions: size is a natural number.

  ;Tests: (sub-size (list 1 2 3) 3) → '((1 2 3))

- **sub-size-map**

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

;Type: [List<Number> * (func: Number → Number) * Number → List<Number>]

;Purpose: Return a list of all the contiguous sublists of lst that have 'size' elements, and where each element of each sublist has been transformed by func.

;Pre-conditions: size is a natural number, func takes 1 argument.

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

- **root**

;signature: root(tree)

;Type: [List<Any> → Any]

;Purpose: Return the value of a binary tree root node.

;Pre-conditions: The input list is a valid representation of a binary tree.

;Tests: (root('(1 (*t 3 4) 2))) → 1

- **left**

;signature: left(tree)

;Type: [List<Any> → List<Any> | Atom].

;Purpose: Retrieve the subtree of the left son of a given binary tree.

;Pre-conditions: The input list is a valid representation of a binary tree.

;Tests: (root('(1 (*t 3 4) 2))) → '(*t 3 4)

- **right**

;signature: right(tree)

;Type: [List<Any> → List<Any> | Atom].

;Purpose: Retrieve the subtree of the right son of a given binary tree.

;Pre-conditions: The input list is a valid representation of a binary tree.

;Tests: (root('(1 (*t 3 4) 2))) → 2

- **count-node**

;signature: count-node(lst, val)

;Type: [List<Any> * Any → Number]

;Purpose: Count the number of nodes in a tree that have a specific value.

;Pre-conditions: The input list is a valid representation of a binary tree, val is an atomic value.

;Tests: (count-node '(1 (*t 3 *t) 2) *t) → 2

- mirror-tree

;signature: mirror-tree(tree)

;Type: [List<Any> → List<Any>]

;Purpose: Return a mirrored tree of the input tree.

;Pre-conditions: The input list is a valid representation of a binary tree.

;Tests: (mirror-tree '(1 (*t 3 4) 2) → '(1 2 (*t 4 3))

- make-ok

;signature: make-ok(val)

;Type: [Any → Result<Any>]

;Purpose: Create an "ok" structure for a given value.

;Pre-conditions: None.

;Tests: (make-ok 1) → {tag: "ok", value: 1}

- make-error

;signature: make-error(msg)

;Type: [String → Result<String>]

;Purpose: Create an "error" structure with the given message.

;Pre-conditions: msg is a String.

;Tests: (make-error "Error") → {tag: "Failure", message: "Error"}

- ok?

;signature: ok?(res)

;Type: [Result<Any> → Boolean]

;Purpose: Check if "res" is an instance of an "ok" structure.

;Pre-conditions: none.

;Tests: (define ok(make-ok 1))
(ok? ok) → #t

- **error?**

;signature: error?(res)

;Type: [Result<Any> → Boolean]

;Purpose: Check if "res" is an instance of an "error" structure.

;Pre-conditions: none

;Tests: (define ok(make-ok 1))
(error? ok) → #f

- **result?**

;signature: result?(res)

;Type: [Any → Boolean]

;Purpose: Check if "res" is an instance of an "error" structure.

;Pre-conditions: None

;Tests: (define ok(make-ok 1))
(result? ok) → #t

- **result→val**

;signature: result→val(res)

;Type: Any → Any .

;Purpose: To extract the value of a result structure if it represents a valid result, or the error message if it represents an error.

;Pre-conditions: None.

;Tests: (define ok(make-ok 1))
(result→val ok) → 1

- **bind**

;signature: bind(func)

;Type: [func:(Any → result<Any>) → func:(result<Any> → result<Any>)

;Purpose: To transform a function that operates on non-result values into a function that operates on result values.

;Pre-conditions: The input function has to take a non-result value as an input and returns a result.

;Tests: (bind (lambda (x) (make-ok (+ x 1)))
(make-ok 2) ⟶ {tag: "ok", value: 3}