פקולטה: מדעי הטבע

מחלקה: מדעי המחשב ומתמטיקה

שם הקורס: שפות תכנות

קוד הקורס: 2-7036010-1/3

תאריך בחינה: 05/11/2014 סמ' ק' מועד ב'

משך הבחינה: 3 שעות

שם המרצה: ערן עמרי

חומר עזר: אסור

שימוש במחשבון: לא

הוראות כלליות:

- כתבו את תשובותיכם בכתב קריא ומרווח. על כל התשובות להופיע במחברת התשובות המצורפת. הבדיקה לא תחשיב תשובות על שאלון הבחינה 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.
- אפשר להסתמך על סעיפים קודמים גם אם לא עניתם עליהם.
- יש לענות על כל השאלות.
- ניתן להשיג עד 108 נקודות במבחן.
- לנוחיותכם מצורפים שלושה קטעי קוד עבור ה- interpreters של FLANG בסוף טופס המבחן. הראשון – במודל ה-substitution , השני במודל הסביבות והשלישי במודל ה-substitution cache.

שאלה 1 — BNF — (22 נקודות): נתון הדקדוק (BNF) הבא:

```
<ME> ::= <N>
         | <N>  +  <ME>
         | <N>  -  <ME>
         | <ME> *  <N>
         | <ME> /  <N>
<N> ::= 0 | 1 | x | y | z
```

סעיף א' (5 נקודות): **(סמנו את כל התשובות הנכונות):**

א. השפה שמגדיר הדקדוק מכילה בפרט את כל הביטויים האריתמטיים על מספרים בינאריים.

ב. השפה שמגדיר הדקדוק הינה שפת הביטויים האריתמטיים עם ארבעה סימני פעולות חשבון.

ג. בשפה שמגדיר הדקדוק יש מילים באורך גדול מ-n, לכל מספר טבעי n.

ד. לא ניתן להגדיר סמנטיקה לשפה שמגדיר הדקדוק מבלי להגדיר interrupt עבור ניסיון חלוקה ב-0.

סעיף ב' (12 נקודות):

עבור כל אחת מן המילים הבאות, קבעו האם ניתן לייצר את המילה מהדקדוק הנתון. אם תשובתכם חיובית, הראו עץ גזירה עבורה. אם תשובתכם שלילית, הסבירו מדוע לא ניתן לגזור את המילה.

1. $x + y * z - z$
2. $x * 0 + z$
3. $0 + 1 + 0 + 1 / 1 / 0$
4. $\{\{0 + 0\} / \{x - y\}\}$
5. $+ + + + + a\ a\ a\ a\ a\ a$
6. $0 - 1 - z + 1$
7. $0 / 1 * x / 0$

סעיף ג' (5 נקודות): בחרו תשובה **אחת** נכונה:

1. הדקדוק הנתון אינו חד-משמעי. זאת מכיוון שיש בשפה שהוא מגדיר מילים שמשמעותן תתברר רק כאשר יוצבו מספרים במשתנים.
2. הדקדוק הנתון חד-משמעי. זאת מכיוון שמשמעותה של מילה בשפה צריכה להתברר רק בשלב ה-eval ובשלב הסופי של parse.
3. הדקדוק הנתון חד-משמעי. זאת מכיוון שלכל מילה שנגזרה ממנו, ברור מה הכלל הראשון שהופעל בגזירה.
4. הדקדוק הנתון אינו חד-משמעי. זאת מכיוון שקיימת מילה שיש עבורה שני עצי גזירה שונים.

אוניברסיטת
אריאל
בשומרון

שאלה 2 — שאלות כלליות — (20 נקודות):

לפניכם מספר שאלות פשוטות. עליכם לבחור את התשובות הנכונות לכל סעיף (ייתכנו מספר תשובות נכונות – סמנו את כולן).

סעיף א' (7 נקודות): (סמנו את כל התשובות הנכונות)

אילו מהמשפטים הבאים נכונים לגבי תהליך ה-Parsing?

א. תהליך ה-Parsing עשוי להיות מנותק מתהליך ההערכה של התכנית. בפרט, על התוצר של התהליך הראשון יכולות להיות מופעלות פונקציות הערכה שונות לגמרי המחזירות ערכים מטיפוסים שונים.

ב. Compositionality הינה תכונה חשובה לתהליך ה-parsing ואם היא אינה קיימת אז תהליך ה-parsing עשוי להפוך לא יעיל.

ג. בשפות המאפשרות הכרזה על שמות מזהים, תהליך ה-Parsing אינו יכול להתבצע בסריקה יחידה של הקוד. ראשית, עלינו לבצע סריקה לסילוק מופעים חופשיים. במימוש שלנו, מתבצעת סריקה זו בשלב ה-eval, אך זוהי פעולה תחבירית במהותה.

ד. במימוש שלנו, במודל הסביבות – בתום תהליך ה-Parsing מתקבל FLANG. הוא מייצג עץ תחביר אבסטרקטי בו כל קודקוד מיוצג על-ידי בנאי של FLANG או ארגומנט של אותו בנאי. רק לבנאי עשויים להיות בנים בעץ.

סעיף ב' (8 נקודות): (סמנו את כל התשובות הנכונות)

אילו מהמשפטים הבאים נכונים לגבי dynamic vs. static (lexical) scoping?

א. ההבדל המרכזי בין שני המודלים הוא בהתייחסות לפונקציות. ב-static (lexical) scoping , פונקציה הינה אובייקט שלם (סגור) וגם אם תופעל מאוחר יותר בתכנית, תשתמש תמיד באותם ערכים לשמות מזהים בגוף הפונקציה. ב-dynamic scoping , ערכים אלו עשויים להשתנות בין הפעלות.

ב. במימוש האינטרפרטר שלנו במודל ההחלפה קיבלנו lexical scoping ובמימוש במודל הסביבות קיבלנו dynamic scoping. אולם, במודל ההחלפה היה באג, שהתנהג כמו dynamic scoping במקרה מסויים.

ג. במודל dynamic scoping, הקוד הבא מחזיר 1 ואחר כך 2:

```
(define (foo)  x)
(let ([x 1])  (foo))
(define (bar x)  (foo))
(let ([x 1]) (bar 2))
```

ד. במודל static scoping, הקוד מסעיף ג', מחזיר 1 ואחר כך שוב 1.

סעיף ג' (5 נקודות): (סמנו את כל התשובות הנכונות)

א. חשיבותה המרכזית של הגדרת טיפוסים באופן סטטי הינה ביכולת האבסטרקציה של המתכנת, שכן ללא טיפוסים מובנים ומוגדרים לא הייתה לנו האפשרות לייצר מבניות ברורה לכל אובייקט בתוכנית.

ב. השפה שכתבנו בקורס מתייחסת לפונקציות כ- first class ובפרט, מאפשרת לפונקציה לקבל כפרמטר פונקציה שיוצרה בזמן ריצה. אולם, היא אינה מאפשרת רקורסיה.

ג. האופטימיזציה של Racket לקריאות זנב נועדה לשפר את ניהול הזכרון שבשימוש התכנית.

אוניברסיטת
אריאל
בשומרון

שאלה 3 — שפת הרגיסטרים — (48 נקודות):

בשאלה זו נבנה אינטרפרטר לשפה פשוטה מאד, שתקרא RegE – שפת הפעולות האריתמטיות על רגיסטרים (למעשה, וקטורים מעל הקבוצה {0,1}). השפה תאפשר:

1. **פעולת and על שני רגיסטרים (מתבצעת לכל ביט בנפרד).**
2. **פעולת or על שני רגיסטרים (מתבצעת לכל ביט בנפרד).**
3. **פעולת shl (קיצור עבור SHIFT-LEFT) על רגיסטר.**
4. **קישור שם משתנה לערך של רגיסטר (ביטויי with).**

להלן מספר טסטים אשר אמורים לעבוד:

```
;; tests
(test (run "{1 0 0 0}") => '(1 0 0 0))
(test  (run "{shl {1 0 0 0}}") => '(0 0 0 1))
(test (run "{and {shl {1 0 1 0}}{shl {1 0 1 0}}}") => '(0 1 0
1))
(test (run "{ or {and {shl {1 0 1 0}} {shl {1 0 0 1}}} {1 0 1
0}}") => '(1 0 1 1))
(test (run "{ or {and {shl {1 0}} {1 0}} {1 0}}") => '(1 0))
(test (run "{with {x {1 1 1 1}} {shl y}}") =error> "free
identifier")
(test (run "{ with {x { or {and {shl {1 0}} {1 0}} {1 0}}}
                {shl x}}") => '(0 1))
```

כפי שניתן לראות, נתייחס לרגיסטר כרשימה של אפסים ואחדות. אנחנו נניח שכל הרגיסטרים הם באותו אורך (ואין צורך לוודא נכונות הנחה זו), אך לא נניח שאנחנו יודעים אורך זה בעת כתיבת האינטרפרטר.

מבנה האינטרפרטר עבור RegE יהיה מבוסס על מבנה האינטרפרטר שכתבנו עבור השפה WAE.

ענו על השאלות הבאות והשלימו את הקוד החסר במקומות המסומנים (כתבו במחברת התשובות את שורות הקוד החסרות – אין צורך להעתיק קוד המופיע בטופס המבחן).

ראשית, נגדיר שני טיפוסים חדשים:

```
;; Defining two new types
(define-type BIT = (U 0 1))
(define-type Bit-List = (Listof BIT))
```

סעיף א' (8 נקודות):

נגדיר פונקציה עבור טיפול בפעולות הנדרשות למימוש הפעולות המוזכרות לעיל, בשפה **pl** (הגרסה של **Racket** בה אנו משתמשים). להלן דוגמאות לטסטים שאמורים לעבוד:

```
(test (bit-or 0 1) => 1)
(test (bit-and 1 0) => 0)
(test (bit-and 1 1) => 1)
(test (shift-left '(1 0 1 1)) => '(0 1 1 1))
```

השלימו את הקוד לכתיבת הפונקציה המקבלת שני ביטים ומחזירה 1 אם שניהם שווים 1 ו-0 אחרת.

```
;; Defining functions for dealing with arithmetic operations
;; on the above types
 (: bit-and : BIT BIT -> BIT)        ;; Arithmetic and
 (define(bit-and a b)
  —«fill-in #1»—)              ;; ; Add -- you can use logical and
```

השלימו את הקוד לכתיבת הפונקציה המקבלת שני ביטים ומחזירה 1 אם לפחות אחד מהם שווה 1 ו-0 אחרת.

```
 (: bit-or : BIT BIT -> BIT)         ;; Aithmetic or
 (define(bit-or a b)
   —«fill-in #2»—)              ;; ; Add --  ;; you can use logical or
```

השלימו את הקוד לכתיבת הפונקציה המקבלת רשימה של ביטים (רגיסטר) ומחזירה רשימה (באותו אורך) שנתקבלה מהרשימה המקורית על ידי הזזת כל איבר - תא אחד שמאלה והזזת השמאלי ביותר לימין הרשימה. ניתן להניח שהרשימה אינה ריקה.

```
 (: shift-left : Bit-List -> Bit-List)
  ;; Shifts left a list of bits (once)
 (define(shift-left bl)
  —«fill-in #3»—)              ;; ; Add
```

סעיף ב' (4 נקודות):

בסעיף זה נגדיר דקדוק עבור השפה **RegE** (הוסיפו את הקוד הנדרש היכן שכתוב —«fill-in»—):

הערה: לצורך פשטות הדקדוק, נניח כאן שאורך כל רגיסטר הוא 4, אך את כל האינטרפרטר נכתוב לאורך כללי (לאו דווקא 4).

```
#| BNF for the RegE language:
    <RegE> ::= { 0 0 0 0 } | { 0 0 0 1 } | { 0 0 1 0 } |··· | { 1 1 1 1 }
        | { and <RegE> <RegE> }
        | { or —«fill-in #4»—}   ;; ; Add
        | { —«fill-in #5»— } ;; העזרו בטסטים שמעלה והשתמשו בשם הפעולה המתאים
        | { with { <id> <RegE> } <RegE> }
        | <id>                                  |#
```

סעיף ג' (5 נקודות):

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

```
;; RegE abstract syntax trees
 (define-type RegE
  [Reg  Bit-List]
  [And  –«fill-in #6»–]  ;; ; Add
  [Or   –«fill-in #7»–]  ;; ; Add
  [Shl  –«fill-in #8»–]  ;; ; Add
  [Id   Symbol]
  [–«fill-in #9»–])      ;; ; Add
```

סעיף ד' (7 נקודות):

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 (and a (or 1 0)) ... ) (Reg (list->bit-list a))]  ;; see def. of list->bit-list below.
   [(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)])]
   [(list 'and lreg rreg) [–«fill-in #10»–] ;; ; Add
   [–«fill-in #11»–]  ;; ; Add
   [–«fill-in #12»–]  ;; ; Add
   [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

הפונקציה הבאה למעשה מבצעת casting ואין חובה להבינה לטובת מענה על הסעיף –

```
;; Next is a technical function that converts (casts)
;; (any) list into a bit-list. We use it in parse-sexpr.
 (: list->bit-list : (Listof Any) -> Bit-List)
 ;; to cast a list of bits as a bit-list
 (define (list->bit-list lst)
  (cond [(null? lst) null]
        [(eq? (first lst) 1)(cons 1 (list->bit-list (rest lst)))]
        [else (cons 0 (list->bit-list (rest lst)))]))

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

סעיף ה' (8 נקודות):

השתמשו בהגדרות הפורמליות עבור החלפות והוסיפו את הקוד הנדרש בהגדרת הפונקציה subst (היכן שמופיע –«fill-in»–).

```
#| Formal specs for `subst':
   (`BL' is a Bit-List, `E1', `E2' are <RegE>s,
                `x' is some <id>, `y' is a  *different* <id>)
      BL[v/x]                 = BL
      {and E1 E2}[v/x]        = {and E1[v/x] E2[v/x]}
      {or E1 E2}[v/x]         = {or E1[v/x] E2[v/x]}
      {shl E}[v/x]            = {shl E[v/x]}
      y[v/x]                  = y
      x[v/x]                  = v
      {with {y E1} E2}[v/x]   = {with {y E1[v/x]} E2[v/x]}
      {with {x E1} E2}[v/x]   = {with {x E1[v/x]} E2}
|#

(: subst : RegE Symbol RegE -> RegE)
;; 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
   [(Reg bl) –«fill-in #13»–] ;; Add שימו לב מה קורה במימוש שלנו
   [(And l r) –«fill-in #14»–] ;; Add
   [–«fill-in #15»–]  ;; Add
   [–«fill-in #16»–]  ;; Add
   [(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)))]))
```

סעיף ו' (9 נקודות):

עתה נרצה לאפשר לפונקציה eval להפעיל פונקציות אריתמטיות על רגיסטרים (לטובת מימוש or ו-and).
לצורך כך נגדיר פונקציה שתקבל פעולה op (הפועלת על זוג ביטים) ושני רגיסטרים ותפעיל את op על כל שני ביטים (שמיקומם ברגיסטר זהה) . בסעיף זה עליכם להשלים את הקוד עבור פונקציה זו.

```
(: bit-arith-op : (BIT BIT -> BIT) Bit-List Bit-List -> Bit-List)
 ;; Consumes two registers and some binary bit-operation 'op',
 ;; and returns the register obtained by applying op on the
 ;; i'th bit of both registers for all i.
(define(bit-arith-op op reg1 reg2)
 [–«fill-in #17»–) ;; Add
```

סעיף ז' (7 נקודות):

השתמשו בהגדרות הפורמליות עבור eval (ובטסטים לעיל) והוסיפו את הקוד הנדרש בהגדרת הפונקציה eval (היכן שמופיע –«fill-in»–).

```
#| Formal specs for `eval':
   eval(pl)          = pl
   eval({and E1 E2}) =
             (<x1 bit-and y1> <x2 bit-and y2> ... <xk bit-and yk>)
             where eval(E1) = (x1 x2 ... xk) and eval(E2) = (y1 y2 ... yk)
   eval({or E1 E2}) = (<x1 bit-or y1> <x2 bit-or y2> ... <xk bit-or yk>)
             where eval(E1) = (x1 x2 ... xk) and eval(E2) = (y1 y2 ... yk)
   eval({shl E}) = (x2 ... xk x1), where eval(E) = (x1 x2 ... xk)
   eval(id)        = error!
   eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
 |#
```

```
(: eval : RegE -> Bit-List)
;; evaluates RegE expressions by reducing them to bit-lists
(define (eval expr)
  (cases expr
    [(Reg n) n]
    [(And l r) –«fill-in #18»–]
    [(Or l r) –«fill-in #19»–]
    [–«fill-in #20»–]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
            bound-id
            [–«fill-in #21»–]))]
    [(Id name) (error 'eval "free identifier: ~s" name)]))
```

```
(: run : String -> Bit-List)
 ;; evaluate a RegE program contained in a string
 (define (run str)
   (eval (parse str)))
```

אוניברסיטת
אריאל
בשומרון

שאלה 4 — הרצת קוד בשפה FLANG — (18 נקודות):

נתון הקוד הבא:

```
(run "{with {x {fun {y} y}}
           {with {f {fun {z} {call x z}}}
              {with {x 3}
                 {call f x}}}}")
```

סעיף א' (13 נקודות):

בתהליך ההערכה של הקוד מסעיף א' במודל ה- environment (על-פי ה-interpreter התחתון מבין השניים המצורפים מטה) תופעל הפונקציה eval 13 פעמים. לכל הפעלה מספר *i* מתוך 13 ההפעלות הללו (על-פי סדר הופעתן בחישוב), עליכם לתאר 3 ערכים (**שימו לב: עליכם להסביר בקצרה כל מעבר**):

AST_i – הפרמטר האקטואלי הראשון בהפעלה מספר *i* *(עץ התחביר האבסטרקטי)*.

ENV_i – הפרמטר האקטואלי השני בהפעלה מספר *i* *(הסביבה)*.

RES_i – הערך המוחזר מהפעלה מספר *i*.

דוגמה: עבור הקוד:

```
(run "{with {x 1} {+ x 2}}")
```

היה עליכם לענות

```
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)
```

סעיף ב' (5 נקודות):

מה הייתה תוצאת החישוב של הקוד מסעיף א' אם היינו מריצים אותו באינטרפרטר המשתמש ב-substitution caches?
לשימושכם, נתון קוד האינטרפרטר הנ"ל בסוף טופס המבחן (מופיע שלישי).
הסבירו את תשובתכם.

```
---<<<FLANG>>>-------------------------------------------------------

  ;; The Flang interpreter (substitution model)

  #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:
    subst:
      N[v/x]                 = N
      {+ E1 E2}[v/x]         = {+ E1[v/x] E2[v/x]}
      {- E1 E2}[v/x]         = {- E1[v/x] E2[v/x]}
      {* E1 E2}[v/x]         = {* E1[v/x] E2[v/x]}
      {/ E1 E2}[v/x]         = {/ E1[v/x] E2[v/x]}
      y[v/x]                 = y
      x[v/x]                 = v
      {with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]} ; if y =/= x
      {with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}
      {call E1 E2}[v/x]      = {call E1[v/x] E2[v/x]}
      {fun {y} E}[v/x]       = {fun {y} E[v/x]}            ; if y =/= x
      {fun {x} E}[v/x]       = {fun {x} E}
    eval:
      eval(N)             = N
      eval({+ E1 E2})     = eval(E1) + eval(E2)   \ if both E1 and E2
      eval({- E1 E2})     = eval(E1) - eval(E2)    \ evaluate to numbers
      eval({* E1 E2})     = eval(E1) * eval(E2)    / otherwise error!
      eval({/ E1 E2})     = eval(E1) / eval(E2)   /
      eval(id)            = error!
      eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
      eval(FUN)           = FUN ; assuming FUN is a function expression
      eval({call E1 E2}) = eval(Ef[eval(E2)/x]) if eval(E1)={fun {x} Ef}
                          = 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)))

(: 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)))]))
```

```
(: 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 -> 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)]))]))

 (: run : String -> Number)
 ;; evaluate a FLANG program contained in a string
 (define (run str)
   (let ([result (eval (parse str))])
     (cases result
       [(Num n) n]
       [else (error 'run
                    "evaluation returned a non-number: ~s" result)])))

 ;; tests
 (test (run "{call {fun {x} {+ x 1}} 4}")
       => 5)
 (test (run "{with {add3 {fun {x} {+ x 3}}}
                {call add3 1}}")
       => 4)
```

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

;; tests
(test (run "{call {fun {x} {+ x 1}} 4}")
      => 5)
(test (run "{with {add3 {fun {x} {+ x 3}}}
               {call add3 1}}")
      => 4)
(test (run "{with {add3 {fun {x} {+ x 3}}}
               {with {add1 {fun {x} {+ x 1}}}
                 {with {x 3}
                   {call add1 {call add3 x}}}}}")
      => 7)
(test (run "{with {identity {fun {x} x}}
               {with {foo {fun {x} {+ x 1}}}
                 {call {call identity foo} 123}}}")
      => 124)
(test (run "{with {x 3}
               {with {f {fun {y} {+ x y}}}
                 {with {x 5}
                   {call f 4}}}}")
      => 7) ;; the example we considered for subst-caches
(test (run "{call {with {x 3}
                    {fun {y} {+ x y}}}
                  4}")
      => 7)
(test (run "{call {call {fun {x} {call x 1}}
                        {fun {x} {fun {y} {+ x y}}}}
                  123}")
      => 124)
```

```
--<<<FLANG-Subst-cache>>>-----------------------------------------------

  ;; The Flang interpreter, using substitution caches

  #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)])))
```