אוניברסיטת
אריאל
בשומרון

פקולטה: מדעי הטבע

מחלקה: מדעי המחשב ומתמטיקה

שם הקורס: שפות תכנות

קוד הקורס: 7036010

תאריך בחינה: _30/07/2017_ סמ' _ב'___מועד __ב'__

משך הבחינה: 3 שעות

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 109 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 — with vs. call — (24 נקודות):

בכיתה דיברנו על כך שכל ביטוי FLANG יכול להיות מוחלף בביטוי FLANG שקול, אשר אינו מכיל הפעלות של בנאי With (מחליף אותן בעזרת בנאים Call ו-Fun).

סעיף א' – מקרה פרטי – (9 נקודות): נתון ה- **FLANG** הבא

```
(With x
      (Add (Num 2) (Num 3))
      (With y
            (Mul (Id x) (Id x))
            (With z
                  (Num 4)
                  (Add (Id z) (Id y)))))
```

מצאו עבורו ביטוי שקול שאינו מכיל הפעלות של בנאי With (כמוסבר למעלה).

סעיף ב' – פתרון כללי – (15 נקודות):

כתבו פונקציה **SynSugarWith**, אשר מקבלת כקלט עץ דקדוק אבסטרקטי FLANG עוברת עליו ומחזירה ביטוי FLANG שקול שאינו מכיל הפעלות של בנאי With (כמוסבר למעלה).

שאלה 2 — ממתק תחבירי (Syntactic sugar) — (16 נקודות):

תזכורת: מבנים תחביריים אשר הופכים קוד לקריא, פשוט או ברור יותר מכונים "ממתקים תחביריים" (Syntactic sugar). תיאור מקובל נוסף לממתק תחבירי הוא כל תוספת לשפה אשר אינה מרחיבה את יכולות השפה, אך מאפשרת כתיבת קוד קריא יותר.

בשפה Racket הקוד הבא מכיל שני ממתקים תחביריים שונים. (ביטוי let הוא אחד כזה).

```
#lang racket
(* (+ 2 3)
   (let ([x 2])
     (let ([y (cond
                [(< x 0) 100]
                [(> x 4) 200]
                [else 300])])
       (* x y))))
```

סעיף א' – (8 נקודות): ענו בחמש שורות לכל היותר על כל סעיף.

מהם המבנים התחביריים שמהווים ממתק תחבירי? לכל מבנה כזה הסבירו כיצד הוא הופך את הקוד לקריא יותר (באופן כללי, לאו דווקא בדוגמה מעלה) ומדוע הוא נדרש. הסבירו איזה קוד הוא מחליף.

סעיף ב' – (8 נקודות): עבור הקוד מעלה, מהו הקוד השקול בשפת Racket ללא הממתקים התחביריים? (כתבו קוד שקול שבו אין שימוש בממתק תחבירי).

אוניברסיטת
אריאל
בשומרון

שאלה 3 — (20 נקודות):

נתון הקוד הבא:

```
(run "{call {fun {f}
              {call f 4}}
            {with {z 8}
              {fun {w} {* w z}}}}")
```

סעיף א' (13 נקודות):

תארו את הפעלות הפונקציה **eval** בתהליך ההערכה של הקוד מעלה במודל ה-**substitution-cache** (על-פי ה-interpreter התחתון מבין השניים 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):

הסבירו מדוע יצאה התוצאה הזו בסעיף א'. מה היה קורה אם היינו מבצעים הערכה של הקוד מעלה במודל הסביבות – אין צורך בחישוב מלא (הסבירו). מהי התשובה הרצויה? מדוע? כתבו שלוש שורות לכל היותר. תשובה מלאה לסעיף זה לא תהיה ארוכה משלוש שורות (תשובה ארוכה מדי תקרא חלקית בלבד).

אוניברסיטת
אריאל
בשומרון

שאלה 4 — Interpreter עבור השפה ROL המורחבת — (49 נקודות):

לצורך פתרון שאלה זו נעזר בקוד ה- interpreter של FLANG במודל הסביבות, המופיע בסוף טופס המבחן (האמצעי מבין השלושה המופיעים שם). בהמשך לשאלה ראשונה, נרצה לממש Interpreter עבור השפה ROL (המורחבת) במודל הסביבות ולאפשר שימוש בביטויים ופעולות על רגיסטרים.

להלן דוגמאות לטסטים שאמורים לעבוד:

```
;; tests
  (test (run "{ reg-len =  4  {1 5 88 0}}") => '(1 5 88 0))
  (test (run "{ reg-len =  4  {exp 4 {1 0 0 0}}}") => '(1 0 0 0))
  (test (run "{ reg-len =  3  {add {1 2 3} {4 5 6}}}") => '(5 7 9))
  (test (run "{ reg-len =  4  {exp 2 {2 1 4 5}}}") => '(4 1 16 25))

  (test (run "{ reg-len =  2  {with {x {exp 3 {2 5}}}
                                 {call {fun {y} {add x y}}
                                       {2 1}}}}") => '(10 126))
 (test (run "{ reg-len =  2  {with {x {exp 3 {2 5}}}
                                 {fun {y} {add x y}}}}") => 'procedure)
(test (run "{ reg-len =  3  {add {exp 3 {1 2 3}}
                                   {exp 2 {4 5 6}}}}") => '(17 33 63))

     טסטים המחזירים הודעת שגיאה:

(test (run "{ reg-len =  2  {with {x {exp 3 {2 5}}}
                                 {call {fun {y} {add x y}}
                                       {2 1 4 5}}}}")
        =error> "parse-sexpr: wrong number of elements in (2 1 4 5)")
(test (run "{ reg-len =  3  {add {1 2 3} {fun {x} x}}}")
   =error>
   "eval: expects a list of numbers, got: (FunV x (Id x) (EmptyEnv))")
(test  (run "{ reg-len =  4  {exp {1 2 3 7} {2 1 4 5}}}")
   =error> "parse-sexpr: bad syntax in (exp (1 2 3 7) (2 1 4 5))")
```

סעיף א' — BNF — (10 נקודות):

בסעיף זה, עליכם לכתוב תחביר עבור השפה, על-פי הכללים הבאים ועל-פי הדוגמאות מעלה (**למעשה, הדוגמאות מספיקות**). תיאור מילולי: כל ביטוי בשפה, הוא מהצורה "{ reg-len = `len` A}", באשר len הוא מספר טבעי ו-A הוא ביטוי המתאר סדרת פעולות על רגיסטרים. ביטוי כזה מקיים את הכללים הבאים:

- סדרה של מספרים – עטופים בסוגריים מסולסלים (ייצוג של ערך נתון של רגיסטר).
- אם B,A סדרות פעולות על רגיסטרים, אז כן גם הביטוי המתקבל ע"י שימוש באופרטור add, באשר A, B הם האופרנדים, והביטוי כולו עטוף בסוגריים מסולסלים.
- עבור מספר n, הביטוי המתקבל ע"י שימוש באופרטור exp כאשר n ו- A הם האופרנדים, והביטוי כולו עטוף בסוגריים מסולסלים.

ביטויי with וביטויי fun הם חוקיים בדומה מאד למה שעשינו עבור השפה FLANG (כמובן שתתי הביטויים שבעזרתם יוצרים ביטוי חדש, הם עתה סדרת פעולות על רגיסטרים, במקום ביטויים בשפה FLANG).

כתבו דקדוק עבור השפה ROL בהתאם להגדרות ולדוגמאות מעלה. תוכלו להשתמש ב-<num> וב-<id> בדומה לשימוש שלהם ב-FLANG. אתם מוזמנים להשתמש בשלד הדקדוק הבא **(אין להשתמש ב- "...")**.

מספרו כל כלל שאתם מגדירים.

```
#| BNF for the ROL language:
      <ROL>  ::=
      <RegE> ::=

      <Nums> ::=
  |#
```

סעיף ב' הטיפוס RegE (7 נקודות):

כיוון שהחלק המרכזי בניתוח הסינטקטי הוא החלק של RegE, נגדיר טיפוס רק עבורו. בהמשך לסעיף א' ולטסטים מעלה, הרחיבו את הטיפוס בהתאם. הוסיפו את הקוד הנדרש (היכן שכתוב –«fill-in»–).

```
;; RegE abstract syntax trees
(define-type RegE
    [Reg  <--fill in 1-->]
    [Add  <--fill in 2-->]
    [Exp  <--fill in 3-->]
    [Id   Symbol]
    [With <--fill in 4-->]
    [Fun  Symbol RegE]
    [Call RegE RegE])
```

סעיף ג' parse (9 נקודות): כתבו את הפונקציה **parse-sexpr** בהתאם. בפונקציה זו, עליכם גם לבדוק שאורך כל רגיסטר הוא בהתאם למה שכתוב בקוד וכן שהוא לפחות 1 (אסור לכתוב תכנית עם רגיסטרים ריקים). הוסיפו את הקוד הנדרש (היכן שכתוב –«fill-in»–) ל –

```
(: parse-sexpr : Sexpr -> RegE)
;; to convert s-expressions into RegEs
(define (parse-sexpr sexpr)
  (match sexpr
    [(list 'reg-len '= (number: len) reg-sexpr)
     (if <--fill in 5--> ;; we do not allow empty registers
         (error 'parse-sexpr "Register length must be at least 1")
         <--fill in 6-->]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

```
(: parse-sexpr-RegL : Sexpr Number -> RegE)
  ;; to convert s-expressions into RegEs
  (define (parse-sexpr-RegL sexpr reg-len)
    (match sexpr
      [(list (number: ns) ... )
       (if <--fill in 7--> ;; verifying length
           (<--fill in 8-->)
           (error <--fill in 9-->] ; מהי ההודעה המתאימה?
      [(list 'add lreg rreg) <--fill in 10-->]
      [(list 'exp <--fill in 11-->]
      [(symbol: name) (Id name)]
      [(cons 'with more)
       (match sexpr
         [(list 'with (list (symbol: name) named) body)
          (With name
                (parse-sexpr-RegL named reg-len)
                (parse-sexpr-RegL body reg-len))]
         [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
      [(cons 'fun more)
       (match sexpr
         [(list 'fun (list (symbol: name)) body)
          <--fill in 12-->]
         [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
      [(list 'call fun arg) (Call <--fill in 13-->]
      [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

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

<u>סעיף ד' – הכנה לקראת eval (8 נקודות):</u>

להלן הגדרת הטיפוסים הנדרשים (בהמשך נתונות ההגדרות הפורמליות לסמנטיקה של השפה). לעיונכם.

```
(define-type ENV
    [EmptyEnv]
    [Extend Symbol VAL ENV])

  (define-type VAL
    [RegV (Listof Number)]
    [FunV Symbol RegE 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))]))
```

השתמשו בהגדרות הבאות, הנותנות ניסוח פורמלי לאופן הרצוי להערכת קוד בשפה המורחבת.

```
#| Formal specs for `eval':          Evaluation rules:
   eval(Reg,env)                  = Reg
   eval(bl)          = bl
   eval(true) = true
   eval(false) = false
   eval({add E1 E2},env) = (<x1 + y1> <x2 + y2> ... <xk + yk>)
           where eval(E1,env) = (x1 x2 ... xk)
           and eval(E2,env) = (y1 y2 ... yk)
   eval({exp n E},env) = (x1^n x2^n ... xk^n),
           where eval(E,env) = (x1 x2 ... xk).
           x^n denotes x to the power of n.
   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
 |#
```

להלן פונקצית עזר לשימושכם:

```
(: RegV->ListofNumber : VAL -> (Listof Number))
   ;; Consumes a VAL, verifies it is a RegV,
   ;; and returns the number-list obtained wrapped therein.
  (define(RegV->ListofNumber v)
    (cases v
      [(RegV ns) ns]
      [else (error 'arith-op "expects a list of numbers, got: ~s" v)]))
```

ראשית נכתוב פונקציה, אשר מקבלת כקלט מספר n **ומחזירה כפלט פונקציה ממספר למספר** – אשר לוקחת קלט x ומעלה את x בחזקת n (מחזירה x^n). לצורך כך, תוכלו להשתמש בפונקציה expt של racket אשר לוקחת a ו-b ומחזירה a^b. עוד על פעולתה בהמשך טופס הבחינה. לצורך כך תוכלו להשתמש בקוד החלקי הבא (מותר לכם גם לכתוב קוד אחר עבור אותה מטרה).

```
(: createPowerRaiser : Number ->   <--fill in 14-->)
  (define (createPowerRaiser y)
    (: powery : Number -> Number)
     <--fill in 15-->
    powery)
```

עתה נרצה לאפשר לפונקציה eval לטפל בביטויים בשפה ע"פ הגדרות אלו והטסטים מעלה. הוסיפו את הקוד הנדרש (היכן שכתוב –«fill-in»–). כנראה תרצו להשתמש בפונקציה map (תזכורת בהמשך).

```
(: reg-arith-op : (Number Number -> Number) VAL VAL -> VAL)
    ;; Consumes two registers and some binary numeric operation 'op',
    ;; and returns the register obtained by applying op on the
    ;; i'th element of both registers for all i.
(define (reg-arith-op op reg1 reg2)
  <--fill in 16-->)
```

סעיף ה' – eval – (9 נקודות):

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

```
(: eval : RegE ENV -> VAL)
  ;; evaluates RegE expressions by reducing them to bit-lists
  (define (eval expr env)
    (cases expr
      [(Reg n) <--fill in 17-->]
      [(Add l r) <--fill in 18-->]
      [(Exp n reg) (RegV <--fill in 19-->)]
      [(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)]))]))
```

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

סעיף ו' – פרוצדורת מעטפת run – (6 נקודות):

נרצה שפונקציה זאת תדע לטפל בכל ערך מוחזר מ-eval. השלימו את הקוד בהתאם לכך ולפי הטסטים מעלה.

```
(: run : String -> (U (Listof Number) Symbol))
  ;; evaluate a ROL program contained in a string
  (define (run str)
    <--fill in 20-->])
```

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)])))
```