פקולטה: מדעי הטבע

מחלקה: מדעי המחשב ומתמטיקה

שם הקורס: שפות תכנות

קוד הקורס: 7036010

תאריך בחינה: _18/06/2019_ סמ' _ב'___מועד __א'__

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 106 נקודות במבחן.
- לנוחיותכם מצורפים שני קטעי קוד עבור ה- interpreter של FLANG בסוף טופס המבחן. הראשון – במודל הסביבות והשני – במודל ה-substitution cache. כמו כן, מופיעות פרוצדורות eval, subst של מודל ההחלפות.

בהצלחה!

שאלה 1 — מופעים חופשיים של שמות מזהים — (32 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.

סעיף א' – free-instance – (6 נקודות):

יהא E ביטוי (תכנית בשפה FLANG). הסבירו מהו מופע חופשי בביטוי E וכיצד עשוי מופע של x להיות לא-חופשי (קשור) בביטוי E כולו, אך חופשי בתת-ביטוי של E. תנו תשובה קצרה (לא יותר ארבע שורות). השתמשו במילים scope, binding-instance.

סעיף ב' – count-free-instances – (6 נקודות):

נתון הקוד הבא בשפה FLANG:

```
"{with {x {fun {y} {- x y}}}
    {+ {with {x {- x 3}} {+ x y}}
        y}}"
```

מהם המופעים החופשיים בקוד זה (רק כאלה שהינם חופשיים בביטוי כולו)? הסבירו לגבי כל אחד שקבעתם שהוא חופשי – מדוע הוא אכן כזה.

סעיף ג' – CFI – (20 נקודות):

בסעיף זה תכתבו פונקציה נומרית CFI (קיצור עבור CountFreeInstance) המקבלת ביטוי (תכנית) בצורת FLANG ומחזירה את המספר (הטבעי) של מופעים חופשיים של שמות מזהים, אשר מכיל הביטוי – 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.

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

אוניברסיטת
אריאל
בשומרון

הדרכה:

1. **תפקידכם למנות מופעים חופשיים ולא להעריך את הביטוי. כך, הטיפול בכל הבנאים יהיה שונה מהטיפול ב-`eval`. אל תפעילו את `eval` עצמה בקוד שלכם. מותר לכם להשתמש בפונקציות אחרות, כגון `subst`.**
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 –

```
(: count-free : String -> Natural)
(define (count-free str)
  (CFI (parse str)))
```

כל הטסטים הבאים צריכים לעבוד –

```
;tests
(test (count-free "{+ z z}") => 2)
(test (count-free "{call {fun {x} {/ x 0}} 4}") => 0)
(test (count-free "{call {fun {y} {+ x 0}} 4}") => 1)
(test (count-free "{fun {y} {- x z}}") => 2)
(test (count-free "{with {foo {fun {y} {- y r}}}
                     {call foo {* c foo}}}") => 2)
(test (count-free "{call foo {+ t r}}") => 3)
(test (count-free "{fun {x} {+ x {/ 5 0}}}") => 0)
```

שאלה 2 — באג במימוש מודל ההחלפות — (24 נקודות):

סעיף א' – `Dynamic vs. static scoping` – (6 נקודות): הסבירו בקצרה (עד ארבע שורות) מה ההבדל בין `static scoping` ל- `dynamic scoping`. התמקדו בטיפול בפונקציות.

סעיף ב' – באג במימוש מודל ההחלפות – (6 נקודות): לקראת סיום הקורס, לאחר שמימשנו אינטרפרטר במודל הסביבות, ראינו כי, למעשה, היה לנו "באג" במימוש האינטרפרטר במודל ההחלפות. בפרט, ראינו שניתן לכתוב קוד בשפה FLANG שהערכתו אמורה להחזיר הודעת שגיאה (כך אכן קורה כשמעריכים קוד זה באינטרפרטר במודל הסביבות). אולם, כשמעריכים קוד זה באינטרפרטר במודל ההחלפות – ההערכה מסתיימת ללא הודעת שגיאה. השלימו את הדוגמה הבאה לקוד בעייתי שכזה והשתמשו בה בכדי להסביר את מהות הבאג. (כתבו שלוש שורות לכל היותר, מעבר לדוגמה.)

```
"{with {foo <-- fill in --> }"
```

סעיף ג' – תיקון הבאג – (12 נקודות):

הסבירו כיצד ניתן לתקן את האינטרפרטר במודל ההחלפות.

הדרכה: ישנה שורה אחת בפונקציה eval הדורשת שינוי (פרטו מהי שורה זאת ומה השינוי הנדרש). אתם רשאים להשתמש בקוד שכתבתם בשאלה הקודמת. הסבירו (כתבו שלוש שורות לכל היותר.)

אוניברסיטת
אריאל
בשומרון

שאלה 3 — (25 נקודות):

נתון הקוד הבא:

```
(run "{with {u 2}
           {call {with {u 3}
                   {fun {x} {* x u}}}
                 7}}")
```

סעיף א' (17 נקודות):

נתון התיאור החסר הבא עבור הפעלות הפונקציה **eval** בתהליך ההערכה של הקוד מעלה במודל הסביבות (על-פי ה-interpreter העליון מבין השניים המצורפים מטה). השלימו את המקומות החסרים - באופן הבא – לכל הפעלה מספר *i* נתון לכם AST_i – הפרמטר האקטואלי הראשון בהפעלה מספר *i* *(עץ התחביר האבסטרקטי)*,*תארו את* ENV_i – הפרמטר האקטואלי השני בהפעלה מספר *i* *(סביבה) 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 = <--fill in 1-->
Env_1 = (EmptyEnv)
Res_1 = <--fill in 2-->

AST_2 = (Num 2)
Env_2 = (EmptyEnv)
Res_2 = (NumV 2)

AST_3 = <--fill in 3-->
Env_3 = (Extend 'u (NumV 2) (EmptyEnv))
Res_3 = <--fill in 4-->

AST_4 = (With 'u (Num 3) (Fun 'x (Mul (Id 'x) (Id 'u))))
Env_4 = (Extend 'u (NumV 2) (EmptyEnv))
Res_4 = <--fill in 5-->

AST_5 = (Num 3)
Env_5 = (Extend 'u (NumV 2) (EmptyEnv))
Res_5 = (NumV 3)

AST_6 = (Fun 'x (Mul (Id 'x) (Id 'u)))
Env_6 = <--fill in 6-->
Res_6 = <--fill in 5-->

AST_7 = (Num 7)
Env_7 = <--fill in 7-->
Res_7 = (NumV 7)

AST_8 = <--fill in 8-->
Env_8 = <--fill in 9-->
Res_8 = <--fill in 10-->

AST_9 = (Id 'x)
Env_9 = <--fill in 9-->
Res_9 = <--fill in 11-->

AST_{10} = (Id 'u)
Env_{10} = <--fill in 9-->
Res_{10} = <--fill in 12-->

Final result: ?

סעיף ב' (8 נקודות):

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

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

שאלה 4 — שינוי השפה FLANG – הוספת ביטויים לוגיים — (25 נקודות):

לצורך פתרון שאלה זו נעזר בקוד ה- interpreter של FLANG במודל הסביבות, המופיע בסוף טופס המבחן. נרצה להרחיב את השפה ולאפשר שימוש בביטויים ופעולות לוגיות (על ערכים בוליאניים). ביתר פירוט – נוסיף ביטויי **if** (על-פי תחביר מיוחד המודגם בטסטים למטה); אופרטורים בינאריים – **<**, **=**; וערכי **True** ו – **False** (כאן, לא נרשה #t, #f). להלן דוגמאות לטסטים שאמורים לעבוד (בפרט, כל טסט שעבד לפני הרחבת השפה, ימשיך לפעול גם לאחר ההרחבה):

```
;; tests

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

      => 5)
(test (run "{ if True then 3 else 4 }") => 3)
(test (run "{ if {= 2 4} then 3 else 4 }") => 4)
(test (run "{ if {= 2 4} then { call 3 5 } else 4 }") => 4)
(test (run "{with {is-5? {fun {x} {= x 5}}}
               {with {foo {fun {x} {if {call is-5? {+ x 1}} then 4 else 0}}}
                 {call foo 123}}}")
      => 0)
(test (run "{with {is-5? {fun {x} {= x 5}}}
                 {with {foo {fun {x} {if {call is-5? {+ x 1}} 4 0}}}
                   {call foo 123}}}")
      =error> "parse-sexpr: bad `if' syntax in (if (call is-5? (+ x 1)) 4
0)")
(test (run "{with {x {> 3 4}}
                {with {foo {fun {y} {if x then True else y}}}
                   {call foo 5}}}") => 5)
```

רוב הקוד הנדרש ניתן לכם מטה ואתם נדרשים רק להשלים את הדקדוק ואת 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.

סעיף א' — עדכון הדקדוק — (5 נקודות):

עדכנו את הדקדוק, כך שיתאים לטסטים וההגדרות הכתובות מעלה. השלימו את ארבע השורות החסרות. כל שאר הקוד עבור הניתוח התחבירי נתון לכם ואינו דורש השלמות.

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

  #lang pl
  #|  The grammar:
    The grammar:
    <FLANG> ::= <num>
              | <<-- fill in 1 -->>
              | <<-- fill in 2 -->>
              | { + <FLANG> <FLANG> }
              | { - <FLANG> <FLANG> }
              | { * <FLANG> <FLANG> }
              | { / <FLANG> <FLANG> }
              | { > <FLANG> <FLANG> }
              | <<-- fill in 3 -->>
              | <<-- fill in 4 -->>
              | <<-- fill in 5 -->>
              | { with { <id> <FLANG> } <FLANG> }
              | <id>
              | { fun { <id> } <FLANG> }
              | { call <FLANG> <FLANG> }
|#
  (define-type FLANG
  ...
  [Bool Boolean]
  [Eq FLANG FLANG]
  [Gt FLANG FLANG]
  [If FLANG FLANG FLANG])

(: parse-sexpr : Sexpr -> FLANG)
;; to convert s-expressions into FLANGs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n) (Num n)]
    ['True (Bool #t)]
    ['False (Bool #f)]
    [(cons 'if more)
     (match sexpr
       [(list 'if cond 'then t-do 'else e-do)
        (If (parse-sexpr cond) (parse-sexpr t-do) (parse-sexpr e-do))]
       [else (error 'parse-sexpr "bad `if' syntax in ~s" sexpr)])]
    [(list '> lhs rhs) (Gt (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '= lhs rhs) (Eq (parse-sexpr lhs) (parse-sexpr rhs))]
    ...))
(: parse : String -> FLANG)
;; parses a string containing a FLANG expression to a FLANG AST
(define (parse str)
```

```
  (parse-sexpr (string->sexpr str)))
```

סעיף ב' — eval — (15 נקודות): נתונות לכם ההגדרות הבאות עבור הפונקציה eval. הן עודכנו בהתאם להגדרת השפה המורחבת. **קראו היטב את הדרישות הפורמאליות הבאות** וכתבו את eval על-פיהן. הוסיפו את הקוד הנדרש (היכן שכתוב `—«fill-in»—` -- סה"כ חמישה מקומות).

```
  #|  Evaluation rules:
    eval(N,env)                  = N
    eval(True,env)               = #t
    eval(False,env)              = #f
    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({> 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(if Ec then Ey else En)= eval(En, env) if eval(Ec) = #f
                                   eval(Ey, env) otherwise
    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
  |#
;; ;; 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]
  [BoolV Boolean])

(: 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
(...)
```

הפונקציה הבאה נועדה לסייע לכם. היא מפעילה פונקציה בוליאנית דו-מקומית (על שני מספרים) ומחזירהVAL. השלימו בה את הקוד החסר.

```
(: logic-op : (Number Number -> Boolean) VAL VAL -> VAL)
;; gets a Racket logic binary operator, and uses it within a NumV
;; wrapper to return the result inside a BoolV wrapper.
(define (logic-op op val1 val2)
```

```
  (: NumV->number : VAL -> Number)
  (define (NumV->number v)
    (cases v
      [(NumV n) n]
      [else (error 'logic-op "expects a number, got: ~s" v)]))
  (BoolV <<-- fill in 1 -->>))
```

גם הפונקציה הבאה נועדה לסייע לכם. היא מקבלת FLANG ומחזירה את הערך הבוליאני המתאים לו (על פי כללים דומים לאלו ש-ראקט משתמשת בהם). השלימו בה את הקוד החסר.

```
(: extract-bool-val : VAL -> Boolean)
;; Return #f if the wrapped value is False, and #t otherwise
(define (extract-bool-val val)
  (cases val
    [(BoolV b) b]
    [else <<-- fill in 2 -->>]))

  (: eval : FLANG ENV -> VAL)
;; evaluates FLANG expressions by reducing them to values
(define (eval expr env)
  (cases expr
    ...
    [(Eq l r) (logic-op <<-- fill in 3 -->>)]
    [(Gt l r) (logic-op > <<-- fill in 4 -->>)]
    [(If cond do-yes do-no)
     <<-- fill in 5 -->>]
    ...))
```

<u>הערה:</u> שימו לב כי בטיפול בביטוי לוגי, עליכם להסיר את המעטפת של הביטוי בכדי להעריך את אמיתותו. כמו כן, בביטוי if תמיד יוערך רק אחד משני ביטויים אפשריים.

<u>סעיף ג' — שינוי הממשק מול המתכנת — (5 נקודות):</u>

לבסוף, נרצה לאפשר לאינטרפרטר המורחב להחזיר ערך מכל אחד משלושת הטיפוסים הקיימים עתה בשפה. נגדיר מחדש את הפונקציה run.

הוסיפו את הקוד הנדרש (היכן שכתוב –«fill-in»–) ל –

```
(: run : String -> (U Number Boolean VAL))
;; evaluates a FLANG program contained in a string
(define (run str)
  (let ([result (eval (parse str) (EmptyEnv))])
    (cases result
       [<--fill in 1 -->]
       [<--fill in 2 -->]
       [<--fill in 3 -->])))
--<<<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)]))]))
```