אוניברסיטת
אריאל
בשומרון

פקולטה: מדעי הטבע

מחלקה: מדעי המחשב ומתמטיקה

שם הקורס: שפות תכנות

קוד הקורס: 2-7036010-1/2

תאריך בחינה: _ 09/07/2013 סמ' _ב'___מועד __ג'__

משך הבחינה: 3 שעות

שם המרצה: ערן עמרי

חומר עזר: דף A4 אחד (ניתן לכתוב משני צדיו)

שימוש במחשבון: לא

הוראות כלליות:

- כתבו את תשובותיכם בכתב קריא ומרווח.
- בכל שאלה או סעיף, ניתן לכתוב – **לא יודע/ת** (מבלי להוסיף דבר מעבר לכך) – ולקבל 20% מהניקוד על השאלה או הסעיף.
- אפשר להסתמך על סעיפים קודמים גם אם לא עניתם עליהם.
- יש לענות על כל השאלות. נא לציין באופן ברור היכן מתחילה התשובה לכל שאלה!!
- ניתן להשיג עד 104 נקודות במבחן.

בהצלחה!

אוניברסיטת
אריאל
בשומרון

שאלה 1 — BNF (23 נקודות):

נתון הדקדוק (BNF) הבא:

```
<TREE> ::= <ATOM> | ( <TREE> <OP> <TREE> )

<ATOM> ::= <num>

<OP>   ::= + | - | / | *
```

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

סעיף א' (5 נקודות):

מהי השפה שמגדיר הדקדוק? כתבו תאור קצר (עד ארבע שורות), אך ברור (הקפידו להשתמש במונחים נכונים).

סעיף ב' (5 נקודות):

ציירו עץ גזירה עבור מילה תוך שימוש באורך לפחות 10 תווים (ציינו בצורה מפורשת מהי המילה אותה גזרתם).

סעיף ג' (5 נקודות):

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

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'.

שאלה 2 — (33 נקודות) FLANG:

לצורך פתרון שאלה זו מצורף קוד ה- interpreter של FLANG (במודל ה-substitution) בסוף טופס המבחן.

נתון הקוד הבא:

```
(run "{with {foo1 {fun {a} {fun {a} {call a b}}}}
                {with {b 5}
                   {call {call foo1 {fun {x} x}}
                         {fun {a} {+ a 4}}}}}")
```

סעיף א' (8 נקודות):

ציירו את עץ התחביר האבסטרקטי המתאר את הביטוי הנתון במרכאות (כלומר את התוצאה של הפעלת parse על ביטוי זה).
דוגמא: העץ המתאר את הביטוי `{2 1 +}` הוא:

סעיף ב' (15 נקודות):

בתהליך ההערכה של ביטוי זה (הפונקציה eval) תתבצענה חמש פעולות החלפה (הפונקציה subst) – בפעולה כזו מוחלף קדקוד בעץ (שנוצר עם בנאי Id) בעץ אחר. ציירו את העץ המתקבל לאחר כל פעולת החלפה כזו (ציירו לפחות חמישה עצים בסעיף זה לפי סדר הופעתם בחישוב).

סעיף ג' (2 נקודות):

מהי תוצאת החישוב של הביטוי כולו (על-פי קוד ה- interpreter של FLANG (במודל ה-substitution)?

סעיף ד' (8 נקודות):

אוניברסיטת
אריאל
בשומרון

הסבירו מה הבעיה בתשובתכם לסעיף ג' ומה צריך לקרות כאשר אנו דנים ב – Lexical scoping. לשימושכם מצורף חלק הקוד הרלוונטי ה- interpreter של FLANG (במודל ה- סביבות) המחשב על-פי Lexical scoping.

```
;; The Flang interpreter, using environments
;; Types for environments, values, and a lookup function
  (define-type ENV
    [EmptyEnv]
    [Extend Symbol VAL ENV])

  (define-type VAL
    [NumV Number]
    [FunV 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))]))

  (: eval : FLANG ENV -> VAL)
  ;; evaluates FLANG expressions by reducing them to values
  (define (eval expr env)
    (cases expr
      [(Num n) (NumV n)]
      [(Add l r) (arith-op + (eval l env) (eval r env))]
      [(Sub l r) (arith-op - (eval l env) (eval r env))]
      [(Mul l r) (arith-op * (eval l env) (eval r env))]
      [(Div l r) (arith-op / (eval l env) (eval r env))]
      [(With bound-id named-expr bound-body)
       (eval bound-body
             (Extend bound-id (eval named-expr env) env))]
      [(Id name) (lookup name env)]
      [(Fun bound-id bound-body)
       (FunV bound-id bound-body env)]
      [(Call fun-expr arg-expr)
       (let ([fval (eval fun-expr env)])
         (cases fval
           [(FunV bound-id bound-body f-env)
            (eval bound-body
                  (Extend bound-id (eval arg-expr env)
f-env))])
```

```
                [else (error 'eval "`call' expects a function, got:
~s"
                                  fval)]))]))
```

שאלה 3 — הרחבת השפה 26) — **WAE** נקודות):

נרצה להרחיב את השפה **WAE** ולאפשר מספר משתנה של ארגומנטים לאופרטורים האריתמטיים. 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 "{/}")  =error> "bad syntax")
```

לצורך כך נרחיב את הדקדוק באופן הבא:

```
#| BNF for the WAE language:
      <WAE> ::= <num>
              | { + <WAE> ... }
              | { - <WAE> <WAE> ...}
              | { * <WAE> ... }
              | { / <WAE> <WAE> ...}
              | { with { <id> <WAE> } <WAE> }
              | <id>
  |#
```

סעיף א' (3 נקודות):

הוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים – 3 שורות קוד סה"כ לסעיף זה) ל –

```
(define-type WAE
    [Num  Number]
    [Add  (Listof WAE)]
```

```
    [Sub  —«fill-in 01»—]
    [Mul  —«fill-in 02»—]
    [Div  —«fill-in 03»—]
    [Id   Symbol]
    [With Symbol WAE WAE])
```

<u>סעיף ב' (3 נקודות):</u>

השתמשו בקוד הבא –

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

והוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים – 4 שורות קוד סה"כ לסעיף זה) ל –

```
(: parse-sexpr : Sexpr -> WAE)
  ;; to convert s-expressions into WAEs
  (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)])]
      [—«fill-in 04»—]
      [—«fill-in 05»—]
      [—«fill-in 06»—]
      [—«fill-in 07»—]
      [else (error 'parse-sexpr "bad syntax in ~s"
sexpr)]))
```

<u>סעיף ג' (8 נקודות):</u>

השתמשו בהגדרות הפורמליות הבאות –

```
#| Formal specs for `subst':
   (`N' is a <num>, `E1', `E2' are <WAE>s, `x' is some <id>,
`y' is a *different* <id>)
        N[v/x]                  = N
        {+ E ...}[v/x]          = {+ E[v/x] ...]}
        {- E1 E ...}[v/x]          = {- E1[v/x] E[v/x] ...}
        {* E ...}[v/x]          = {* E[v/x] ...}
        {/ E1 E ...}[v/x]          = {/ E1[v/x] E[v/x] ...}
        y[v/x]                  = y
        x[v/x]                  = v
        {with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]}
        {with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}
 |#
```

והוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים – 4 שורות קוד סה"כ לסעיף זה) ל –

```
(: subst : WAE Symbol WAE -> WAE)
  ;; substitutes the second argument with the third argument
  ;; in the first argument, as per the rules of substitution
  ;; the resulting expression contains no free instances of
  ;; the second argument
  (define (subst expr from to)
    ;; convenient helper -- no need to specify `from' and `to'
    (: subst-helper : WAE -> WAE)
    (define (subst-helper x) (subst x from to))
    ;; helper to substitute lists
    (: subst* : (Listof WAE) -> (Listof WAE))
    (define (subst* exprs) (map subst-helper exprs))
    (cases expr
      [(Num n) expr]
      [(Add args) —«fill-in 08»—]
```

```
      [(Sub fst args) —«fill-in 09»—]
      [(Mul args) —«fill-in 10»—]
      [(Div fst args) —«fill-in 11»—]
      [(Id name) (if (eq? name from) to expr)]
      [(With bound-id named-expr bound-body)
       (With bound-id
             (subst named-expr from to)
             (if (eq? bound-id from)
               bound-body
               (subst bound-body from to)))]))
```

<u>סעיף ד' (12 נקודות):</u>

השתמשו בהגדרות הפורמליות הבאות –

```
#| Formal specs for `eval':
       eval(N)          = N
       eval({+ E ...}) = eval(E) + ...
       eval({- E1 E ...}) = eval(E1) - (eval(E) + ...)
       eval({* E ...}) = eval(E) * ...
       eval({/ E1 E ...}) = eval(E1) / (eval(E) * ...)
       eval(id)         = error!
       eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
  |#
```

והוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים – 4 שורות קוד סה"כ לסעיף זה) ל –

```
(: eval : WAE -> Number)
  ;; evaluates WAE expressions by reducing them to numbers
  (define (eval expr)
    (cases expr
      [(Num n) n]
      [—«fill-in 12»—]
      [—«fill-in 13»—]
      [—«fill-in 14»—]
      [—«fill-in 15»—]
      [(With bound-id named-expr bound-body)
       (eval (subst bound-body
                    bound-id
                    (Num (eval named-expr))))]
      [(Id name) (error 'eval "free identifier: ~s" name)]))
```

הדרכה: השתמשו בפונקציות map ו- foldl של RACKET המתוארות מטה.
במקרה של חלוקה באפס יש לתת הודעת שגיאה.

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))
```

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 כערך 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)
```

שאלה 4 — (22 נקודות):

נתון הקוד הבא:

```
#lang pl

(define-type BOX
  [Ebox]
  [Fbox Number BOX])

(: f : BOX -> BOX)

(define (f box)
  (foo box (Ebox)))

(: foo : BOX  BOX -> BOX)

(define (foo boxa boxb)
  (cases boxa
    [(Ebox) boxb]
    [(Fbox v n) (let ((B (foo n (Ebox))))
                  (foo1 B (Fbox v (Ebox))))]))

(: foo1 : BOX  BOX -> BOX)
```

```
(define (foo1 boxa boxb)
  (cases boxa
    [(Ebox) boxb]
    [(Fbox v n) (Fbox v (foo1 n boxb))]))
```

סעיף א' (6 נקודות):

הריצו את הקוד הבא:

```
(f (Fbox 4 (Fbox 3 (Fbox 2 (Fbox 1 (Ebox))))))
```

תארו את שלבי הריצה. מהו הערך המוחזר?

סעיף ב' (4 נקודות):

מה עושה הפונקציה f באופן כללי?

סעיף ג' (5 נקודות):

האם הקריאה הרקורסיבית הינה קריאת זנב (tail-recursion)? הסבירו את תשובתכם.

סעיף ד' (7 נקודות):

כתבו פונקציה foo* שתחזיר תמיד את אותו ערך כמו foo באופן הבא: אם תשובתכם בסעיף הקודם הייתה שלילית – על foo* להשתמש בקריאות זנב בלבד. אחרת – על foo* להשתמש גם בקריאות שאינן קריאות זנב. (ניתן להגדיר פונקציות עזר).

```
---<<<FLANG>>>-------------------------------------------------------

  ;; The Flang interpreter

  #lang pl

  #|
  The grammar:
    <FLANG> ::= <num>
              | { + <FLANG> <FLANG> }
```

```
            | { - <FLANG> <FLANG> }
            | { * <FLANG> <FLANG> }
            | { / <FLANG> <FLANG> }
            | { with { <id> <FLANG> } <FLANG> }
            | <id>
            | { fun { <id> } <FLANG> }
            | { call <FLANG> <FLANG> }

Evaluation rules:

  subst:
    N[v/x]                  = N
    {+ E1 E2}[v/x]          = {+ E1[v/x] E2[v/x]}
    {- E1 E2}[v/x]          = {- E1[v/x] E2[v/x]}
    {* E1 E2}[v/x]          = {* E1[v/x] E2[v/x]}
    {/ E1 E2}[v/x]          = {/ E1[v/x] E2[v/x]}
    y[v/x]                  = y
    x[v/x]                  = v
    {with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]} ; if y =/= x
    {with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}
    {call E1 E2}[v/x]     = {call E1[v/x] E2[v/x]}
    {fun {y} E}[v/x]      = {fun {y} E[v/x]}            ; if y =/= x
    {fun {x} E}[v/x]      = {fun {x} E}

  eval:
    eval(N)             = N
    eval({+ E1 E2})     = eval(E1) + eval(E2)   \ if both E1 and E2
    eval({- E1 E2})     = eval(E1) - eval(E2)    \ evaluate to numbers
    eval({* E1 E2})     = eval(E1) * eval(E2)    / otherwise error!
    eval({/ E1 E2})     = eval(E1) / eval(E2)   /
    eval(id)            = error!
    eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
    eval(FUN)           = FUN ; assuming FUN is a function expression
    eval({call E1 E2}) = eval(Ef[eval(E2)/x]) if eval(E1)={fun {x} Ef}
                        = error!                otherwise
|#

(define-type FLANG
  [Num  Number]
  [Add  FLANG FLANG]
  [Sub  FLANG FLANG]
  [Mul  FLANG FLANG]
  [Div  FLANG FLANG]
  [Id   Symbol]
  [With Symbol FLANG FLANG]
  [Fun  Symbol FLANG]
  [Call FLANG FLANG])

(: 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: name)) body)
         (Fun name (parse-sexpr body))]
        [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list 'call fun arg) (Call (parse-sexpr fun) (parse-sexpr arg))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

(: parse : String -> FLANG)
;; parses a string containing a FLANG expression to a FLANG AST
(define (parse str)
  (parse-sexpr (string->sexpr str)))

(: subst : FLANG Symbol FLANG -> FLANG)
;; substitutes the second argument with the third argument in the
;; first argument, as per the rules of substitution; the resulting
;; expression contains no free instances of the second argument
(define (subst expr from to)
  (cases expr
    [(Num n) expr]
    [(Add l r) (Add (subst l from to) (subst r from to))]
    [(Sub l r) (Sub (subst l from to) (subst r from to))]
    [(Mul l r) (Mul (subst l from to) (subst r from to))]
    [(Div l r) (Div (subst l from to) (subst r from to))]
    [(Id name) (if (eq? name from) to expr)]
    [(With bound-id named-expr bound-body)
     (With bound-id
           (subst named-expr from to)
           (if (eq? bound-id from)
             bound-body
             (subst bound-body from to)))]
    [(Call l r) (Call (subst l from to) (subst r from to))]
    [(Fun bound-id bound-body)
     (if (eq? bound-id from)
       expr
       (Fun bound-id (subst bound-body from to)))]))

(: arith-op : (Number Number -> Number) FLANG FLANG -> FLANG)
;; gets a Racket numeric binary operator, and uses it within a FLANG
;; `Num' wrapper
(define (arith-op op expr1 expr2)
  (: Num->number : FLANG -> Number)
  (define (Num->number e)
    (cases e
      [(Num n) n]
      [else (error 'arith-op "expects a number, got: ~s" e)]))
  (Num (op (Num->number expr1) (Num->number expr2))))
```

```
(: eval : FLANG -> FLANG)
;; evaluates FLANG expressions by reducing them to *expressions*
(define (eval expr)
  (cases expr
    [(Num n) expr]
    [(Add l r) (arith-op + (eval l) (eval r))]
    [(Sub l r) (arith-op - (eval l) (eval r))]
    [(Mul l r) (arith-op * (eval l) (eval r))]
    [(Div l r) (arith-op / (eval l) (eval r))]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (eval named-expr)))]
    [(Id name) (error 'eval "free identifier: ~s" name)]
    [(Fun bound-id bound-body) expr]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr)])
       (cases fval
         [(Fun bound-id bound-body)
          (eval (subst bound-body
                       bound-id
                       (eval arg-expr)))]
         [else (error 'eval "`call' expects a function, got: ~s"
                      fval)]))]))

(: run : String -> Number)
;; evaluate a FLANG program contained in a string
(define (run str)
  (let ([result (eval (parse str))])
    (cases result
      [(Num n) n]
      [else (error 'run
                   "evaluation returned a non-number: ~s" result)])))

;; tests
(test (run "{call {fun {x} {+ x 1}} 4}")
      => 5)
(test (run "{with {add3 {fun {x} {+ x 3}}}
              {call add3 1}}")
      => 4)
(test (run "{with {add3 {fun {x} {+ x 3}}}
              {with {add1 {fun {x} {+ x 1}}}
                {with {x 3}
                  {call add1 {call add3 x}}}}}")
      => 7)
(test (run "{with {identity {fun {x} x}}
              {with {foo {fun {x} {+ x 1}}}
                {call {call identity foo} 123}}}")
      => 124)
(test (run "{call {call {fun {x} {call x 1}}
                        {fun {x} {fun {y} {+ x y}}}}
                  123}")
      => 124)
```