אוניברסיטת
אריאל
בשומרון

פקולטה: מדעי הטבע

מחלקה: מדעי המחשב ומתמטיקה

שם הקורס: שפות תכנות

קוד הקורס: 7036010

תאריך בחינה: 02/07/2018 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 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 — שמות מזהים בשפה — (25 נקודות):

תזכורת: במהלך הקורס החלטנו שרצוי להוסיף לשפה את האפשרות לתת שמות מזהים לערכים ולקטעי קוד.

סעיף א' – (8 נקודות): מנו לפחות ארבעה יתרונות שהזכרנו להוספת שמות מזהים – כתבו הסבר קצר וממצה ליד כל יתרון (לא יותר משני משפטים).

סעיף ב' – (9 נקודות): עבור האינטרפטר של השפה WAE (במודל ההחלפות) כתבנו את הפרוצדורה eval. **נתון הקוד החסר הבא עבור פרוצדורה זאת (קוד הפרוצדורה subst מתוך האינטרפרטר של מודל ההחלפות, מופיע בתחתית טופס המבחן):**

```
(: eval : WAE -> Number)
 ;; evaluates WAE expressions by reducing them to numbers
(define (eval expr)
  (cases expr
    [(Num n) n]
    [(Add l r) (+ (eval l) (eval r))]
    [(Sub l r) (- (eval l) (eval r))]
    [(Mul l r) (* (eval l) (eval r))]
    [(Div l r) (/ (eval l) (eval r))]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  <-- קטע קוד חסר -->))]
    [(Id name) (error 'eval "free identifier: ~s" name)]))
```

מה מבין שתי האפשרויות הבאות הינו הקוד שבו השתמשנו בקטע החסר?

אפשרות א' – `(Num (eval named-expr))`

אפשרות ב' – `named-expr`

הסבירו מה היה קורה אחרת לו היינו משתמשים באפשרות האחרת. היו תמציתיים ומדויקים בתשובתכם.

סעיף ג' – (8 נקודות): הראו קוד בשפה WAE שעבורו תהיה התנהגותם של שני האינטרפרטרים המוצגים בסעיף ב' שונה באופן מהותי (אל תסתפקו בכך שהבנאי שונה). תארו בקצרה מה יהיה השוני.

שאלה 2 — Functions as first class — (30 נקודות):

בכיתה דיברנו על כך שבשפה racket **כמו גם בשפה שלנו** FLANG **-- ההתייחסות לפונקציות היא כ-**first class.

אוניברסיטת
אריאל
בשומרון

סעיף א' – יצירת פונקציה המחשבת פולינום – (20 נקודות):

כתבו פונקציה, `createPolynomial` אשר מקבלת כקלט רשימה של k מספרים $a_0, \dots, a_{k-1}$ **ומחזירה כפלט פונקציה ממספר למספר** – אשר לוקחת קלט x ומחשבת את הפולינום $a_0 \cdot x^0 + \cdots + a_{k-1} \cdot x^{n-1}$ (מחזירה את ערך הפולינום בנקודה x). לצורך כך, תוכלו להשתמש בפונקציה `expt` של `racket` אשר לוקחת a ו-b ומחזירה a^b. עוד על פעולתה של `expt` בהמשך טופס הבחינה.

להלן דוגמאות הרצה:

```
> (createPolynomial '(1 2 4 2))
- : (Number -> Number)
#<procedure:polyX>

(define p2345 (createPolynomial '(2 3 4 5)))
(test (p2345 0) =>
   (+ (* 2 (expt 0 0)) (* 3 (expt 0 1)) (* 4 (expt 0 2)) (* 5 (expt 0 3))))
(test (p2345 4) =>
   (+ (* 2 (expt 4 0)) (* 3 (expt 4 1)) (* 4 (expt 4 2)) (* 5 (expt 4 3))))
(test (p2345 11) => (+ (* 2 (expt 11 0)) (* 3 (expt 11 1)) (* 4 (expt 11
2)) (* 5 (expt 11 3))))


(define p536 (createPolynomial '(5 3 6)))
(test (p536 11) => (+ (* 5 (expt 11 0)) (* 3 (expt 11 1)) (* 6 (expt 11
2))))

(define p_0 (createPolynomial '()))
(test (p_0 4) => 0)
```

שימו לב:

1. `createPolynomial` אמורה להחזיר פונקציה ממספר למספר.
2. כל הקריאות הרקורסיביות צריכות להיות ברקורסית זנב. ניקוד חלקי ינתן על פתרון שאינו מקפיד על רקורסית זנב.

לצורך כך תוכלו להשתמש בקוד החלקי הבא (מותר לכם גם לכתוב קוד אחר עבור אותה מטרה).

```
(: createPolynomial : (Listof Number) ->   <--fill in 1-->)
(define (createPolynomial coeffs)
  (: poly : (Listof Number) Number Integer Number -> Number)
  (define (poly argsL x power accum)
     (if <--fill in 2-->
         <--fill in 3-->
         <--fill in 4--> )
  (: polyX : Number -> Number)
  (define (polyX x)
     ←fill in 5→)
  ←fill in 6→)
```

רמז: `poly` היא פונקצית עזר המחשבת את הפולינום הלכה למעשה לאחר שניתנים לה מקדמי הפולינום, הפרמטר x ועוד כמה ארגומנטים הדרושים לרקורסיה (שתהא בקריאות זנב).
הפונקציה `polyX` מכירה את `poly`.

<u>סעיף ב' – (5 נקודות):</u> הסבירו במשפט או שניים, מה הכוונה ששפה מתייחסת לפונקציות כ- first class **וכיצד זה בא לידי ביטוי בקוד שכתבתם.**

<u>סעיף ג' – (5 נקודות):</u> הסבירו במשפט או שניים, מה היתרון בשימוש ברקורסית זנב בראקט וכיצד זה בא לידי ביטוי בקוד שכתבתם.

<u>שאלה 3 — (25 נקודות):</u>

נתון הקוד הבא:

```
(run "{call {with {x 3}
              {fun {f} {call f x}}}
            {fun {w} {* 8 w}}}")
```

<u>סעיף א' (17 נקודות):</u>

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.

```
AST1 = (Call (With 'x (Num 3) (Fun 'f (Call (Id 'f) (Id 'x))))
             (Fun 'w (Mul (Num 8) (Id 'w))))
Env1 = (EmptyEnv)
Res1 = <--fill in 1-->

AST2 = (With 'x (Num 3) (Fun 'f (Call (Id 'f) (Id 'x))))
Env2 = (EmptyEnv)
Res2 = <--fill in 2-->

AST3 = (Num 3)
Env3 = (EmptyEnv)
Res3 = (NumV 3)

AST4 = (Fun 'f (Call (Id 'f) (Id 'x)))
Env4 = (Extend 'x (NumV 3) (EmptyEnv))
Res4 = <--fill in 3-->

AST5 = (Fun 'w (Mul (Num 8) (Id 'w)))
Env5 = (EmptyEnv)
Res5 = (FunV 'w (Mul (Num 8) (Id 'w)) (EmptyEnv))
```

AST_6 = (Call (Id 'f) (Id 'x))
Env_6 = <--fill in 4-->
Res_6 = <--fill in 1-->

AST_7 = (Id 'f)
Env_7 = <--fill in 4-->
Res_7 = <--fill in 5-->

AST_8 = (Id 'x)
Env_8 = <--fill in 4-->
Res_8 = <--fill in 6-->

AST_9 = (Mul (Num 8) (Id 'w))
Env_9 = <--fill in 7-->
Res_9 = <--fill in 1-->

AST_{10} = (Num 8)
Env_{10} = <--fill in 7-->
Res_{10} = (NumV 8)

AST_{11} = (Id 'w)
Env_{11} = <--fill in 7-->
Res_{11} = <--fill in 8-->

Final result: ?

<u>סעיף ב' (8 נקודות):</u>

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 משלוש שורות (תשובה ארוכה מדי תקרא חלקית בלבד).

<u>שאלה 4 — הרחבת השפה FLANG – טיפול בפולינומים — (25 נקודות):</u>

לצורך פתרון שאלה זו נעזר בקוד ה- interpreter של FLANG במודל הסביבות, המופיע בסוף טופס המבחן (העליון מבין השניים המופיעים שם). למעשה, נרחיב אותו על-מנת לאפשר שימוש בביטויים ופעולות על פולינומים. להלן דוגמאות לטסטים שאמורים לעבוד:

```
;; tests
(test (parse "{poly 1 2 3}") => (Poly (list (Num 1) (Num 2) (Num 3))))
(test (run "{applyPoly {poly 1 2 3} 2}") => 17)
(test (run "{with { x {applyPoly {poly 1 2 3} 2}} x}") => 17)
(test (run "{with {p {poly 2 1 2 1}}
               {+ {applyPoly p 1}
                  {applyPoly p 2}}}") => 26)
```

רוב הקוד הנדרש ניתן לכם מטה ואתם נדרשים רק להשלים את הפונקציות שיאפשרו את הרחבת ה-eval. 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' ו-ב', הנתונים מטה.

```
;; The Flang interpreter with polynomials, using environments

#lang pl

#|
The grammar:
  <FLANG> ::=
            | { poly <FLANGs> }
            | { applyPoly <FLANG> <FLANG> }

   <FLANGs> ::= <FLANG> | <FLANG> <FLANGs>

Evaluation rules:
…
 eval({poly E_0 ... E_{k-1}} ,env) = polynomial \sum_{i=0}^{k-1}(a_i x^i)
                                   where a_i = (eval E_i env)
 eval({applyPoly E1 E2},env)
           = (p (eval E2 env))     if eval(E1,env1) = p
           = error!                otherwise
|#

(define-type FLANG
 …
  [Poly (Listof FLANG)]
  [AppP FLANG FLANG])

(: parse-sexpr : Sexpr -> FLANG)
;; to convert s-expressions into FLANGs
(define (parse-sexpr sexpr)
  (match sexpr
…
    [(list 'poly args ...) (Poly (map parse-sexpr args))]
    [(list 'applyPoly p arg) (AppP (parse-sexpr p) (parse-sexpr arg))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

<u>סעיף א' — פונקציות עזר ל-eval — (12 נקודות):</u> בסעיף זה, עליכם לכתוב פונקציות שיעזרו בהמשך לכתיבת ה-eval עצמה. שימו לב להרחבה הבאה לטיפוס VAL.

```
(define-type VAL
  [NumV Number]
  [PolyV (Number -> Number)]
  [FunV Symbol FLANG ENV])
```

כמו כן, הפונקציה תשמש אתכם במקומות אחרים ולכן, תהיה חיצונית (גלובאלית).

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

כתבו פונקציה `make-poly-val` אשר לוקחת כקלט רשימה של `FLANG` וסביבה ומחזירה ערך מהטיפוס המוחזר של eval העוטף פולינום.

הדרכה:

1. הפונקציה מעריכה את כל הביטויים בסביבה הנתונה. כל הביטויים הללו אמורים להיות מוערכים למקדמי הפולינום. בדקו זאת.
2. תוכלו להשתמש בפרוצדורה `map` (ראו הסבר בהמשך טופס המבחן) – לצורך כך, תוכלו להגדיר פונקציה פנימית בתוך `make-poly-val`.
3. תוכלו להשתמש בפונקציה שכתבתם בסעיף א', שאלה 2.

```
(: make-poly-val : <--fill in 1--> )
<--fill in 2--> ;;   כתבו את הפונקציה הנ"ל

(: 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))))
```

סעיף ב' — eval — (13 נקודות):

כתבו את eval. הוסיפו את הקוד הנדרש (היכן שכתוב –«fill-in»–).

```
(: eval : FLANG ENV -> VAL)
;; evaluates FLANG expressions by reducing them to values
(define (eval expr env)
  (cases expr
    …
    [<--fill in 3-->]
    [<--fill in 4-->
        [else (error 'eval "`applyPoly' expects a polynomial, got: ~s"
                     p-val)]))]))
```

הדרכה: בהשלימכם את הקוד מעלה, ודאו שאתם מקפידים על הטיפוס הנכון של אובייקט הנשלח כארגומנט לפרוצדורה אחרת או כערך מוחזר לחישוב הנוכחי.

תזכורת: (expt, map)

הפונקציה map:

קלט: פרוצדורה k-מקומית proc ורשימות $lst_1, lst_2, \ldots, lst_k$ באותו אורך.

פלט: רשימה אחת שמכילה אותו מספר איברים כמו ברשימות שהן הארגומנטים – שנוצרה ע"י הפעלת הפרוצדורה proc על האיברים עם אותו אינדקס בכל אחת מהרשימות.

(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 y) (list x y))
     '(sym1 sym2 33) '(x1 x2 44))
'((sym1 x1) (sym2 x2) (33 44))

> (map + '(1 2 3) '(4 5 6))
'(5 7 9)
```

```
Procedure
(expt z w) → number?

  z : number?

  w : number?
Returns z raised to the power of w.
Examples:
> (expt 2 3)
8
> (expt 4 0.5)
2.0
> (expt +inf.0 0)
1
--<<<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)])))
```

הפרוצדורה 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)))]))
```