עבודה 3 בעקרונות שפות תכנות

# חלק 1

## שאלה 1

Special form הוא ביטוי מורכב שיש לו חוק חישוב אחר מאשר ביטוי מורכב רגיל (application expression) בו מחשבים את הביטוי הראשון (האופרטור) והאופרנדים (באופן בלתי תלוי) ולאחר מכן מפעילים את האופרטור (Closure or primitive operator) עם ה-Values של האופרנדים.

Let לא מחושב באותו האופן המתואר לעיל ולכן let הוא Special form.

## שאלה 2

טעויות סמנטיות היכולות להיגרם בעת הפעלת תוכנית L3:

1. הפעלה של אופרטורים פרימיטיביים ו-Closures שלא תואמת את החתימות שלהם. דוגמאות:
   1. Type לא נכון של הפרמטרים (למשל העברת ערכים בוליאניים להפעלת +).
   2. מס' לא נכון של פרמטרים (למשל הפעלה ללא פרמטרים של -).
2. שגיאה בעת חישוב של אופרטור פרימיטיבי. דוגמאות:
   1. חלוקה באפס.
   2. ניסיון הפעלת cdr על רשימה ריקה.
3. Evaluation של ביטוי מורכב שהרכיב הראשון שלו איננו Primitive operator, איננו Closure ואיננו Special form keyword.  
   לדוגמה: (3 1 4).
4. ניסיון חישוב של משתנה חופשי. לדוגמה: (L3 (+ x 3)).

## שאלה 3

### סעיף 3.1

הרחבת ה-AST של L3 על מנת לתמוך בקבלת values כדי להימנע משימוש ב-valueToLitExp:

<program> ::= (L3 <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> ::= <num-exp> / NumExp(val:number)

| <bool-exp> / BoolExp(val:boolean)

| <str-exp> / StrExp(val:string)

| ( lambda ( <var>\* ) <cexp>+ ) / ProcExp(params:VarDecl[], body:CExp[]))

| ( if <cexp> <cexp> <cexp> ) / IfExp(test: CExp, then: CExp, alt: CExp)

| ( let ( binding\* ) <cexp>+ ) / LetExp(bindings:Binding[], body:CExp[]))

| ( quote <sexp> ) / LitExp(val:SExp)

| ( <cexp> <cexp>\* ) / AppExp(operator:CExp, operands:CExp[]))

| <value> / Value

<binding> ::= ( <var> <cexp> ) / Binding(var:VarDecl, val:Cexp)

<prim-op> ::= + | - | \* | / | < | > | = | not | and | or | eq? | string=?

| cons | car | cdr | pair? | number? | list

| boolean? | symbol? | string? ##### L3

<num-exp> ::= a number token

<bool-exp> ::= #t | #f

<str-exp> ::= a sequence of characters between double quotes

<var-ref> ::= an identifier token

<var-decl> ::= an identifier token

<sexp> ::= symbol | number | bool | string |

(<sexp>+ . <sexp>) | ( <sexp>\* ) ##### L3

<value> ::= closure | <sexp>

### סעיף 3.2

החלקים ב-Interpreter שישתנו על מנת לאמץ את השינוי:

* ב-L3-ast.ts:
  + יש להוסיף ל-disjoint union בשם CExp את הטיפוס Value.
* ב-L3-eval.ts
  + יש לשנות את L3applicativeEval כך שאם exp הוא מסוג Value, היא תחזיר אותו ללא שינוי.
  + להסיר את הפונקציה valueToLitExp.
  + בתוך הפונקציה applyClosure:
    - להסיר את הקריאה ל-valueToLitExp (ואת ההשמה ל-litArgs).
    - לקרוא ל-substitute עם args (ולא עם litArgs, שכן שורה זו נמחקה).

### סעיף 3.3

האפשרות המועדפת היא שימוש ב-valueToLitExp, משום שכך שומרים על היגיון וסדר במבנה ה-interpreter וגם בקוד שלו. התוספת של הטיפוס Value ל-disjoint union בשם CExp מטעה ולא הגיונית, שכן ערך הוא איננו ביטוי.

בנוסף, בכל פונקציה שמקבלת CExp יש להוסיף בדיקת טיפוס נוספת אשר מסבכת את הקוד, ולא אינטואיטיבית.

יתר על כן, ה-syntax specification של השפה נהיה מורכב יותר.

## שאלה 4

אין צורך בפונקציה valueToLitExp במימוש ה-interpreter בשיטת ה-normal evaluation, נסביר.

בשיטת applicative evaluation, בעת הפעלה של closure האופרנדים מחושבים לפני האופרטור ונהפכים לערך. אז, האופרטור מחושב ולבסוף מתבצע החישוב של גוף ה-closure (שפוטנציאלית מכיל התייחסות לארגומנטים של ה-closure, כלומר לאופרנדים). מאחר שחישוב מתבצע על ביטויים ולא ערכים, יש צורך להמיר את הערכים של האופרנדים אשר חושבו בחזרה לביטויים.

לעומת זאת בשיטת normal evaluation בעת הפעלה של closure האופרטור מחושב לפני האופרנדים. מיד לאחר מכן מתבצע החישוב של גוף ה-closure, כאשר הארגומנטים שלו (האופרנדים) הם עדיין ביטויים. האופרנדים יומרו לערך רק כאשר מחשבים הפעלה של Primitive operation (בשלב ה"אחרון" בחישוב). בשום שלב לא יועברו ערכים לפונקציה שמקבלת ביטויים, לפיכך אין צורך בפונקציה valueToLitExp שממירה ערך לביטוי.

## שאלה 5

דוגמה לתוכנית שתרוץ מהר יותר תוך שימוש בשיטת normal eval:

((lambda (x) (5)) (+ 2 7))

נסביר: בשיטת normal eval יתבצע חישוב אחד (החזרה של הערך 5) בעוד שבשיטת applicative eval יתבצעו שני חישובים (ראשית (+ 2 7) ואז החזרה של הערך 5).

דוגמה לתוכנית שתרוץ מהר יותר תוך שימוש בשיטת applicative eval:

((lambda (x) (+ x x x x)) (+ 1 2))

נסביר: בשיטת normal eval יתבצעו 5 חישובים (נחשב את (+ 1 2) עבור כל מופע של x בגוף הפונקציה, ולבסוף נחשב את פעולת החיבור ביניהם), בעוד שבשיטת applicative eval יתבצעו רק שני חישובים (ראשית (+ 1 2) ואז פעולת החיבור בין ה-x-ים).

# חלק 3

## שאלה 1

### חלק 1 (שתי התוכניות)

נסביר את התנהגות התוכניות כמבוקש.

בשפת "lazy" הביטוי (define x (-)) מבצע binding בין המשתנה x ל-promise, אשר מכמס (encapsulates) את הביטוי (-) שיחושב רק כאשר "חייבים" לחשב אותו (למשל ע"י ניסיון הפעלה שלו, או למשל ע"י הפעלת Primitive operation שה-promise הוא אחד הפרמטרים שלו).

בתוכנית הראשונה כאשר רוצים לקבל את הערך של x אכן מקבלים promise, שכן לא ביצענו פעולה שמחייבת חישוב של ה-promise הזה.

בתוכנית השנייה הביטוי "1" מחושב ומוחזר הערך 1. שוב לא היה צורך בחישוב של ה-promise ולכן לא נתקלנו בשגיאה אף שהביטוי (-) לא תקין.

### חלק 2 (test-define-normal.ts)

הבעיה היא ש-define יוצר binding בין x לבין הביטוי (-), אשר מחושב בעת ה-binding (לפי normal evaluation, אבל מכיוון שזהו ביטוי מסוג AppExp והאופרטור פרימיטיבי, הוא מחושב מיד).

הפתרון הוא להוסיף כלל ל-evalDefineExps שיעטוף כל AppExp ב-promise.

## שאלות 2-3 (הקוד והבונוס)

הקוד שכתבנו מטפל גם במקרים המפורטים בדרישות מהבונוס, אך שומר על הממשק מההגדרה המקורית של המטלה (אין שינוי בטיפוס של Closure, וה-Env מכיל CExp בלבד (ולא Promise).

כפי שתיארתי למיכאל בשעת הקבלה, המימוש נעשה בעזרת המרת אופרנדים של AppExp בביטוי let שה-bindings שלו הם כל האיברים שנמצאים ב-Env טרם חישוב ה-AppExp.