אוניברסיטת
אריאל
בשומרון

פקולטה: מדעי הטבע

מחלקה: מדעי המחשב ומתמטיקה

שם הקורס: שפות תכנות

קוד הקורס: 7036010

תאריך בחינה: 23/07/2018 סמ' ב' מועד ב'

משך הבחינה: שעתיים ורבע

שם המרצה: ערן עמרי

חומר עזר: אסור

שימוש במחשבון: לא

הוראות כלליות:

- **קראו היטב את הוראות המבחן.**
- 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 (אין זה תקף לחלקי סעיף).
- אפשר להסתמך על סעיפים קודמים גם אם לא עניתם עליהם.
- יש לענות על כל השאלות.
- בכל מקום הדורש תכנות, מותר להגדיר פרוצדורות עזר (אלא אם נאמר אחרת במפורש).
- ניתן להשיג עד 105 נקודות במבחן.
- לנוחיותכם מצורפים שני קטעי קוד עבור ה- interpreter של FLANG בסוף טופס המבחן. הראשון – במודל הסביבות והשני – במודל ה-substitution cache.

בהצלחה!

אוניברסיטת
אריאל
בשומרון

שאלה 1 — מופעים חופשיים של שמות מזהים — (30 נקודות):

קוד בשפה שכתבנו FLANG המכיל מופעים חופשיים של שמות מזהים הוא קוד שגוי – עליו יש להוציא הודעת שגיאה. באינטרפרטר שכתבנו במודל ההחלפות, הודעת השגיאה ניתנת בזמן הערכת התוכנית. למעשה, שגיאה כזאת יכולה להתגלות עוד לפני הפעלת ה-eval.

בשאלה זו תכתבו פונקציה בוליאנית CFI? (containsFreeInstance? קיצור עבור) המקבלת ביטוי (תכנית) בצורת FLANG ומחזירה true אם ורק אם הביטוי מכיל מופעים חופשיים של שמות מזהים – כך שהפונקציה אינה מבצעת שום הערכה סמנטית של התכנית.

בסעיפים הבאים תכתבו קוד כתוספת לאינטרפרטר של FLANG במודל ההחלפות. הניחו, אם כן, כי כל הפונקציות והטיפוסים שבו מוגדרים לכם.

סעיף א' – CFI? – (20 נקודות):

בסעיף זה תכתבו פונקציה בוליאנית CFI?.

הקוד הבא מבוסס על הפונקציה eval (באינטרפרטר של FLANG במודל ההחלפות – הקוד עבור פונקציה זאת ועבור subst מופיע בתחתית טופס המבחן) – השלימו את הקוד במקומות החסרים.

```
(: CFI? : FLANG -> Boolean)
;; Scans FLANG expressions for free instances of identifiers
(define (CFI? expr)
    (cases expr
      [(Num n) —«fill-in 1»— ]
      [(Add l r) (or —«fill-in 2»—)]
      [—«fill-in 3»— ]
      [—«fill-in 4»—]
      [—«fill-in 5»—]
      [(With bound-id named-expr bound-body)
         (or —«fill-in 6»—)]
      [(Id name) —«fill-in 7»—]
      [(Fun bound-id bound-body) —«fill-in 8»—]
      [(Call fun-expr arg-expr) —«fill-in 9»—]))
```

הדרכה:

1. **תפקידכם לחפש מופעים חופשיים ולא להעריך את הביטוי. כך, הטיפול בכל הבנאים יהיה שונה מהטיפול ב-eval. אל שפעילו את eval עצמה בקוד שלכם.**
2. **כשאתם מחליטים כיצד יש לטפל בבנאי מסוים, מומלץ לצייר את מבנהו ולשים לב לכל תת ביטוי שעשוי להכיל מופעים חופשיים. מעבר לכך, יתכן שצריך לבצע פעולה כלשהי על תת הביטוי לפני שבודקים אותו.**

בדקו את עצמכם: אם נגדיר את פונקציית המעטפת הבאה –

```
(: check-code : String -> Boolean)
(define (check-code str)
  (CFI? (parse str)))
```

כל הטסטים הבאים צריכים לעבוד –

```
;tests
(test (check-code "z") => #t)
(test (check-code "{call {fun {x} {/ x 0}} 4}") => #f)
(test (check-code "{call {fun {y} {+ x 0}} 4}") => #t)
(test (check-code "{fun {y} {- x 1}}") => #t)
(test (check-code "{with {foo {fun {y} {- y 1}}
                     {call foo c}}") => #t)
(test (check-code "{call foo 4}") => #t)
                 (test (check-code "{fun {x} {+ x {/ 5 0}}}") => #f)
```

<u>סעיף ב' – (10 נקודות):</u> תארו מה יקרה בהרצת הקוד הבא – על פי הבניה שלכם עבור הפונקציה CFI? בסעיף א'. האם התוצאה המתקבלת הינה התוצאה הרצויה?

```
(check-code "{with {foo {fun {x} {+ x y}}}
                    {with {y 4}
                      {call foo 2}}}")
```

<u>שאלה 2 — שאלות כלליות — (18 נקודות):</u>

<u>סעיף א':</u> ביטוי cond הוא syntactic sugar בשפה ראקט.

מהו syntactic sugar באופן כללי? הסבירו מה הסיבה להוסיפו לשפה. מה הוא אינו נותן? הדגימו דבריכם לגבי ביטויי cond. (כתבו שלוש שורות לכל היותר!)

<u>סעיף ב':</u>

מהו דקדוק דו-משמעי? הסבירו מה הבעיה בדקדוק כזה ביחס לשפת תכנות. תנו דוגמה קצרה (אין צורך בדוגמה מלאה, ניתן להזכיר דוגמאות שנתנו בקורס).

(כתבו שלוש שורות לכל היותר, מעבר לדוגמה.)

<u>סעיף ג':</u>

הסבירו מדוע החלטנו לזנוח את המימוש של האינטרפרטר במודל ההחלפות. מה הייתה הבעיה? האם היא נפתרה במודל הסביבות? הסבירו. (כתבו שלוש שורות לכל היותר.)

<u>שאלה 3 — (27 נקודות):</u>

נתון הקוד הבא:

```
(run "{with {foo {fun {z} y}}
         {call {fun {y} {call foo 4}}
               7}}")
```

אוניברסיטת
אריאל
בשומרון

סעיף א' (17 נקודות):

נתון התיאור החסר הבא עבור הפעלות הפונקציה **eval** בתהליך ההערכה של הקוד מעלה במודל ה-**substitution-cache** (על-פי ה-interpreter התחתון מבין השניים המצורפים מטה). השלימו את המקומות החסרים - 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 שהשלמתם (אין צורך להסביר מעברים קיימים). ציינו מהי התוצאה הסופית.

שימו לב: ישנן השלמות עם אותו שם – מה שמצביע על השלמה זהה – אין צורך להשלים מספר פעמים, אך ודאו כי מספרתם נכון את ההשלמות במחברת.

AST_1 = (With 'foo (Fun 'z (Id 'y)) (Call (Fun 'y (Call (Id 'foo) (Num 4))) (Num 7)))
$Cache_1$ = '()
RES_1 = <--fill in 1-->
AST_2 = (Fun 'z (Id 'y))
$Cache_2$ = '()
RES_2 = <--fill in 2-->
AST_3 = (Call (Fun 'y (Call (Id 'foo) (Num 4))) (Num 7))
$Cache_3$ = <--fill in 3-->
RES_3 = <--fill in 1-->
AST_4 = (Fun 'y (Call (Id 'foo) (Num 4)))
$Cache_4$ = <--fill in 3-->
RES_4 = <--fill in 4-->
AST_5 = (Num 7)
$Cache_5$ = <--fill in 3-->
RES_5 = (Num 7)
AST_6 = (Call (Id 'foo) (Num 4))
$Cache_6$ = <--fill in 5-->
RES_6 = <--fill in 1-->
AST_7 = (Id 'foo)
$Cache_7$ = <--fill in 5-->
RES_7 = <--fill in 6-->
AST_8 = (Num 4)
$Cache_8$ = <--fill in 5-->
RES_8 = (Num 4)
AST_9 = <--fill in 7-->
$Cache_9$ = <--fill in 8-->
RES_9 = <--fill in 1-->

Final result: ?

סעיף ב' (10 נקודות):

הסבירו מה היה קורה אם היינו מבצעים הערכה של הקוד מעלה במודל ההחלפות – אין צורך בחישוב מלא (הסבירו). מהי התשובה שעליה החלטנו בקורס כרצויה? מדוע?

הסבירו כיצד קשורה הפונקציה שכתבתם בשאלה 1 לקוד הנתון וכיצד ניתן להשתמש בפונקציה זאת במודל ההחלפות.

תשובה מלאה לסעיף זה לא תהיה ארוכה מחמש שורות (תשובה ארוכה מדי תקרא חלקית בלבד).

שאלה 4 — שינוי השפה FLANG – פעולות אריתמטיות כפונקציות — (30 נקודות):

לצורך פתרון שאלה זו נעזר בקוד ה- interpreter של FLANG במודל הסביבות, המופיע בסוף טופס המבחן. למעשה, נשנה אותו על-מנת ש**חיבור, חיסור, כפל וחילוק יהיו פונקציות – הנמצאות בסביבה הגלובאלית** – ולא ביטויים מיוחדים, כפי שהיו בקורס. לצורך כך, נבצע שלושה שינויים:

שינוי ראשון: נשנה ביטויי פונקציה כך שלפונקציה יהיו תמיד **בדיוק** שני ארגומנטים (לצורך פשטות, לא נממש פונקציות עם מספר אחר של פרמטרים).

שינוי שני: נבטל את האפשרות לכתוב ביטוי כגון **{+ 3 1}** ומעתה נוכל לכתוב רק **{call + 3 1}** (כלומר, לקרוא לפונקציה + על ארגומנטים 3 ו-1).

שינוי שלישי: נרחיב את הסביבה הגלובאלית – שעד כה הייתה ריקה – כך שמעתה תכיר את הפונקציות הפרימיטיביות (לצורך פשטות וקיצור, תידרשו לטפל רק בחיבור וחיסור).

להלן דוגמאות לטסטים שאמורים לעבוד:

```
;; tests

(test (run "{call + 4 5}") => 9)
(test (run "{call - 9 5}") => 4)
(test (run "{+ 4 5}") =error> "parse-sexpr: bad syntax in")
(test (run "{- 9 5}") =error> "parse-sexpr: bad syntax in")
(test (run "{call {fun {x} x} 2}")
        =error> "parse-sexpr: bad syntax in") ; פונקציה חייבת שני ארגומנטים
(test (run "{call {fun {x y} {call + x { call - y 1}}}
                  4 2}")
        => 5)
(test (run "{with {first {fun {x y} x}}
                {call first 12 14}}")
        => 12)
(test (run "{with {first {fun {x y} x}}
                {with {second {fun {x y} y}}
                  {call first {call second 2 123} 124}}}")
        => 123)
```

רוב הקוד הנדרש ניתן לכם מטה ואתם נדרשים רק להשלים את הפונקציות שיאפשרו הערכת ביטויים. למען הקיצור, מובאות בפניכם רק התוספות לקוד הקיים. (שלוש נקודות "..." מופיעות במקום הקוד המושמט).
הקוד לשימושכם, אך אין צורך להתעמק בכולו. השתמשו רק בחלקים שחשובים לכם לפתרון סעיפים א', ב' ו-ג', הנתונים מטה.

```
  ;; The Flang interpreter primitive functions, using environments

  #lang pl
  #|  The grammar: שימו לב שהדקדוק הוא קצר בהרבה
    <FLANG> ::= <num>
              | { with { <id> <FLANG> } <FLANG> }
              | <id>
              | { fun { <id> <id> } <FLANG> }
                         ;;a function has two formal parameters
              | { call <FLANG> <FLANG> <FLANG> }
                         ;;a function is called with two actual parameters
|#
  (define-type FLANG
    [Num  Number]
    [Id   Symbol]
    [Add FLANG FLANG] ; Never created by user
    [Sub FLANG FLANG] ; Never created by user
    [Mul FLANG FLANG] ; Never created by user
    [Div FLANG FLANG] ; Never created by user
    [With Symbol FLANG FLANG]
    [Fun  Symbol Symbol FLANG]
    [Call FLANG FLANG FLANG])

  (: parse-sexpr : Sexpr -> FLANG)
  ;; to convert s-expressions into FLANGs
  (define (parse-sexpr sexpr)
    (match sexpr
  …
      [(cons 'fun more)
       (match sexpr
         [(list 'fun (list (symbol: name1) (symbol: name2)) body)
          (Fun name1 name2 (parse-sexpr body))]
         [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
      [(list 'call fun arg1 arg2)
       (Call (parse-sexpr fun) (parse-sexpr arg1) (parse-sexpr arg2))]
      [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

סעיף א' — eval — (12 נקודות): נתונות לכם ההגדרות הבאות עבור הפונקציה eval.

```
;; Types for environments, values, and a lookup function
  (define-type ENV …)   ;; ההגדרה ללא שינוי מהקוד המקורי
  (define-type VAL
    [NumV Number]
    [FunV Symbol Symbol FLANG ENV]) ;; פונקציה צריכה להחזיק שני פרמטרים
  (: lookup : Symbol ENV -> VAL)
  (define (lookup name env)…) ;; ההגדרה ללא שינוי מהקוד המקורי

  (: arith-op : (Number Number -> Number) VAL VAL -> VAL)
  (define (arith-op op val1 val2)…) ;; ההגדרה ללא שינוי מהקוד המקורי
```

קראו היטב את הדרישות הפורמאליות הבאות וכתבו את eval על-פיהן. הוסיפו את הקוד הנדרש (היכן שכתוב –«fill-in»–).

```
Evaluation rules:
  …
   eval({fun {x1 x2} E},env)       = <{fun {x1 x2} E}, env>
   eval({call E-op E1 E2},env1)
             = eval(Ef,extend(x2,eval(E2,env),extend(x1,eval(E1,env),envf))
                                if eval(E-op,env) = <{fun {x} Ef}, envf>
             = error!           otherwise
|#

  (: eval : FLANG ENV -> VAL)
  ;; evaluates FLANG expressions by reducing them to values
  (define (eval expr env)
    (cases expr
      [(Num n) (NumV n)]
      [(With bound-id named-expr bound-body)
       (eval bound-body
              (Extend bound-id (eval named-expr env) env))]
      [(Id name) (lookup name env)]
      [(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))]
      [(Fun bound-id1 bound-id2  bound-body)
       <--fill in 1-->]
      [(Call fun-expr arg-expr1 arg-expr2)
       (let ([fval (eval fun-expr env)])
         (cases fval
             <--fill in 2-->
           [else (error 'eval "`call' expects a function, got: ~s"
                        fval)]))]))
```

הדרכה: בהשלימכם את הקוד מעלה, ודאו שאתם מקפידים על הטיפוס הנכון של אובייקט הנשלח כארגומנט לפרוצדורה אחרת או כערך מוחזר לחישוב הנוכחי.
שימו לב שבעת הקריאה לפונקציה יש להרחיב את הסביבה בעזרת שני הארגומנטים האקטואליים. חישבו באיזו סביבה אתם מבצעים כל הערכה – הסבירו מדוע בחרתם כך (מאד בקצרה).

סעיף ב' — יצירת הסביבה הגלובאלית — (13 נקודות):

באינטרפרטר שכתבנו בכיתה, הסביבה הגלובאלית הייתה ריקה. כעת, אנחנו רוצים להתחיל את תהליך ההערכה של תכנית, כאשר הסביבה שלנו מכירה את הפונקציות הפרימיטיביות של חיבור וחיסור. לצורך כך, כתבו פרוצדורה שאינה מקבלת פרמטרים ומחזירה סביבה אשר מכירה את הפונקציה + ואת הפונקציה -.
השלימו את הקוד הבא.

```
(: createGlobalEnv : -> ENV)
(define (createGlobalEnv)
  (Extend <--fill in 3--> ))
```

הדרכה: בהשתמשו בבנאים הקיימים בכדי להגדיר את הפונקציות הפרימיטיביות. ודאו שאתם מקפידים על הטיפוס הנכון של האובייקטים שאתם מכניסים לסביבה שאתם מייצרים.

סעיף ג' — שימוש בסביבה הגלובאלית — (5 נקודות):

הסבירו בקצרה מה השינוי שנדרש בקוד של האינטרפרטר, על מנת שהסביבה הגלובאלית לא תהיה ריקה, אלא תהיה זו שיצרתם בסעיף ב'. **תארו את מיקום הקוד שיש לשנות ואת השינוי שיש לבצע.**

```
--<<<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
                        החלק של ה-parser מושמט, כיוון שהוא זהה למודל הסביבות
  ;; 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)])))
```

הפרוצדורות eval **ו-** subst **מתוך האינטרפרטר של מודל ההחלפות:**

```
(: 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)))]))
(: 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)]))]))
```