פקולטה: מדעי הטבע

מחלקה: מדעי המחשב ומתמטיקה

שם הקורס: שפות תכנות

קוד הקורס: 7036010

תאריך בחינה: _09/07/2017_ סמ' _ב'___מועד __א'__

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.
- 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 — Eliminating free instances (30 נקודות):

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

בשאלה זו תכתבו פונקציה בוליאנית containsFreeInstance? 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 הסביבות. הניחו, אם כן, כי כל הפונקציות והטיפוסים שבו מוגדרים לכם.

סעיף א' – פרוצדורת עזר – (6 נקודות):

כתבו פרוצדורת עזר is-free? המקבלת כקלט שם-מזהה (סימבול) וסביבה ומחזירה ערך בוליאני השווה ל-true אם ורק אם המזהה אינו מוגדר בסביבה.

הדרכה – השתמשו ב- lookup כבסיס לקוד שלכם.

סעיף ב' – רמז לקראת סעיף ג' ושאלות לגביו – (8 נקודות):

בסעיף הבא תכתבו פונקציה בוליאנית containsFreeInstance?, אשר עוברת על עץ הדקדוק האבסטרקטי המייצג את הקוד (תוצר של parse) באופן דומה ל-eval (אך אינה מבצעת שום פעולת הערכה). כרמז, נספר לכם כי הטיפול בביטוי פונקציה – כלומר, בביטוי Fun – ניתן לביצוע כבר באותו שלב (זאת בניגוד ל-eval, שאינה יכולה להעריך את הפונקציה לפני ביטוי ה-Call המתאים).

ענו על השאלות הבאות (במשפט או שניים לכל אחת לכל היותר):

1. **כיצד הפונקציה שלכם תטפל בביטוי Fun?**
2. **מדוע הרמז נכון?**
3. **האם היה שוני במודל ההחלפות או במודל ה-substitution cache?**

סעיף ג' – פרוצדורה עיקרית – (16 נקודות):

בסעיף זה תכתבו פונקציה בוליאנית containsFreeInstance?.

הקוד הבא מבוסס על הפונקציה eval (באינטרפרטר של FLANG במודל הסביבות) – השלימו את הקוד במקומות החסרים.

```
(: containsFreeInstance? : FLANG ENV -> Boolean)
;; Scans FLANG expressions for free instances of identifiers
(define (containsFreeInstance? Expr env)
    (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
           (—«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 בביטויי With ו-Call יהיה מעט שונה מהטיפול ב-eval.
2. שימו לב לכל תת ביטוי שיכול להכיל מופעים חופשיים.
3. למעשה, לכל שם מזהה יש להוסיף למחסנית ערך עבורו, אך ערך זה יכול להיות תמיד אותו ערך. חשבו מדוע.
4. השתמשו גם בפונקציה שכתבתם בסעיף א.

שימו לב: אם נגדיר את פונקציית המעטפת הבאה –

```
(: check-code : String -> Boolean)
(define (check-code str)
  (containsFreeInstance? (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 "{call foo 4}") => #t)
(test (check-code "{fun {x} {+ x {/ 5 0}}}") => #f)
```

שאלה 2 — dynamic scoping (11 נקודות): **להזכירכם, ראינו בכיתה כיצד מתנהגת הגרסה של Racket כאשר היא פועלת על-פי dynamic scoping – כלומר, פונקציה רצה בסביבה בה היא נקראת ולא בסביבה בה היא הוגדרה.**

נתון הקוד הבא, מהן התוצאות שיתקבלו? – מה צריכים להיות הערכים במקום סימני השאלה הכפולים, כדי שכל הטסטים יצליחו?

הסבירו כל חלק בתשובתכם.

```
#lang pl dynamic

  (define x 22)
  (define (getx) x)
  (define (bar x) (getx))

  (test (getx) => ??)
  (test (let ([x 45]) (getx)) => ??)
  (test (getx) => ??)
  (test (bar 999) => ??)

  (define foo (let ([+ *])
                (lambda (x y) (+ x y))))
  (test (foo 6 7) => ??)
```

שאלה 3 — (20 נקודות):

נתון הקוד הבא:

```
(run "{with {foo {with {y 8}
                    {fun {x} {+ x y}}}}
           {call foo 4}}")
```

סעיף א' (13 נקודות):

תארו את הפעלות הפונקציה `eval` בתהליך ההערכה של הקוד מעלה במודל ה- environment (על-פי ה- interpreter העליון מבין השניים המצורפים מטה) - באופן הבא – לכל הפעלה מספר *i* תארו את AST_i – הפרמטר האקטואלי הראשון בהפעלה מספר *i* *(עץ התחביר האבסטרקטי), את* ENV_i – הפרמטר האקטואלי השני בהפעלה מספר *i* *(הסביבה) 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)

```
AST1  = (With x (Num 1) (Add (Id x) (Num 2)))
ENV1  = (EmptyEnv)
RES1  = (NumV 3)
AST2  = (Num 1)
ENV2  = (EmptyEnv)
RES2  = (NumV 1)
AST3  = (Add (Id x) (Num 2))
ENV3  = (Extend x (NumV 1) (EmptyEnv))
RES3  = (NumV 3)
AST4  = (Id x)
ENV4  = (Extend x (NumV 1) (EmptyEnv))
RES4  = (NumV 1)
AST5  = (Num 2)
ENV5  = (Extend x (NumV 1) (EmptyEnv))
RES5  = (NumV 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 ה- substitution-cache (על-פי ה-interpreter התחתון מבין השניים המצורפים מטה) – אין צורך בחישוב מלא (הסבירו). מהי התשובה הרצויה? מדוע? כתבו שלוש שורות לכל היותר.

שאלה 4 — הרחבת השפה FLANG מודל הסביבות – טיפול בביטויים בוליאנים — (48 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 FLANG במודל הסביבות ולהוסיף לו טיפול בערכים בוליאנים. בפרט, נוסיף את הערכים השמורים true, false ואת האופרטורים and ו- or כדלקמן:

הביטויים true, false יהיו קבועים בשפה (ערכים סימבוליים). ביטויי or ו- and הם ביטויים מיוחדים בשפה המאפשרים כל מספר סופי של ארגומנטים שכל אחד מהם הינו ביטוי חוקי בשפה FLANG.

להלן דוגמאות לטסטים שאמורים לעבוד:

```
(test (run "false") => #f)
(test (run "true") => #t)
(test (run "{or}") => #f)
(test (run "{and}") => #t)
(test (run "{and true true}") => #t)
(test (run "{and true false}") => #f)
(test (run "{and true 3 true}") => #t)
(test (run "{and true 3 4}") => 4)
(test (run "{and true false 3 4}") => #f)
(test (run "{or true false false}") => #t)
(test (run "{or false true}") => #t)
(test (run "{or true}") => #t)
(test (run "{or 1 2 true}") => 1)
(test (run "{or false false true}") => #t)
(test (run "{or false false false}") => #f)
(test (run "{or {and 2 false} {or 2 4} false}") => 2)
(test (run "{with {f {fun {x} {or x 2}}}{call f {and 1 3}}}") => 3)
(test (run "{with {me false} {and me not you}}") => #f)
(test (run "{with {me 2} {or me not you}}") => 2)
```

<u>סעיף א' (הרחבת הדקדוק) (8 נקודות):</u> עדכנו את הדקדוק הקיים. כדי להרשות מספר ארגומנטים לא ידוע, השתמשו בתו לא סופי חדש (אין להשתמש ב- "..."). אתם יכולים להשתמש בסימן λ בכדי לייצג את המילה הריקה. שימו לב שהדקדוק חד-משמעי.

```
#|
  The grammar:
    <FLANG> ::= <num>
              | { + <FLANG> <FLANG> }
              | { - <FLANG> <FLANG> }
              | { * <FLANG> <FLANG> }
              | { / <FLANG> <FLANG> }
              | { with { <id> <FLANG> } <FLANG> }
              | <id>
              | { fun { <id> } <FLANG> }
              | { call <FLANG> <FLANG> }
              | <<fill-in 1>>
              | <<fill-in 2>>
              | <<fill-in 3>>
              | <<fill-in 4>>
  <<-- fill-in 5 -->>
|#
```

<u>סעיף ב' (הרחבת הטיפוס FLANG) (3 נקודות):</u> השתמשו בטיפוס הבא ושנו את הקוד הנדרש (שימו לב שהבנאים החסרים מצפים לרשימת ארגומנטים).

```
(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]
  [And <<fill-in 6>>]
  [Or <<fill-in 7>>]
  [Tru]
  [Fal])
```

<u>סעיף ג' (parsing) (12 נקודות):</u>

השתמשו בדוגמאות הטסטים מעלה כדי להבין אילו הודעות שגיאה רצויות. הוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים – 7 השלמות סה"כ לסעיף זה) ל –

השתמשו בפונקצית העזר הבאה. (לשימושכם מצורפת תזכורת לתחביר של match, map **בהמשך טופס המבחן.)**

```
(: parse-sexpr* : (Listof Sexpr) -> (Listof FLANG))
 (define (parse-sexpr* lsp)
   (map parse-sexpr lsp))

(: parse-sexpr : Sexpr -> FLANG)
;; to convert s-expressions into FLANGs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)     (Num n)]
    ['true <<fill-in 8>>]
    [<<fill-in 9>>]
    [(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))]
    [(list 'or exprs ...) (Or <<fill-in 10>>)]
    [ <<fill-in 11>> ]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

סעיף ד' – פונקציות עזר להערכה – (18 נקודות):

בסעיף זה אתם נדרשים לכתוב פונקציות עזר שבהן תשתמש eval לטיפול ברשימת ארגומנטים לביטויי or או and. השתמשו בהגדרות הסמנטיקה הבאות (ובדוגמאות שניתנו מעלה):

הסמנטיקה עבור הביטויים הנ"ל תהיה בדומה לסמנטיקה של הביטויים המקבילים בשפת Racket.

בביטוי and, כל ארגומנט יחושב בתורו משמאל לימין החל מהארגומנט מהראשון. במקרה בו שיחושב ארגומנט שערכו הוא הערך השמור false, הערך המוחזר יהיה false וכמובן שלא יחושבו הארגומנטים שאחריו. אם כל הארגומנטים אינם false יוחזר הערך הארגומנט האחרון שחושב. במקרה בו לא ניתנו ארגומנטים כלל יחזיר true.

בביטוי or יקבל גם הוא כל מספר סופי של ארגומנטים ויחשב אותם בזה אחר זה החל מהראשון, אם יתקל בארגומנט שערכו אינו false יחזיר את ערכו וכמובן שלא יחשב את הארגומנטים הבאים. במידה וכל הארגומנטים הם false או שלא ניתנו ארגומנטים כלל יחזיר false.

לצורך ההערכה, עליכם לעדכן גם את הטיפוס VAL.

```
;; Types for environments, values, and a lookup function
(define-type ENV
  [EmptyEnv]
  [Extend Symbol VAL ENV])
(define-type VAL
  [NumV Number]
  [<<fill-in 12>>]
  [<<fill-in 13>>]
  [FunV Symbol FLANG ENV])

(: eval-and : (Listof FLANG) ENV -> VAL)
(define (eval-and args env)
   <<fill-in 14>>)

(: eval-or : (Listof FLANG) ENV -> VAL)
(define (eval-or args env)
   <<fill-in 15>>)
```

סעיף ה' (evaluation) (7 נקודות):

עתה, נעדכן את הפרוצדורה eval על פי ההגדרות הפורמליות שהוגדרו מעלה ועל פי הטסטים מעלה. השלימו את הקוד החסר עבור הפרוצדורה eval.

```
 (: eval : FLANG ENV -> VAL)
;; evaluates FLANG expressions by reducing them to values
;; printing the value of arguments in call number counterx
(define (eval expr env)
  (cases expr
    ;;rest as before
    [(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)]))]
    [(And args) <<fill-in 16>>]
    [<<fill-in 17>>]
    [<<fill-in 18>>]
    [<<fill-in 19>>]))
```

<u>פונקצית ממשק run – ללא ניקוד:</u>

לבסוף, נעדכן את הפונקציה run כך שתדע להחזיר גם ערכים בוליאנים, במקרים שבהם הערך המוחזר הינו true או false. ודאו שקוד זה מתאים להשלמתכם את הקוד החסר.

```
(: run : String -> (U Number Boolean))
;; evaluate a FLANG program contained in a string
(define (run str)
  (let ([result (eval (parse str) (EmptyEnv))])
    (cases result
      [(NumV n)  n]
      [(TrueV) #t]
      [(FalseV) #f]
      [else
       (error 'run "eval returned a non-number non-Boolean: ~s" result)])))
```

<u>תזכורת: (match, map)</u>

<u>הפונקציה map:</u>

קלט: פרוצדורה proc ורשימה lst

פלט: רשימה שמכילה אותו מספר איברים כמו ב- lst – שנוצרה ע"י הפעלת הפרוצדורה proc על כל אחד מאיברי הרשימה lst. (ההסבר הבא הוא כללי יותר – כי למעשה הפונקציה map יכולה לטפל במספר רשימות – לצורך השאלה הנתונה לא תזדקקו לשימוש כזה)

(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) (list x))
    '(sym1 sym2 33))
'((sym1) (sym2) (33))
```

- **The `match' Form**

```
The syntax for `match' is

  (match value
    [pattern result-expr]
    ...)
A few more useful patterns:

  id              -- matches anything, binds `id' to it
  _               -- matches anything, but does not bind
  (number: n)     -- matches any number and binds it to `n'
  (symbol: s)     -- same for symbols
  (string: s)     -- strings
  (sexpr: s)      -- S-expressions (needed sometimes for Typed Racket)
  (and pat1 pat2) -- matches both patterns
  (or pat1 pat2)  -- matches either pattern (careful with bindings)

The patterns are tried one by one *in-order*, and if no pattern matches the
value, an error is raised.

Note that `...' in a `list' pattern can follow *any* pattern, including all
of the above, and including nested list patterns.

--<<<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)])))
```