פקולטה: מדעי הטבע

מחלקה: מדעי המחשב ומתמטיקה

שם הקורס: שפות תכנות

קוד הקורס: 2-7036010

תאריך בחינה: _17/07/2016_ סמ' _ב'___מועד __ב'__

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 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.
- ניתן להשיג עד 109 נקודות במבחן.
- לנוחיותכם מצורפים שני קטעי קוד עבור ה- interpreter של FLANG בסוף טופס המבחן. הראשון – במודל הסביבות והשני – במודל ה-Substitution-cache .

שאלה 1 — BNF — (20 נקודות):

סעיף א' (6 נקודות):

כתבו דקדוק עבור שפת המחרוזות הסופיות (באורך גדול מ-0) של אפסים ואחדות. להלן דוגמאות למילים חוקיות בשפה. הראו גזירה למילה באורך ארבעה תווים.

```
1 0 0 0
1 0 1 0 1 0 1 0
1 1
```

סעיף ב' (7 נקודות):

הסבירו את מושג ה-ambiguity **ביחס לדקדוק שהגדרתם. בפרט הראו שני דקדוקים לאותה שפה (של סעיף א). אחד שמתקיימת בו התכונה ואחר שאינה מתקיימת בו. הסבירו את תשובתכם.**

סעיף ג' (7 נקודות):

הסבירו את מושג ה-compositionality **ביחס לדקדוק שהגדרתם. בפרט, הראו בחירה של סמנטיקה ושני דקדוקים לאותה שפה (של סעיף א). אחד שמתקיימת בו התכונה ואחר שאינה מתקיימת בו. הסבירו את תשובתכם.**

שאלה 2 — With vs. Call — (20 נקודות):

סעיף א' (10 נקודות):

נתון הקוד הבא:

```
(run "{with { x {* 3 4}}
          {with {foo {fun {y} {- x y}}}
             {call foo 2}}}")
```

החליפו את הקוד הנ"ל בקוד שקול בו לא מופיעה המילה with **– לצורך כך השתמשו במילה** call.

סעיף ב' (10 נקודות):

סטודנט מחק את הווריאנט (הבנאי) With של הטיפוס FLANG באינטרפרטר (במודל הסביבות) ואת כל שורות הקוד שהשתמשו בבנאי With. הכלילו את הרעיון מסעיף א', כדי לדאוג שהאינטרפטר ימשיך לפעול כמצופה, גם ללא הבנאי With (אסור שהממשק מול מתכנת בשפה FLANG ישתנה).

הדרכה: **החליפו את השורות המתאימות בפרוצדורה `parse-sexpr` – בשורות קוד שאינן מכילות הפעלה של בנאי With (אלא בנאים חלופיים קיימים). עליכם לכתוב במחברת רק את שורות הקוד המקוריות שבחרתם לשנות ומתחתיהן, את שורות הקוד החלופיות.**

שאלה 3 — (37 נקודות): **נתון הקוד הבא:**

```
(run "{with {foo {fun {x} {* x x}}}
         {call {with {foo {fun {y} {- y 1}}}
                  {fun {x} {call foo x}}}
               4}}")
```

סעיף א' (15 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)

AST_1 = `(With x (Num 1) (Add (Id x) (Num 2)))`
$Cache_1$ = `'()`
RES_1 = `(Num 3)`
AST_2 = `(Num 1)`
$Cache_2$ = `'()`
RES_2 = `(Num 1)`
AST_3 = `(Add (Id x) (Num 2))`
$Cache_3$ = `'((x (Num 1))`
RES_3 = `(Num 3)`
AST_4 = `(Id x)`
$Cache_4$ = `'(x (Num 1))`
RES_4 = `(Num 1)`
AST_5 = `(Num 2)`
$Cache_5$ = `'((x (Num 1))`
RES_5 = `(Num 2)`
`Final result: 3`

סעיף ב' (7 נקודות):

מה היה קורה לו היינו מבצעים את ההערכה במודל הסביבות? מהי התשובה הרצויה? מדוע? (אין צורך לבצע הערכה). תשובה מלאה לסעיף זה לא תהיה ארוכה משלוש שורות (תשובה ארוכה מדי תקרא חלקית).

סעיף ג' (15 נקודות):

האינטרפרטר שכתבנו במודל ה-substitution-cache מממש dynamic-scoping **ואילו האינטרפרטר שכתבנו במודל הסביבות מממש** lexical-scoping. **בשאלה זאת, עליכם לשנות את הקוד של האינטרפרטר שכתבנו במודל הסביבות (העליון מבין השניים המופיעים מטה) – על מנת להמירו למימוש של** dynamic-scoping. **עליכם לשנות מעט ככל שתוכלו מהקוד.**

העתיקו למחברת רק את שורות הקוד שברצונכם לשנות וציינו בבירור מהו השינוי הנדרש. ככל שתשובתכם תכלול פחות שינויים לקוד המקורי, היא תזכה אתכם ביותר ניקוד. הסבירו בבירור מדוע השינויים שביצעתם נדרשים ומדוע הם מספיקים. תשובה ללא הסבר, לא תזכה אתכם בניקוד.

רמז: תשובה טובה תהיה קצרה מאד.

שאלה 4 — הרחבת השפה FLANG **מודל הסביבות — (32 נקודות):**

לצורך פתרון שאלה זו נעזר בקוד ה- interpreter **של** FLANG **במודל הסביבות, המופיע בסוף טופס המבחן (העליון מבין השניים המופיעים שם).**

נרצה להרחיב את השפה FLANG ולאפשר חישוב לולאות for **עם פעולת חיבור או עם פעולת כפל.**

כלומר, נאפשר חזרה על קטע קוד, עם מונה (counter**), כך שבתוך קטע הקוד מותר להשתמש בערך המונה (קטע הקוד עשוי להיות כל ביטוי על פי התחביר בשפה). הערך המוחזר מחישוב כל הלולאה יהיה סכום כל הערכים המוחזרים מכל החישובים בכל האיטרציות של הלולאה (אם הביטוי משתמש באופרטור +, ראו דוגמאות מטה) או מכפלת כל הערכים הללו (אם הביטוי משתמש באופרטור *).**

להלן דוגמה לביטוי אפשרי:

```
"{ for + { i } = 1 to 12 do {+ 1 i} }"
```

כאן, i **הוא שם המונה, 1 הוא ערך התחלתי למונה ו-12 הוא גבול עליון, {+ 1 i} הוא גוף הלולאה והפעולה היא פעולת החיבור (זאת על-פי השימוש באופרטור +). לשם פשטות, בהמשך ניתן תמיד להניח שערך הגבול העליון למונה גדול ממש מן הערך ההתחלתי עבור המונה וכי שניהם מספרים שלמים.**

להלן דוגמאות לטסטים שאמורים לעבוד:

```
(test (run "{ for + { i } = 1 to 12 do 1 }")
        => 12)
```

```
(test (run "{ for + { i } = {+ 1 0} to 5 do i }")
        => 15)
(test (run "{with {x { for + { i } = 1 to 5 do i }}
                {- x 10}}")
        => 5)

(test (run "{ for * { i } = 1 to 4 do i }")
        => 24)

(test (run
    "{with {factorial {fun {n} { for * { i } = 1 to n do i }}}
         {call factorial 4}}")
      => 24)

(test (run "{ for + { i } = {with {x {- 5 3}} x} to {- 7 4} do
           {with {sqr {fun {x} {* x x}}}
              {call sqr i}}}")
      => 13)
(test (run
     "{ + 5 { for + { i } = {with {x {- 5 3}} x} to {- 7 4} do
                {with {sqr {fun {x} {* x x}}}
                   {call sqr i}}}}")
      => 18)

(test (run "{ for - { i } = 1 to 12 do 1 }")
        =error> "parse-sexpr: bad `for' syntax in")
```

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

<u>סעיף א' (הרחבת הדקדוק) (6 נקודות):</u>

כתבו מהם שני כללי הדקדוק שיש להוסיף לדקדוק הקיים.

```
—«fill-in 1»—

—«fill-in 2»—
```

<u>סעיף ב' (הרחבת הטיפוס FLANG) (5 נקודות):</u>

הוסיפו את הקוד הנדרש (יש להוסיף בנאי יחיד) ל –

```
(define-type FLANG
…
[For —«fill-in 3»—]
…)
```

הדרכה: הארגומנט הראשון צריך להיות זה שיגדיר את הפעולה המתאימה (חיבור או כפל).

סעיף ג' (parsing) (10 נקודות):

השתמשו בדוגמאות הטסטים מעלה כדי להבין אילו הודעות שגיאה רצויות. הוסיפו 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
      ….
      [(cons –«fill-in 4»–)
       (match –«fill-in 5»–
         [(list 'for –«fill-in 6»–)
           –«fill-in 7»–]
         [(list 'for –«fill-in 8»–)
           –«fill-in 9»–]
         [else –«fill-in 10»–])]
      ….
      [else (error 'parse-sexpr "bad syntax in ~s"
         sexpr)]))
```

סעיף ד' (evaluation) (11 נקודות):

בסעיף זה נשלים את הקוד שיאפשר הערכה של ביטויי לולאה כפי שהגדרנו בסעיפים הקודמים.

ראשית, נהפוך את הפרוצדורה NumV->number **זמינה ומוכרת (עד כה הייתה פנימית ל -** arith-op**).**

```
 (: NumV->number : VAL -> Number)
  (define (NumV->number v)
    (cases v
      [(NumV n) n]
      [else (error ' NumV->number "expected a number, got: ~s" v)]))
(: arith-op : (Number Number -> Number) VAL VAL -> VAL)
  ;; gets a Racket numeric binary operator,
  ;; and uses it within a NumV wrapper
  (define (arith-op op val1 val2)
    (NumV (op (NumV->number val1) (NumV->number val2))))
```

עתה נכתוב פונקציה שתקרא ע"י eval **ותבצע את האיטרציות של הלולאה והחלת הפעולה האריתמטית על כל תוצאות החישובים האיטרטיביים. השלימו את הקוד החסר:**

```
(: eval-for-loop : (Number Number -> Number) Symbol FLANG
FLANG FLANG ENV -> VAL)
 (define (eval-for-loop op cnt-name from-exp to-exp body env)
   (: loop-eval :  Number Number -> VAL)
   (define (loop-eval from to)
     (if —«fill-in 11»—
         (eval —«fill-in 12»—)
         (arith-op —«fill-in 13»—)
   (loop-eval —«fill-in 14»—))
```

הדרכה: כדי להשלים את #11 – חישבו מהו תנאי העצירה המתאים במקרה שאמורה להיות איטרציה אחת. כדי להשלים את #12 – זכרו שבתוך גוף הלולאה ייתכנו מופעים חופשיים של שם המונה.

הערה: לשם פשטות, ניתן להניח שערך הגבול העליון למונה גדול ממש מן הערך ההתחלתי עבור המונה וכי שניהם מספרים שלמים.

לבסוף, השלימו את שורת הקוד המתאימה בפונקציה eval:

```
(: eval : FLANG ENV -> VAL)
  ;; evaluates FLANG expressions by reducing them to values
  (define (eval expr env)
    (cases expr
     …
      [(For —«fill-in 15»—)
       (—«fill-in 16»—)]))
```

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

  ;; The Flang interpreter, using environments

  #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:
    eval(N,env)                 = N
    eval({+ E1 E2},env)         = eval(E1,env) + eval(E2,env)
    eval({- E1 E2},env)         = eval(E1,env) - eval(E2,env)
    eval({* E1 E2},env)         = eval(E1,env) * eval(E2,env)
```

```
  eval({/ E1 E2},env)         = eval(E1,env) / eval(E2,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 E2},env1)
           = eval(Ef,extend(x,eval(E2,env1),env2))
                              if eval(E1,env1) = <{fun {x} Ef}, env2>
           = 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)))

;; 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))]))

(: arith-op : (Number Number -> Number) VAL VAL -> VAL)
;; gets a Racket numeric binary operator, and uses it within a NumV
;; wrapper
(define (arith-op op val1 val2)
  (: NumV->number : VAL -> Number)
  (define (NumV->number v)
    (cases v
      [(NumV n) n]
      [else (error 'arith-op "expects a number, got: ~s" v)]))
  (NumV (op (NumV->number val1) (NumV->number val2))))

(: 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)]))]))

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

```
--<<<FLANG-Substitution-cache>>>-----------------------------------------

  ;; The Flang interpreter, using Substitution-cache

#lang pl

  (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)))

  ;; a type for substitution caches:
  (define-type SubstCache = (Listof (List Symbol FLANG)))

  (: empty-subst : SubstCache)
  (define empty-subst null)

  (: extend : Symbol FLANG SubstCache -> SubstCache)
```

```
(define (extend name val sc)
  (cons (list name val) sc))

(: lookup : Symbol SubstCache -> FLANG)
(define (lookup name sc)
  (let ([cell (assq name sc)])
    (if cell
      (second cell)
      (error 'lookup "no binding for ~s" name))))

(: 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 SubstCache -> FLANG)
;; evaluates FLANG expressions by reducing them to expressions
(define (eval expr sc)
  (cases expr
    [(Num n) expr]
    [(Add l r) (arith-op + (eval l sc) (eval r sc))]
    [(Sub l r) (arith-op - (eval l sc) (eval r sc))]
    [(Mul l r) (arith-op * (eval l sc) (eval r sc))]
    [(Div l r) (arith-op / (eval l sc) (eval r sc))]
    [(With bound-id named-expr bound-body)
     (eval bound-body
           (extend bound-id (eval named-expr sc) sc))]
    [(Id name) (lookup name sc)]
    [(Fun bound-id bound-body) expr]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr sc)])
       (cases fval
         [(Fun bound-id bound-body)
          (eval bound-body
                (extend bound-id (eval arg-expr sc) sc))]
         [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) empty-subst)])
    (cases result
      [(Num n) n]
      [else (error 'run
                   "evaluation returned a non-number: ~s" result)])))
```