אוניברסיטת
אריאל
בשומרון

פקולטה: מדעי הטבע

מחלקה: מדעי המחשב ומתמטיקה

שם הקורס: שפות תכנות

קוד הקורס: 2-7036010-1/3

תאריך בחינה: 06/07/2014 סמ' ב' מועד ב'

משך הבחינה: 3 שעות

שם המרצה: ערן עמרי

חומר עזר: אסור

שימוש במחשבון: לא

הוראות כלליות:

- כתבו את תשובותיכם בכתב קריא ומרווח.
- בכל שאלה או סעיף, 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 107 נקודות במבחן.
- לנוחיותכם מצורפים שני קטעי קוד עבור ה- interpreter של FLANG בסוף טופס המבחן. הראשון – במודל ה-substitution והשני במודל הסביבות.

אוניברסיטת
אריאל
בשומרון

שאלה 1 — BNF — (22 נקודות):

סעיף א' (13 נקודות):

השתמשו בשלד הדקדוק הנתון מטה, בכדי לכתוב BNF עבור השפה "**LE**" שהיא שפה של "List Expressions" כמוגדר להלן. מילים חוקיות בשפה "**LE**" הן ביטויים המוערכים ל-S-expressions של Racket אשר בהם תתי הביטוי האטומיים הם מספרים וסימבולים של Racket. סימני הפעולה המותרים בביטויים אלו הם: **append**, **list**, **cons**. גם **null** הוא ביטוי חוקי בשפה. לפניכם מספר כללים נוספים לגבי ביטויים בשפה:

1. כל ביטוי בשפה LE שאינו מספר או סימבול – אם יוערך בעזרת האינטרפרטר של Racket – יוערך לרשימה (שימו לב: לא כל ביטוי חוקי המייצג רשימה ב-Racket הינו חוקי ב-LE).
2. כל ביטוי בשפה LE שאינו מספר או סימבול ואינו null – מתחיל בסוגר שמאלי ונגמר בסוגר ימני.
3. ביטוי שהאופרטור שלו הוא **cons** יכיל שני אופרנדים, השני בהם אינו מספר או סימבול.
4. ביטוי שהאופרטור שלו הוא **append** יכיל מספר כלשהו (כולל 0) של אופרנדים, אשר אף אחד מהם אינו מספר או סימבול.
5. ביטוי שהאופרטור שלו הוא **list** יכיל מספר כלשהו (כולל 0) של אופרנדים כלשהם.
6. **<num>** מתאר ערך מספרי כלשהו על-פי הגדרת RACKET.
7. **<sym>** מתאר סימבול כלשהו על-פי הגדרת RACKET. שימו לב שסימבול אינו כולל גרש.
8. הסימן **...** מייצג אפשרות חזרה של מספר כלשהו (כולל 0) של פעמים של האלמנט הצמוד משמאל.

דוגמאות לביטויים בשפה:

```
Null
12
'boo
(cons 1 (cons 'two null))
(list 1 2 3)
(list (list (list (list 1 2 3))))
(append (list 1 2) (list 3 4) (list 5 6))
(list)
(append)
(cons 1 (cons (append (cons 'x null) (list 'y 'z)) null))
```

דוגמאות לביטויים שאינם בשפה:

```
(cons 1 2)
(list (3 4))
(quote boo)
(append 1 (list 2 3) 4)
(cons 1 2 null)
(cons 1 (2))
(cons 1 '())
'(1 2 3)
(cons '1 null)
(list ''a)
(car (list 1 2))
```

השתמשו בשלד הבא לכתיבת הדקדוק. עליכם להשלים את שמונת הכללים החסרים.

```
<LE>   ::= 1 כלל
         | 2 כלל
<LIST> ::= 3 כלל
         | 4 כלל
         | 5 כלל
         | 6 כלל
<ATOM> ::= 7 כלל
         | 8 כלל
```

סעיף ב' (9 נקודות):

הראו כיצד נגזרות שלוש המילים הבאות מן הדקדוק שכתבתם בסעיף הקודם. הקפידו לציין באיזה כלל השתמשתם בכל שלב. שימו לב לא לדלג על שלבים בגזירה.

```
1. 23
2. (list 1 (cons 'two null) (append (append) (list)))
3. (cons 1 (cons (append (cons 'x null) (list 'y 'z)) null))
```

אוניברסיטת
אריאל
בשומרון

שאלה 2 — שאלות כלליות — (35 נקודות):

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

סעיף א' (5 נקודות):

מה יקרה עם הפעלת הקוד הבא ב-Racket?

```
(: s : Any -> Any)
(define (s x)
  (s x))
(s s)
```

א. האינטרפרטר יחזיר שגיאה כיוון שבשפות שאינן מתייחסות לפונקציה כ- first class, לא ניתן להפעיל פונקציה על עצמה.
ב. הפונקציה תכנס ללולאה אינסופית והריצה לא תעצור.
ג. הפונקציה תכנס ללולאה אינסופית והריצה תעצור כאשר יגמר זכרון המחסנית.
ד. אי אפשר לדעת כיוון שזה תלוי במימוש של Racket והאם היא משתמשת באופטימיזציה של קריאות זנב.

סעיף ב' (5 נקודות):

אילו מהמשפטים הבאים נכונים לגבי תהליך ה-Parsing?

א. בתהליך זה אנו ממירים טקסט לעץ תחביר.
ב. בסוף התהליך אנו יודעים מה הערך המוחזר מהרצת התכנית.
ג. תהליך ה-Parsing תלוי באופן מובהק בהגדרת הדקדוק חסר ההקשר המגדיר את שפת התכניות החוקיות.
ד. תהליך ה-Parsing אינו בהכרח תלוי בסדר הופעת האופרטור והאופרנדים בביטוי חוקי, על-פי הדקדוק חסר ההקשר המגדיר את שפת התכניות החוקיות – כל עוד קיים תאום עם תהליך הערכת התכנית.

סעיף ג' (5 נקודות):

אילו מהמשפטים הבאים נכונים לגבי שפות המתייחסות לפונקציות כ- first class?

א. בשפה כזו, כל פונקציה יכולה להחזיר מספר כלשהו (סופי) של ערכים.
ב. בשפות הללו פונקציה חייבת לקבל ארגומנט כלשהו ולהחזיר ערך מוחזר כלשהו.
ג. לא ניתן להגדיר פונקציה ללא מתן שם (בזמן הגדרתה), אולם מאוחר יותר ניתן לשלוח את הפונקציה כפרמטר לפונקציה כלשהי ואף להחזיר פונקציה כערך מוחזר של חישוב כלשהו.
ד. בשפות אלו פונקציות יכולות להיות מוגדרות בזמן ריצה כך שגוף הפונקציה המוגדרת תלוי בקלט של התכנית.

סעיף ד' (8 נקודות):

אילו מהמשפטים הבאים נכונים לגבי **Bindings & Scope**?

א. האפשרות לתת שמות מזהים לערכים היא חשובה לקריאותה של התכנית כמו גם ליכולת ההפשטה (אבסטרקציה) של המתכנת.
ב. האפשרות לתת שמות מזהים לערכים מאפשרת לחסוך בזמן הריצה של התכנית ומקטינה את הסיכוי לבאגים.
ג. שם משתנה הוא binding instance בפעם הראשונה שבה הוא מופיע בתכנית.
ד. ניתן לבדוק האם קיימים מופעים חופשיים של משתנה כלשהו בתכנית עוד בשלב הניתוח התחבירי.
ה. באינטרפרטר שכתבנו הבדיקה האם קיימים מופעים חופשיים של משתנה כלשהו בתכנית מתקיימת בשלב הערכת התכנית, כלומר הודעת שגיאה על משתנה חופשי תוצג רק בזמן ריצת התכנית.
ו. שם משתנה הוא חפשי אם הוא מופיע מחוץ ל-scope של כל binding instance עם אותו שם, או שהוא משתנה מקומי של פונקציה ללא פרמטרים.

סעיף ה' (7 נקודות):

אילו מהמשפטים הבאים נכונים לגבי **Dynamic vs. lexical scoping**?

א. ב- dynamic scoping – בביטוי שהוא הגוף של פונקציה כלשהי – אסור שיהיו משתנים חופשיים.
ב. ב- lexical scoping – בביטוי שהוא הגוף של פונקציה כלשהי – אסור שיהיו משתנים חופשיים.
ג. ב- dynamic scoping – אם בגוף של פונקציה **f**, מתבצעת קריאה לפונקציה **g** שהוגדרה בשלב מוקדם יותר ובהנתן x מחזירה x+1, יתכן שבזמן הרצת התכנית יוחזר x+3 מהקריאה ל-g.
ד. ניתן לבדוק האם קיימים מופעים חופשיים של שם משתנה כלשהו בתכנית עוד בשלב הניתוח התחבירי.
ה. באינטרפרטר שכתבנו במודל הסביבות מבוצע ניתוח על-פי dynamic scoping.

סעיף ו' (5 נקודות):

אילו מהמשפטים הבאים נכונים לגבי מימושי האינטרפרטר שכתבנו עבור FLANG במודל ההחלפה ובמודל הסביבות?

א. במימוש במודל הסביבות ציפינו לקבל יעילות גבוהה יותר מאשר במודל ההחלפה מכיוון שהוא מטפל טוב יותר ברקורסית זנב.
ב. הטיפול בפונקציה eval בבנאי Id של FLANG הוא שונה מכיוון שבמודל ההחלפה אסור שיהיו שמות משתנים חופשיים בקוד ובמודל הסיביות מותרים שמות משתנים חופשיים בקוד.
ג. במימוש במודל ההחלפה אנו קוראים לפונקציה subst מתוך eval עם שם משתנה, ערך מחושב עבור המשתנה ו-עץ FLANG שבו עשויים להופיע מופעים חופשיים של שם המשתנה הנ"ל. הערך המוחזר מקריאה זו הוא עץ FLANG שבו אין אף מופע חופשי של שם המשתנה ששלחנו.
ד. במודל ההחלפה יכולנו לבטל לגמרי את השימוש בבנאי With על-ידי שימוש בבנאים Call ו-Fun. עובדה זו נשארת נכונה גם במודל הסביבות.

אוניברסיטת
אריאל
בשומרון

שאלה 3 — FLANG — (20 נקודות):

לצורך פתרון שאלה זו מצורף בסוף טופס המבחן קטע קוד עבור ה- interpreter של FLANG במודל הסביבות.

נתון ה-AST הבא:

```
(Call (Call (Fun x
                (Fun y (Call (Id y) (Id x))))
            (Num 5))
      (With z
           (Fun y (Add (Id y) (Num 1)))
           (Id z)))
```

סעיף א' (6 נקודות):

כתבו את הקוד בשפה שלנו, אשר אותו מתאר עץ התחביר האבסטרקטי הנ"ל (כלומר, המחרוזת אשר אם נפעיל עליה את parse, יתקבל העץ הנתון). הקפידו על כתיבה ברורה ועל הזחה (אינדנטציה) נכונה.

סעיף ב' (14 נקודות):

בתהליך ההערכה של AST זה במודל ה- environment (על-פי ה-interpreter התחתון מבין השניים המצורפים מטה) תופעל הפונקציה eval 14 פעמים. להלן תאור חסר של 14 ההפעלות הללו על-פי סדר הופעתן בחישוב. לכל הפעלה מספר i נתון לכם הפרמטרים האקטואלי הראשון (AST) ועליכם להשלים את הפרמטר האקטואלי השני (הסביבה ENV_i) של הפונקציה eval וכן את הערך המוחזר מהפעלה זו RES_i . הסבירו בקצרה כל מעבר.

```
1) (Call (Call (Fun x (Fun y (Call (Id y) (Id x))))
               (Num 5))
         (With z (Fun y (Add (Id y) (Num 1))) (Id z)))
   <<ENV1>>
   <<Res1>>
2) (Call (Fun x (Fun y (Call (Id y) (Id x)))) (Num 5))
   <<ENV2>>
   <<Res2>>
3) (Fun x (Fun y (Call (Id y) (Id x))))
   <<ENV3>>
   <<Res3>>
4) (Num 5)
   <<ENV4>>
   <<Res4>>
5) (Fun y (Call (Id y) (Id x)))
   <<ENV5>>
   <<Res5>>
```

```
6) (With z (Fun y (Add (Id y) (Num 1))) (Id z))
   <<ENV6>>
   <<Res6>>
7) (Fun y (Add (Id y) (Num 1)))
   <<ENV7>>
   <<Res7>>
8) (Id z)
   <<ENV8>>
   <<Res8>>
9) (Call (Id y) (Id x))
   <<ENV9>>
   <<Res9>>
10)   (Id y)
      <<ENV10>>
      <<Res10>>
11)   (Id x)
      <<ENV11>>
      <<Res11>>
12)   (Add (Id y) (Num 1))
      <<ENV12>>
      <<Res12>>
13)   (Id y)
      <<ENV13>>
      <<Res13>>
14)   (Num 1)
      <<ENV14>>
      <<Res14>>
```

שאלה 4 — הרחבת השפה WAE — (30 נקודות):

נרצה להרחיב את השפה **WAE** ולאפשר מספר משתנה של ארגומנטים לאופרטורים האריתמטיים. להלן דוגמאות לטסטים שאמורים לעבוד:

```
(test (run "{with {x 5} {* x x x}}") => 125)
(test (run "{+ {- 10 2 3 4} {/ 3 3 1} {*}}")  => 3)
(test (run "{+ 1 2 3 4}") => 10)
(test (run "{+}") => 0)
(test (run "{*}") => 1)
(test (run "{/ 4}") => 4)
(test (run "{- 4}") => 4)
(test (run "{/}")  =error> "bad syntax")
```

לצורך כך נרחיב את הדקדוק באופן הבא:

```
#| BNF for the WAE language:
       <WAE> ::= <num>
               | { + <WAE> ... }
               | { - <WAE> <WAE> ...}
               | { * <WAE> ... }
               | { / <WAE> <WAE> ...}
               | { with { <id> <WAE> } <WAE> }
               | <id>
  |#
```

סעיף א' (4 נקודות):

הוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים – 3 שורות קוד סה"כ לסעיף זה) ל –

```
(define-type WAE
    [Num  Number]
    [Add  (Listof WAE)]
    [Sub  —«fill-in 01»—]
    [Mul  —«fill-in 02»—]
    [Div  —«fill-in 03»—]
    [Id   Symbol]
    [With Symbol WAE WAE])
```

סעיף ב' (6 נקודות):

השתמשו בקוד הבא –

```
(: parse-sexpr* : (Listof Sexpr) -> (Listof WAE))
;; to convert a list of s-expressions into a list of WAEs
  (define (parse-sexpr* sexprs)
    (map parse-sexpr sexprs))
```

והוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים – 3 שורות קוד סה"כ לסעיף זה) ל –

```
(: parse-sexpr : Sexpr -> WAE)
  ;; to convert s-expressions into WAEs
  (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)])]
      [(list '+ (sexpr: args) ...)
       (Add (parse-sexpr* args))]
      [(list '- fst (sexpr: args) ...) —«fill-in 04»—]
      [—«fill-in 05»—]
      [—«fill-in 06»—]
      [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

סעיף ג' (8 נקודות):

השתמשו בהגדרות הפורמליות הבאות–

```
#| Formal specs for `subst':
   (`N' is a <num>, `E1', `E2' are <WAE>s, `x' is some <id>,
`y' is a *different* <id>)
        N[v/x]                  = N
        {+ E ...}[v/x]          = {+ E[v/x] ...]}
        {- E1 E ...}[v/x]          = {- E1[v/x] E[v/x] ...}
        {* E ...}[v/x]          = {* E[v/x] ...}
        {/ E1 E ...}[v/x]          = {/ E1[v/x] 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}
 |#
```

והוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים – 4 שורות קוד סה"כ לסעיף זה) ל –

```
(: subst : WAE Symbol WAE -> WAE)
  ;; 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)
    ;; convenient helper -- no need to specify `from' and `to'
    (: subst-helper : WAE -> WAE)
    (define (subst-helper x) (subst x from to))
    ;; helper to substitute lists
    (: subst* : (Listof WAE) -> (Listof WAE))
    (define (subst* exprs) (map subst-helper exprs))
    (cases expr
      [(Num n) expr]
```

```
      [(Add args) —«fill-in 07»—]
      [(Sub fst args) —«fill-in 08»—]
      [(Mul args) —«fill-in 09»—]
      [(Div fst args) —«fill-in 10»—]
      [(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 ד' (12 נקודות):

השתמשו בהגדרות הפורמליות הבאות (תוכלו להעזר גם בטסטים מתחילת השאלה) –

```
#| Formal specs for `eval':
       eval(N)          = N
       eval({+ E ...}) = eval(E) + ...
       eval({- E1 E ...}) = eval(E1) - (eval(E) + ...)
       eval({* E ...}) = eval(E) * ...
       eval({/ E1 E ...}) = eval(E1) / (eval(E) * ...)
       eval(id)         = error!
       eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
  |#
```

והוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים – 4 שורות קוד סה"כ לסעיף זה) ל –

```
(: eval : WAE -> Number)
  ;; evaluates WAE expressions by reducing them to numbers
  (define (eval expr)
    (cases expr
      [(Num n) n]
      [(Add args) —«fill-in 11»—]
      [(Sub fst args) —«fill-in 12»—]
      [(Mul args) —«fill-in 13»—]
      [(Div fst args) —«fill-in 14»—]
      [(With bound-id named-expr bound-body)
       (eval (subst bound-body
                    bound-id
                    (Num (eval named-expr))))]
      [(Id name) (error 'eval "free identifier: ~s" name)]))
```

הדרכה: השתמשו בפונקציות map **ו-** foldl **של** RACKET **המתוארות מטה.**
הערה: אין צורך לטפל בחלוקה באפס.

הפונקציה map:

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))
```

הפונקציה foldl:

קלט: פרוצדורה proc, ערך התחלתי init ורשימה lst

פלט: ערך סופי (מאותו טיפוס שמחזירה הפרוצדורה proc) שנוצר ע"י הפעלת הפרוצדורה proc על כל אחד מאיברי הרשימה lst תוך שימוש במשתנה ששומר את הערך שחושב עד כה – משתנה זה מקבל כערך התחלתי את הערך של init. (ההסבר הבא הוא כללי יותר – כי למעשה הפונקציה foldl יכולה לטפל במספר רשימות – לצורך השאלה הנתונה לא תזדקקו לשימוש כזה)

(foldl proc init lst ...+) → any/c
proc : procedure?

init : any/c

lst : list?

Like map, foldl applies a procedure to the elements of one or more lists. Whereas map combines the return values into a list, foldl combines the return values in an arbitrary way that is determined by proc.

דוגמאות:

```
> (foldl + 0 '(1 2 3 4))
10
> (foldl cons '() '(1 2 3 4))
'(4 3 2 1)
```

אוניברסיטת
אריאל
בשומרון

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