אוניברסיטת
אריאל
בשומרון

פקולטה: מדעי הטבע

מחלקה: מדעי המחשב ומתמטיקה

שם הקורס: שפות תכנות

קוד הקורס: 2-7036010

תאריך בחינה: 09/2015_ סמ' ב' מועד מיוחד

משך הבחינה: 3 שעות

שם המרצה: ערן עמרי

חומר עזר: אסור

שימוש במחשבון: לא

הוראות כלליות:

- כתבו את תשובותיכם בכתב קריא ומרווח (**במחברת התשובות בלבד** ולא על גבי טופס המבחן עצמו.
- בכל שאלה או סעיף (**שבהם נדרשת כתיבה, מעבר לסימון תשובות נכונות**). ניתן לכתוב – **לא ידע/ת** (מבלי להוסיף דבר מעבר לכך) – ולקבל 20% מהניקוד על השאלה או הסעיף (לא חל על חלקי סעיפים).
- אפשר להסתמך על סעיפים קודמים גם אם לא עניתם עליהם.
- יש לענות על כל השאלות.
- ניתן להשיג עד 108 נקודות במבחן.
- לנוחיותכם מצורפים שלושה קטעי קוד עבור ה- interpreter של FLANG בסוף טופס המבחן. הראשון – במודל ה-substitution. השני במודל הסביבות והשלישי במודל ה-Substitution-cache.

אוניברסיטת
אריאל
בשומרון

שאלה 1 — BNF — (23 נקודות):

נתון הדקדוק (BNF) הבא:

```
<TREE> ::=  <ATOM>
         |  <OP> <TREE> <TREE>
<ATOM> ::=  <num>
<OP>   ::= + | - | / | *
```

כאשר <num> מתאר ערך מספרי כלשהו על-פי הגדרת RACKET.

סעיף א' (5 נקודות):

מהי השפה שמגדיר הדקדוק? כתבו תיאור קצר (עד ארבע שורות), אך ברור (הקפידו להשתמש במונחים נכונים). ציינו האם ביטויים בשפה הם prefix, infix, או postfix.

(a+b) (ab+) (+ab)

סעיף ב' (5 נקודות):

הראו גזירה עבור מילה תוך שימוש בלפחות 7 כללי גזירה (ציינו בצורה מפורשת מהי המילה אותה גזרתם). ציינו את הכלל שבו השתמשתם בכל מעבר.

סעיף ג' (5 נקודות):

האם הדקדוק הנתון הינו רב-משמעי (כלומר, האם הוא סובל מ-ambiguity)? הסבירו את תשובתכם (בפרט, אם כן – הסבירו כיצד ניתן לתקן זאת מבלי לשנות את השפה).

סעיף ד' (8 נקודות):

השתמשו בדקדוק הנ"ל ובסמנטיקה של שפת PL בכדי להסביר את מושג ה-compositionality. בפרט, ציינו האם תכונה זו מתקיימת. אם לדעתכם היא אינה מתקיימת הסבירו כיצד ניתן לתקן זאת. אם לדעתכם היא מתקיימת תנו דוגמה לגבי המילה שבחרתם בסעיף ב'.

אוניברסיטת
אריאל
בשומרון

שאלה 2 — שאלות כלליות — (10 נקודות):

לפניכם מספר שאלות פשוטות. עליכם לבחור את התשובות הנכונות לכל סעיף (ייתכנו מספר תשובות נכונות – סמנו את כולן).

סעיף א' (5 נקודות): (סמנו את כל התשובות הנכונות)

אילו מהמשפטים הבאים נכונים לגבי תהליך ה-Parsing?

א. מציאת שמות מזהים חופשיים יכולה להתבצע כבר בתהליך ה-Parsing.
ב. על-פי המימוש שלנו, בסוף תהליך ה-Parsing קיבלנו FLANG שאינו יכול להיות ואריאנט Call.
ג. תהליך ה-Parsing מחוייב להסתיים לפני תחילת תהליך ההערכה של הקוד.
ד. במימוש שלנו, תהליך ה-Parsing במודל ההחלפות – זהה לתהליך ה-Parsing במודל ה-substitution cache.

סעיף ב' (5 נקודות): (סמנו את כל התשובות הנכונות)

אילו מהמשפטים הבאים נכונים לגבי מימושי האינטרפרטר שכתבנו עבור FLANG במודל ההחלפה, ה-substitution cache ובמודל הסביבות?

א. במימוש במודל ההחלפות, הפונקציה eval קוראת ל- subst פחות פעמים מאשר במודל ה-substitution cache.
ב. השוני המרכזי בין מימושי האינטרפרטר שכתבנו עבור FLANG במודל ההחלפות ובמודל הסביבות קשור באופן הדוק להבדל בין lexical scoping ו-dynamic scoping.
ג. במודל ה-substitution cache התייחסנו לפונקציות כ-first class.
ד. הפונקציה eval פעלה באופן זהה עבור פונקציות במימושי האינטרפרטר שכתבנו עבור FLANG במודל הסביבות וה-substitution cache. בפרט, במקרה זה היא החזירה את אותו ערך שקיבלה.

אוניברסיטת
אריאל
בשומרון

שאלה 3 — (34 נקודות):

נתון הקוד הבא:

```
(run "{with {z 5}
        {with {w {fun {y} {/ x y}}}
          {with {x 10}
            {call w z}}}}")
```

סעיף א' (12 נקודות):

תארו את הפעלות הפונקציה eval בתהליך ההערכה (יש 12 הפעלות) של הקוד מעלה במודל ההחלפות (על-פי ה-interpreter העליון מבין השלושה המצורפים מטה) - באופן הבא – לכל הפעלה מספר i תארו את הפרמטר האקטואלי ה - i (AST_i) וכן את הערך המוחזר מהפעלה זו (RES_i) .

הסבירו בקצרה כל מעבר. ציינו מהי התוצאה הסופית.

דוגמת הרצה: עבור הקוד

```
(run "{with {x 1} {+ x 2}}")
```

היה עליכם לענות (בתוספת הסברים)

AST_1 = (With x (Num 1) (Add (Id x) (Num 2)))
RES_1 = (Num 3)
AST_2 = (Num 1)
RES_2 = (Num 1)
AST_3 = (Add (Num 1) (Num 2))
RES_3 = (Num 3)
AST_4 = (Num 1)
RES_4 = (Num 1)
AST_5 = (Num 2)
RES_5 = (Num 2)

Final result: 3

סעיף ב' (14 נקודות):

תארו את הפעלות הפונקציה eval בתהליך ההערכה של הקוד מעלה במודל ה-substitution-cache (על-פי ה-interpreter התחתון מבין השלושה המצורפים מטה) - באופן הבא – לכל הפעלה מספר i תארו את AST_i – הפרמטר האקטואלי הראשון בהפעלה מספר i *(עץ התחביר האבסטרקטי)*, את $Cache_i$ – הפרמטר האקטואלי השני בהפעלה מספר i *(רשימת ההחלפות)* ואת RES_i – הערך המוחזר מהפעלה מספר i.

הסבירו בקצרה כל מעבר. ציינו מהי התוצאה הסופית.

דוגמת הרצה: עבור הקוד

```
(run "{with {x 1} {+ x 2}}")
```

היה עליכם לענות (בתוספת הסברים)

```
AST1  = (With x (Num 1) (Add (Id x) (Num 2)))
Cache1 = '()
RES1  = (Num 3)
AST2  = (Num 1)
Cache2 = '()
RES2  = (Num 1)
AST3  = (Add (Id x) (Num 2))
Cache3 = '((x (Num 1))
RES3  = (Num 3)
AST4  = (Id x)
Cache4 = '(x (NumV 1))
RES4  = (Num 1)
AST5  = (Num 2)
Cache5 = '((x (Num 1))
RES5  = (Num 2)

Final result: 3
```

סעיף ג' (8 נקודות):

מה היה קורה לו היינו מבצעים את ההערכה במודל הסביבות? מהי התשובה הרצויה? מדוע? (אין צורך לבצע הערכה)

הסבירו בקצרה מדוע נתקבלו התוצאות כפי שנתקבלו בסעיפים א' ו-ב'.

תשובה מלאה לסעיף זה לא תהיה ארוכה מחמש שורות.

שאלה 4 — הרחבת השפה FLANG במודל הסביבות — (41 נקודות):

נרצה להרחיב את השפה FLANG ולאפשר מספר משתנה של ארגומנטים לאופרטורים האריתמטיים ולפונקציות המוגדרות בקוד. להלן דוגמאות לטסטים שאמורים לעבוד:

```
(test (run "{with {x 5} {* x x x}}") => 125)
(test (run "{+ {- 10 2 3 4} {/ 3 3 1} {*}}")  => 3)
(test (run "{+ 1 2 3 4}") => 10)
(test (run "{+}") => 0)
(test (run "{*}") => 1)
(test (run "{/ 4}") => 4)
(test (run "{- 4}") => 4)
(test (run "{/ 16 2 4}") => 2)
(test (run "{/}")  =error> "bad syntax")

(test (run "{call {fun {x} {+ x x x 1}} 4}")
      => 13)

(test (run "{with {identity {fun {x} x}}
              {with {foo {fun {} {- 2 3 3}}}
                {call {call identity foo}}}}")
      => -4)
(test (run "{with {f {fun {y z x} {+ z y}}}
         {call f 4 1 3}}")
      =>  5)
(test (run "{with {f {fun {y z y} {+ z y}}}
         {call f 4 1 3}}")
     =error> "parse-sexpr: parameters should not repeat
in `fun' syntax")
(test (run "{with {f {fun {y z w} {+ x y}}}
              {with {x 5}
                 {call f 4 1}}}")
 =error> "eval: wrong number of arguments to function")

(test (run "{call {with {x 3}
                   {fun {x y z} {- x y z}}}
                 4 2 3}")
      => -1)
```

הערה: השתמשו בדוגמאות אלו בכדי להבין את דרישות התחביר, את אופן הערכת הקוד וכן את הודעות השגיאה שיש להדפיס במקרים המתאימים.

סעיף א' (הרחבת הדקדוק) (5 נקודות):

הוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים – 4 השלמות סה"כ לסעיף זה) ל –

```
#|
The grammar:
  <FLANG> ::= <num>
            | { + <FLANG>  ... }
            | { - <FLANG> <FLANG> ... }
            | { —«fill-in 01»—}
            | { —«fill-in 02»— }
            | { with { <id> <FLANG> } <FLANG> }
            | <id>
            | { fun { —«fill-in 03»— } <FLANG> }
            | { call —«fill-in 04»— }
|#
```

סעיף ב' (הרחבת הטיפוס FLANG) (6 נקודות):

הוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים – 6 השלמות סה"כ לסעיף זה) ל –

```
(define-type FLANG
    [Num  Number]
    [Add  —«fill-in 05»— ]
    [Sub  —«fill-in 06»— ]
    [Mul  —«fill-in 07»—]
    [Div  —«fill-in 08»—]
    [Id   Symbol]
    [With Symbol FLANG FLANG]
    [Fun  —«fill-in 09»—]
    [Call —«fill-in 10»— (Listof FLANG)])
```

סעיף ג' (פונקציות עזר לניתוח תחבירי) (8 נקודות):

לצורך תהליך הניתוח התחבירי (parsing), נשתמש בפונקציות העזר שתבנו כעת.
ראשית, כתבו פונקציה:

```
(: not-member :  Symbol (Listof Symbol) -> Boolean)
```

המקבלת סימבול ורשימה של סימבולים ומחזירה true אם הסימבול אינו מופיע ברשימה ו-false אחרת.

שנית, כתבו פונקציה:

```
(: all-unique :  (Listof Symbol) -> Boolean)
```

המקבלת רשימה של סימבולים ומחזירה true אם כל הסימבולים ברשימה שונים זה מזה ו-false אחרת.

סעיף ד' (parsing) (8 נקודות):
השתמשו בקוד הבא (ובפונקציות העזר שכתבתם) –

```
(: parse-sexpr* : (Listof Sexpr) -> (Listof FLANG))
;; to convert a list of s-expressions into a list of
FLANGSs
  (define (parse-sexpr* sexprs)
    (map parse-sexpr sexprs))
```

והוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים – 7 השלמות סה"כ לסעיף זה) ל –

```
 (: parse-sexpr : Sexpr -> FLANG)
  ;; to convert s-expressions into FLANGs
  (define (parse-sexpr sexpr)
    (match sexpr
      [(number: n)     (Num n)]
      [(symbol: name)  (Id name)]
      [(cons 'with more)
       (match sexpr
         [(list 'with (list (symbol: name) named) body)
          (With name (parse-sexpr named)
                     (parse-sexpr body))]
         [else (error 'parse-sexpr "bad `with' syntax in
~s" sexpr)])]
      [(cons 'fun more)
       (match sexpr
         [(list 'fun (list (symbol: names) ... ) body)
          (if —«fill-in 11»— ;; verify all are different
              (Fun —«fill-in 12»— )
              (—«fill-in 13»—)];; If found repetition
         [else (error 'parse-sexpr "bad `fun' syntax in
~s" sexpr)])]
      [(list '+ args ...) (Add (parse-sexpr* args))]
      [(list '- fst args ...) —«fill-in 14»—]
      [—«fill-in 15»—]
      [—«fill-in 16»—]
      [(list 'call fun args ...) —«fill-in 17»—]
      [else (error 'parse-sexpr "bad syntax in ~s"
sexpr)]))
```

סעיף ה' (14 נקודות):

בסעיף זה נכתוב את חלק הקוד המבצע הערכה (evaluation).
שימו לב – ההגדרות הבאות נשארות בדיוק כמו בקוד המצורף מטה.

```
;; Types for environments, values, and a lookup function

(define-type ENV
  [EmptyEnv]
  [Extend Symbol VAL ENV])

(define-type VAL
  [NumV Number]
  [FunV (Listof Symbol) FLANG ENV])

(: lookup : Symbol ENV -> VAL)
(define (lookup name env)
  (cases env
    [(EmptyEnv) (error 'lookup "no binding for ~s" name)]
    [(Extend id val rest-env)
     (if (eq? id name) val (lookup name rest-env))]))
```

השתמשו בהגדרות הפורמליות הבאות –

```
#| Formal specs for `eval':
      eval(N,env)                    = N
  eval({+ E ...}, env) = eval(E,env) + ...
  eval({- E1 E ...}, env) = eval(E1,env) - (eval(E,env) + ...)
  eval({* E ...}, env) = eval(E, env) * ...
  eval({/ E1 E ...}, env)=eval(E1, env) / (eval(E, env) * ...)

  eval(x,env)                    = lookup(x,env)
  eval({with {x E1} E2},env) =
                            eval(E2,extend(x,eval(E1,env),env))
  eval({fun {x ... } E},env)       = <{fun {x ... } E}, env>
  eval({call E1 E ... },env1)
                      = eval(Ef,extend(x,eval(E2,env1),env2))
                    if eval(E1,env1) = <{fun {x} Ef}, env2>
                    and number of parameters is as implied by f
                       = error!          otherwise  |#
```

והוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים – 10 השלמות סה"כ לסעיף זה) לפונקציות הבאות:
פונקציית עזר – ראשית נכתוב פונקציית עזר אשר תדע להוסיף רשימת זוגות לסביבה קיימת על-ידי הפעלה רקורסיבית של הבנאי המתאים.

```
(: extend* : (Listof Symbol) (Listof VAL) ENV -> ENV)
;; extends a given environment according to the
;; lists of names and values, respectively.
;; Assumes that the two lists are of the same length.
(define (extend* names vals env)
  (if (and (null? names)(null? vals))
      —«fill-in 18»—
      —«fill-in 19»—)
```

השתמשו בפונקציה שכתבתם ובשתי הפונקציות הבאות בכדי להשלים את הקוד החסר ל-eval.

```
(: NumV->number : VAL -> Number)
    (define (NumV->number v)
      (cases v
        [(NumV n) n]
        [else (error 'NumV->number "expects a number, got: ~s"
v)]))

(: eval* : (Listof FLANG) ENV -> (Listof VAL))
  ;; evaluates a list of FLANG expressions by
  ;; reducing it to a list of values
(define (eval* l env)
  (: eval-straight : FLANG -> VAL)
  (define (eval-straight exp)
    (eval exp env))
  (map eval-straight l))



(: eval : FLANG ENV -> VAL)
  ;; evaluates FLANG expressions by reducing them to values
(define (eval expr env)
  (cases expr
    [(Num n) (NumV n)]
    [(Add args) (NumV —«fill-in 20»—)]
      [(Sub fst args) —«fill-in 21»—)]
      [—«fill-in 22»— ]
      [—«fill-in 23»— ]
      [(With bound-id named-expr bound-body)
```

```
        (eval bound-body
              (Extend bound-id (eval named-expr env) env))]
      [(Id name) —«fill-in 24»— ]
      [(Fun bound-ids bound-body) —«fill-in 25»— ]
      [(Call fun-expr arg-exprs)
       (let ([fval (eval fun-expr env)])
         (cases fval
           [(FunV bound-ids bound-body f-env)
            (if —«fill-in 26»—
                —«fill-in 27»—
                (error 'eval "wrong number of arguments to
function ~s" fval))]
           [else (error 'eval "`call' expects a function, got:
~s"
                                fval)]))]))
```

הדרכה: השתמשו בפונקציות map ו- foldl של RACKET המתוארות מטה.
אין צורך לטפל בחלוקה באפס.

תוספות:

הפונקציה map:

קלט: פרוצדורה proc ורשימה lst

פלט: רשימה שמכילה אותו מספר איברים כמו ב- lst – שנוצרה ע"י הפעלת הפרוצדורה proc על כל אחד מאיברי הרשימה lst. (ההסבר הבא הוא כללי יותר – כי למעשה הפונקציה map יכולה לטפל במספר רשימות – לצורך השאלה הנתונה לא תזדקקו לשימוש כזה)

(map proc lst ...+) → list?

proc : procedure?

lst : list?

Applies proc to the elements of the lsts from the first elements to the last. The proc argument must accept the same number of arguments as the number of supplied lsts, and all lsts must have the same number of elements. The result is a list containing each result of proc in order.

דוגמאות:

```
> (map add1 (list 1 2 3 4))
'(2 3 4 5)
> (map (lambda (x) (list x))
    '(sym1 sym2 33))
'((sym1) (sym2) (33))
```

אוניברסיטת
אריאל
בשומרון

הפונקציה foldl:

קלט: פרוצדורה proc, ערך התחלתי init ורשימה lst

פלט: ערך סופי (מאותו טיפוס שמחזירה הפרוצדורה proc) שנוצר ע"י הפעלת הפרוצדורה proc על כל אחד מאיברי הרשימה lst תוך שימוש במשתנה ששומר את הערך שחושב עד כה – משתנה זה מקבל בערך התחלתי את הערך של init. (ההסבר הבא הוא כללי יותר – כי למעשה הפונקציה foldl יכולה לטפל במספר רשימות – לצורך השאלה הנתונה לא תזדקקו לשימוש כזה)

(foldl proc init lst ...+) → any/c

proc : procedure?

init : any/c

lst : list?

Like map, foldl applies a procedure to the elements of one or more lists. Whereas map combines the return values into a list, foldl combines the return values in an arbitrary way that is determined by proc.

דוגמאות:

```
> (foldl + 0 '(1 2 3 4))
10
> (foldl cons '() '(1 2 3 4))
'(4 3 2 1)
```

סמסטר - ב', מועד - מיוחד. 9/2015

שפות תכנות

שאלה 1:

(א). הביטויים בשפה הינם - prefix.

שפה בה הביטויים שמכילים סימני פעולה, וכל סימן פעולה מאותם לשני ביטויים (או ביטוי או אטום).

(ב).

<Tree> ::=	<ATOM>	(1)
	| <OP> <Tree> <Tree>	(2)
<ATOM> ::=	<num>	(3)
<OP> ::=	+	(4)
	| -	(5)
	| /	(6)
	| *	(7)

(*) נרצה לבדוק עבור המילה -

+ 2 - 3 / 4 * 5 2

<Tree> $\overset{(2)}{\Rightarrow}$ <OP> <Tree> <Tree> $\overset{(4)}{\Rightarrow}$ + <Tree> <Tree>

$\overset{(1)+(2)}{\Rightarrow}$ + <ATOM> <OP> <Tree> <Tree>

$\overset{(5)}{\Rightarrow}$ + <ATOM> - <Tree> <Tree>

$\overset{(1)+(2)}{\Rightarrow}$ + <ATOM> - <ATOM> <OP> <Tree> <Tree>

$\overset{(6)}{\Rightarrow}$ + <ATOM> - <ATOM> / <Tree> <Tree>

$\overset{(1)+(2)}{\Rightarrow}$ + <ATOM> - <ATOM> / <ATOM> <OP> <Tree> <Tree>

$\overset{(7)}{\Rightarrow}$ + <ATOM> - <ATOM> / <ATOM> * <Tree> <Tree>

$\overset{(1)\times 2}{\Rightarrow}$ + <ATOM> - <ATOM> / <ATOM> * <ATOM> <ATOM>

$\overset{(3)\times 5}{\Rightarrow}$ + <num> - <num> / <num> * <num> <num>

2 3 4 5 2

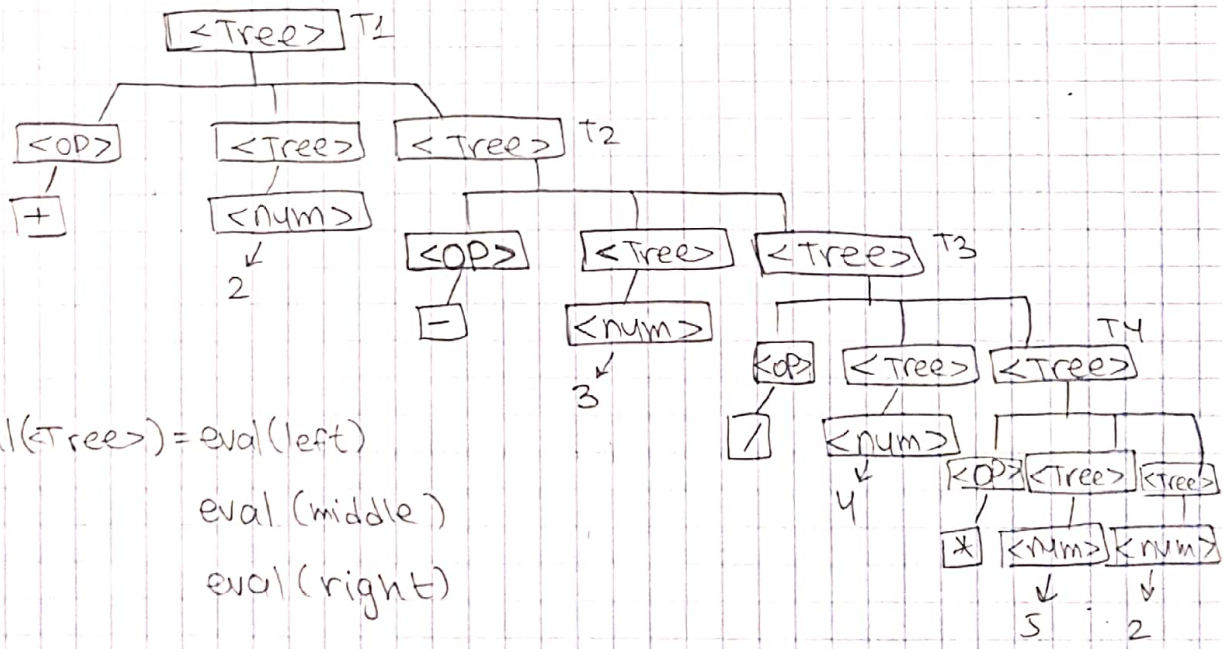
(ב). בשפה בלי אין דו-משמעות, משום שאין מולה בשפה שניתן לצור אותה ע"י שני עצים, ובאות משום שהיו הראשון קודם (שהוא <OP> - משום שהפעולה הוא ב- prefix.

(ג). בבקרוק הלא מתקיים תכונת ה- Compositinality ובאות מכיוון שפי הסמנטיקה של בל נקבע את הפעולה לפי סדר prefix - הפעולה הראשונה בביטוי מתקבע אחרונה בחישוב, לכן אין לזה בעיה.

לפי הדוגמא מעלה ב-

+2 -3 /4 *5 2

נצייר את עץ הגזירה -



eval(<Tree>) = eval(left)
eval(middle)
eval(right)

eval(T1) = (+ 2 (eval(T2))) =
= (+ 2 (- 3 (eval(T3)))) =
= (+ 2 (- 3 (/ 4 (eval(T4))))) =
= (+ 2 (- 3 (/ 4 (* 5 2)))) = 4.6

10

4.6 2.6 0.4

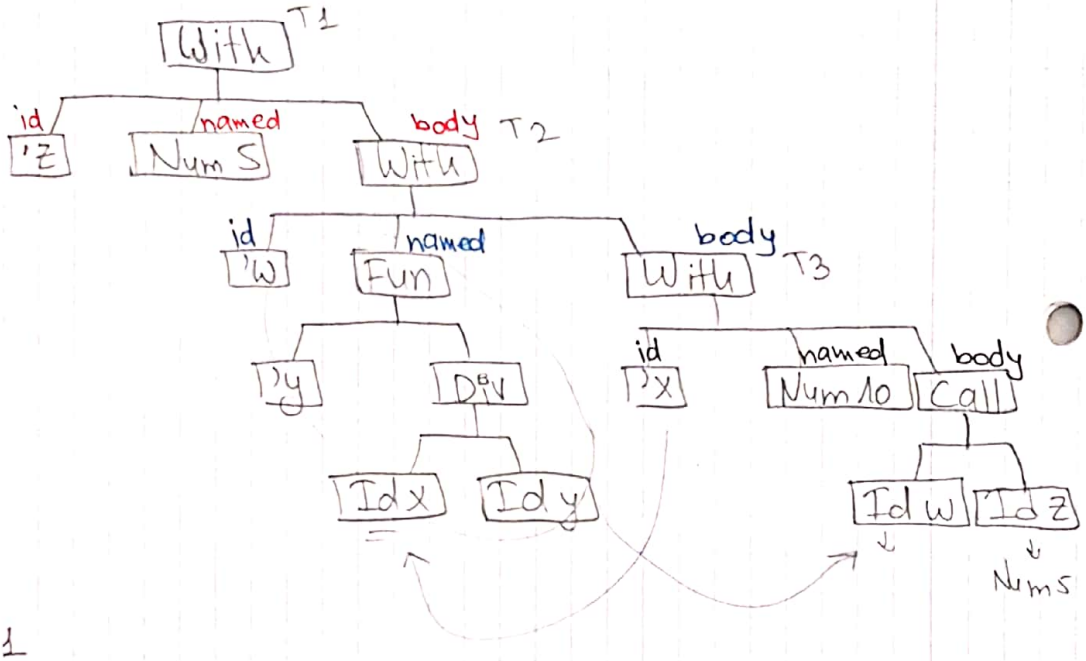
שאלה 2:

(א). ב, ג.

(ב). ד,

שאלה 3:

(א). (שם לפי עדיפות -



AST1: T1

Res1: Res3 = (Num 2)

AST2: (Num 5) eval(named)

Res2: (Num 5)

AST3: T2' . z→Num5 לאחר ההחלפה של T2' -

Res3: Res5 = (Num 2) eval(body)

AST4: (Fun y (Div (Id x) (Id y))) eval(named)

Res4: (Fun y (Div (Id x) (Id y)))

AST5: T3' w→Res4 לאחר ההחלפה של T3' -

Res5: Res7 = (Num 2) eval(body)

AST6: (Num 10) eval(named)

Res6: (Num 10)

eval(body)

AST7: (Call (Fun y (Div (Num 10) (Id y))) (Num 5))

fun arg

Res7: Res 10 = (Num 2)

Eval AST8: (Fun y (Div (Num 10) (Id y)))

Res8: (Fun y (Div (Num 10) (Id y)))

id body

AST9: (Num 5) eval(arg)

Res9: (Num 5)

AST10: (Div (Num 10) (Num 5))

Res 10: (Num 2)

AST 11: (Num 10)

Res 11: (Num 10)

AST 12: (Num 5)

Res12: (Num 5)

AST1: T1 .(P)

Cache1: '()

Res 1: Res3 = (Num 2)

AST2: (Num 5) eval(named)

Cache2: '()

Res2: (Num 5)

AST3: T2 eval(body)

Cache3: '(('z (Num 5)))

→ Res3: Res 5 = (Num 2)

AST4: (Fun y (Div (Id x) (Id y))) } eval(named)

Cache4: '(('z (Num 5)))

Res4: (Fun y (Div (Id x) (Id y)))

AST5: T3 eval(body)

Cache5: '('(w (Fun y (Div (Id x) (Id y)))) '(z (Num 5)))

→ Res5: Res7 = (Num 2)

AST6: (Num 10) eval(named)

Cache6: cache5

Res6: (Num 10)

AST7: (Call (Id w) (Id z)) eval(body)

fun arg

Cache7: '('(x (Num 10)) Cache5)

→ Res7: Res10 = (Num 2)

final AST8: (Id w) eval(fun)

Cache8: Cache7

Res8: (Fun y (Div (Id x) (Id y)))

id body

AST9: (Id z)

Cache9: Cache7

Res9: (Num 5)

AST10: (Div (Id x) (Id y))

10 5

Cache10: ((y (Num 5)) Cache7)

⇒ Res10: (Num 2)

AST11: (Id x)

Cache11: Cache10

Res11: (Num 10)

AST12: (Id y)

Cache12: Cache10

Res12: (Num 5)

(ב). התוצאה תהיה שגיאה- "no binding for x".

משום שעבור - (Fun y (Div (Id x) (Id y))) - נשמה הסביבה שלה, והמשתנה x - מוגדר אחרי כה (בדומה ל מי נמורה בעל) לכן אינו מופיע בסביבה של ה. fun, ולכן לא יהיה לו קישור.

התשובה הרצויה משבלנו - היא השגיאה מאופן - הסביבה, משום שאנו רוצים לעבוד ב- static scoping. ולכן נרצה לשמור את הסוג מהגדרתה, ולא בהרצה שלה.

שאלה 4:

(א).

1. {* <FLANG>...}
2. {/ <FLANG> <FLANG>...}
3. {fun {<id>...} <FLANG>}
4. {Call <FLANG> <FLANG>...}

(ב).

5. (Listof FLANG)
6. FLANG (Listof FLANG)
7. (Listof FLANG)
8. FLANG (Listof FLANG)
9. (Listof Symbol) FLANG)
10. FLANG (Listof FLANG)

(ג).

```
- (: not-member : Symbol (Listof Symbol) -> Boolean)
(define (not-member s slist)
  (if (null? slist) true
      (if (eq? s (first slist)) false
          (not-member s (rest slist)))))
```

- (:all-unique: (Listof Symbol) -> Boolean)

```
(define (all-unique lists)
        (if (null? lists) true
            (if (not-member (first lists) (rest lists))
                (all-unique (rest lists))
                false)))
```

אם כן, הוסף מספר אמשך הלאה ->

אם לא - לא קיים-תחזיר false ->

11. (all-unique names) .(ג)

12. (Fun names (parse-sexpr body))

13. (error 'parse-sexpr "parameters should not repeat in 'fun' syntax")

14. (Sub (parse-sexpr fst) (parse-sexpr* args))

15. [(list '* args...) (Mul (parse-sexpr* args))]

16. [(list '/ fst args...) (Div (parse-sexpr fst) (parse-sexpr* args))]

17. (Call (parse-sexpr fun) (parse-sexpr* args))

18. env .(ד)

19. (extend* (rest names) (rest vals) (Extend (first names) (first vals) env))

20. (NumV (foldl + 0 (map NumV->number (eval* args env))))

21. (NumV (- (NumV->number (eval fst env)) // (foldl + 0 (map NumV->number (eval* args env)))))

(NumV->number (eval (Add args) env))

22. [(Mul args) (NumV (foldl * 1 (map NumV->number (eval* args env))))]

23. [(Div fst args) (NumV (/ (NumV->number (eval fst env)) // (foldl * 1 (map Num->number (eval* args env)))))]

(NumV->number (eval (Mul args) env))

24. [(Id name) (lookup name env)]

25. (FunV bound-ids bound-body env)

26. (= (length arg-exprs) (length bound-ids))

27. (eval bound-body
(extend* bound-ids (eval* arg-exprs env)
f-env))