# עבודה 3 בעקרונות שפות תכנות

## שאלה 1

סעיף 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.

סעיף 2

טעויות סמנטיות היכולות להיגרם בעת הפעלת תוכנית L3:

1. הפעלה לא תקינה של אופרטורים פרימיטיביים ו-Closure. דוגמאות:
   1. חלוקה באפס.
   2. Type לא נכון של הפרמטרים (למשל העברת ערכים בוליאניים להפעלת +).
   3. מס' לא נכון של פרמטרים (למשל הפעלה ללא פרמטרים של -).
2. חישוב לא תקין של ביטויי Special form.  
   לדוגמה: הערכת תנאי (הרכיב הראשון של if) שאיננה מחזירה ערך בוליאני.
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)).

סעיף 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> | <value> / DefExp | CExp

;; <value> ::= <number> / NumExp(val:number)

| <boolean>

| <string>

| <prim-op>

| <Closure>

| SymbolSExp

| EmptySExp

| CompoundSExp

;; <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(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[]))

;; <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

;; <number> ::= a number value

;; <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

סעיף 3.2

החלקים ב-Interpreter שישתנו על מנת לאמץ את השינוי:

* ה- AST. צריך לשנותו בהתאם לשינויים המפורטים מעלה ולהוסיף
* יש להוסיף ל- disjoint union בשם Exp את הטיפוס Value.
* יש לשנות את הפונקציות הבאות כך שאם ה-Exp שהן קיבלו כפרמטר הוא מסוג Value, הן יחזירו אותו ללא שינוי:
  + evalSequence
* להסיר את הקריאה ל-valueToLitExp בתוך הפונקציה applyClosure.
* substitute
  + sub
    - יש לשנות את טיפוס הפרמטר שהיא מקבלת (וטיפוס ההחזרה) מ-CExp ל-Exp.
    - להוסיף את הבדיקה isSExp ואם כן להחזיר את הערך כמו שהוא.
  + subVarRef
    - יש לשנות את טיפוס ההחזרה שלה מ-CExp ל-Exp.

סעיף 4

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

בשיטת normal evaluation בעת הפעלה של פונקציה האופרטור מחושב לפני האופרנדים, ורק כאשר האופרטור הוא פרימיטיבי האופרנדים מחושבים גם הם. אם האופרטור לא פרימיטיבי, הוא מחושב כאמור והאופרנדים מועברים כמו שהם (כ-CExp) ל-Closure שנוצר. לפיכך אין צורך להמירם בשלב כלשהו מ-CExp ל-Value אלא רק בשלב האחרון בחישוב

סעיף 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).

## שאלה 3

סעיף 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.