אוניברסיטת
אריאל
בשומרון

פקולטה: מדעי הטבע

מחלקה: מדעי המחשב ומתמטיקה

שם הקורס: שפות תכנות

קוד הקורס: 2-7036010-1/3

תאריך בחינה: _19/10/2014_ סמ' _ק'___מועד __א'__

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 **(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 שני קטעי קוד עבור ה- interpreter של FLANG בסוף טופס המבחן. הראשון – במודל ה-substitution והשני במודל הסביבות.

אוניברסיטת
אריאל
בשומרון

שאלה 1 — BNF — (20 נקודות): נתון הדקדוק (BNF) הבא:

```
<ME> ::= <AB>
         | + <ME> <ME>
         | - <ME> <ME>
         | * <ME> <ME>
         | / <ME> <ME>
<AB> ::= a | b | x | y | z
```

סעיף א' (4 נקודות): בחרו תשובה **אחת** נכונה:

א. השפה שמגדיר הדקדוק הינה שפה עילית שבה ניתן להציב ערכים בביטויים אריתמטיים.

ב. השפה שמגדיר הדקדוק הינה שפת הביטויים אריתמטיים עם ארבעה סימני פעולות חשבון ומספרים בשפה RACKET.

ג. בשפה שמגדיר הדקדוק ישנן רק מילים באורך סופי.

ד. לא ניתן לייצר מילים בשפה שמגדיר הדקדוק ללא שימוש במספרים של RACKET.

סעיף ב' (10 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. * * 1 2 3
2. +(+ [a] [+ {b} {b}])(x)
3. { { a + b } / { x – y } }
4. +(+ [+ {+ <+ |a||a|><a>}{a}][a])(a)
5. axy

סעיף ג' (6 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' (5 נקודות): (סמנו את כל התשובות הנכונות)

נתון הקוד הבא ב-Racket?

```
(: foo : Number (Number->Boolean) (Listof Numbers) -> Number)
(define (foo x p lst)
  (if (null? lst)
      x
      (let ([f (p (first lst))])
        (if f
            (foo (- x (first lst)) p (rest lst))
            (foo (+ x (first lst)) p (rest lst))))))
```

א. כל הקריאות הרקורסיביות הן קריאות זנב.
ב. זוהי אינה רקורסיית זנב, כיוון שבחישוב יש פתיחת סביבה לוקאלית של let השקולה להפעלה מקומית של פונקציה אנונימית.
ג. f תמיד מקבל או ערך true או ערך false.
ד. אם נשלח עבור הפרמטר p את הפונקציה zero? של RACKET, תמיד נקבל את ערך הפרמטר x ועוד סכום הערכים ברשימה lst.

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

אילו מהמשפטים הבאים נכונים לגבי תהליך ה-Parsing?

א. תהליך ה-Parsing חייב להיות מנותק מתהליך ההערכה של התכנית. בפרט, הראשון חייב להסתיים לפני שהשני מתחיל.
ב. בסוף התהליך אנו יודעים מה הערך המוחזר מהרצת התכנית.
ג. תהליך ה-Parsing אינו יכול להתבצע אם יש רב-משמעיות בדקדוק, כיוון שלא נדע איזה עץ תחביר אבסטרקטי עלינו לייצר.
ד. במימוש שלנו, במודל ההחלפה – בתום תהליך ה-Parsing מתקבל FLANG. לעומת זאת, במודל הסביבות – בתום תהליך ה-Parsing מתקבל טיפוס אחר.

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

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

א. בשפה כזו, פונקציה שנשלחת כפרמטר לפונקציה אחרת, אינה יכולה להיות רקורסיבית, אחרת פונקצית ההערכה (eval) עלולה להכנס ללולאה אינסופית.

ב. בשפות הללו פונקציה אינה חייבת לקבל ארגומנט כלשהו או להחזיר ערך מוחזר כלשהו.

ג. ישנן שפות כנ"ל שאינן מאפשרות 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 פונקציה מקבלות את הפרמטר דרך רגיסטרים, והפונקציה עצמה הינה כתובת בזיכרון.

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

אילו מהמשפטים הבאים נכונים לגבי מימושי האינטרפרטר שכתבנו עבור FLANG במודל ההחלפה, ה-substitution cache ובמודל הסביבות?

א. במימוש במודל ה-substitution cache ציפינו לקבל יעילות גבוהה יותר מאשר במודל ההחלפה מכיוון שהוא דורש פחות סריקות של הקוד בשלב ההרצה של גוף הפונקציה.

ב. השוני המרכזי בין מימושי האינטרפרטר שכתבנו עבור FLANG במודל ה-substitution cache ובמודל הסביבות, הוא ביעילות של הפונקציה eval הנובע משימוש בטיפוס מתוחכם של סביבה כתחליף לרשימה של רשימות.

ג. במימוש במודל ההחלפה והסביבות קיבלנו lexical scoping ובמימוש במודל ה- substitution cache קיבלנו dynamic scoping. אולם, במודל ההחלפה היה באג, שהתנהג כמו dynamic scoping במקרה מסויים.

ד. הפונקציה eval פעלה באופן זהה עבור פונקציות במימושי האינטרפרטר שכתבנו עבור FLANG במודל ההחלפה וה-substitution cache. בפרט, במקרה זה היא החזירה את אותו ערך שקיבלה.

שאלה 3 — With vs. Call — (39 נקודות):

נתון הקוד הבא:

```
(run "{call {fun {x}
               {call {fun {y}
                        {- y x}}
                     {* x 5}}}
            {+ 2 4}}")
```

סעיף א' (10 נקודות):

החליפו את הקוד הנ"ל בקוד שקול בו לא מופיעה המילה call – לצורך כך השתמשו במילה with.
הקפידו על הזחה נכונה.

```
(run "{with {x {+ 2 4}}
        {with {y {* x 5}}
           {- y x}}}")
```

סעיף ב' (4 נקודות):

האם לכל קוד ניתן לבצע את ההחלפה שביצעתם בתשובה לסעיף א'? הסבירו את תשובתכם (לא יותר משלוש שורות הסבר).

לא

סעיף ג' (13 נקודות):

החליפו את הקוד הבא מתוך הפונקציה eval (באינטרפרטר של FLANG במודל ה - substitution) – בשורת קוד שאינה מכילה הפעלה של subst ומכילה הפעלה יחידה של eval (במקום שתי הפעלות):

```
[(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)]
```

הדרכה: בנו מחדש את ה-FLANG שנבנה עם בנאי Call בעזרת בנאי With. עליכם לשנות רק את השורה הרביעית עד השורה השביעית.

```
[(Call fun-expr arg-expr)
       (let ([fval (eval fun-expr)])
         (cases fval
           [(Fun bound-id bound-body)
            (eval (With bound-id arg-expr bound-body))]
           [else (error 'eval "`call' expects a function, got:
~s"
                                                        fval)]
```

סעיף ד' (12 נקודות):

בתהליך ההערכה של הקוד מסעיף א' במודל ה- environment (על-פי ה-interpreter התחתון מבין השניים המצורפים מטה) תופעל הפונקציה eval 13 פעמים. להלן תאור חסר של 13 ההפעלות הללו על-פי סדר הופעתן בחישוב. עבור חמש ההפעלות הראשונות – לכל הפעלה מספר *i* נתון לכם הפרמטרים האקטואלי הראשון (AST_i) ועליכם להשלים את הפרמטר האקטואלי השני (הסביבה ENV_i) של הפונקציה eval וכן את הערך המוחזר מהפעלה זו RES_i . מההפעלה החמישית ואילך, עליכם להשלים **גם** את AST_i *(כמו גם* את הפרמטר האקטואלי השני (הסביבה ENV_i) של הפונקציה eval וכן את הערך המוחזר מהפעלה זו RES_i *)*.

הסבירו בקצרה כל מעבר.

```
- : Real
ActParmam(1) = (Call (Fun x (Call (Fun y (Sub (Id y) (Id
x))) (Mul (Id x) (Num 5)))) (Add (Num 2) (Num 4)))
 Env(1) = (EmptyEnv)
 Res(1) = (NumV 24)

ActParmam(2) = (Fun x (Call (Fun y (Sub (Id y) (Id x)))
(Mul (Id x) (Num 5))))
 Env(2) = (EmptyEnv)
Res(2) = (FunV x (Call (Fun y (Sub (Id y) (Id x))) (Mul
(Id x) (Num 5))) (EmptyEnv))


ActParmam(3) = (Add (Num 2) (Num 4))
 Env(3) = (EmptyEnv)
Res(3) = (NumV 6)


ActParmam(4) = (Num 2)
 Env(4) = (EmptyEnv)
Res(4) = (NumV 2)

 ActParmam(5) = (Num 4)
 Env(5) = (EmptyEnv)
Res(5) = (NumV 4)
```

```
ActParmam(6) = (Call (Fun y (Sub (Id y) (Id x))) (Mul (Id
x) (Num 5)))
 Env(6) = (Extend x (NumV 6) (EmptyEnv))
Res(6) = (NumV 24)

ActParmam(7) = (Fun y (Sub (Id y) (Id x)))
 Env(7) = (Extend x (NumV 6) (EmptyEnv))
Res(7) = (FunV y (Sub (Id y) (Id x)) (Extend x (NumV 6)
(EmptyEnv)))


 ActParmam(8) = (Mul (Id x) (Num 5))
 Env(8) = (Extend x (NumV 6) (EmptyEnv))
Res(8) = (NumV 30)


ActParmam(9) = (Id x)
 Env(9) = (Extend x (NumV 6) (EmptyEnv))
Res(9) = (NumV 6)

 ActParmam(10) = (Num 5)
 Env(10) = (Extend x (NumV 6) (EmptyEnv))
Res(10) = (NumV 5)


ActParmam(11) = (Sub (Id y) (Id x))
 Env(11) = (Extend y (NumV 30) (Extend x (NumV 6)
(EmptyEnv)))
Res(11) = (NumV 24)


ActParmam(12) = (Id y)
 Env(12) = (Extend y (NumV 30) (Extend x (NumV 6)
(EmptyEnv)))
Res(12) = (NumV 30)


 ActParmam(13) = (Id x)
 Env(13) = (Extend y (NumV 30) (Extend x (NumV 6)
(EmptyEnv)))
Res(13) = (NumV 6)
```

שאלה 4 — הרחבת השפה — (29 נקודות):

לצורך פתרון שאלה זו נעזר בקוד ה- interpreter של FLANG במודל ה substitution, המופיע בסוף טופס המבחן (העליון מבין השניים המופיעים שם).

נרצה להרחיב את השפה ולאפשר מציאת עצרת של משתנה ***n***.

תזכורת: $0! = 1$ ולכל $n > 0$ מתקיים $n! = 1 \cdot \ldots \cdot (n-1)$.

להלן דוגמאות לטסטים שאמורים לעבוד:

```
(test (run "{fact 2}") => 2)

(test (run "{+ 2 {fact {+ 3 2}}}") => 122)

 (test (run "{call {fun {x}

                {+ {fact x} 1}}

               4}")        => 25)
```

סעיף א' (10 נקודות):

ראשית נגדיר פונקציה עבור עצרת בשפה **pl** (הגרסה של **Racket** בה אנו משתמשים). שורת ההכרזה על הפונקציה תהיה

```
(: factorial : Number -> Number)
```

ניתן להניח (ואין צורך בבדיקת נכונות) שהקלט x הוא מספר טבעי (יכול להיות 0). הפונקציה תחשב את סכום **העצרת** של x. כתבו את הפונקציה כך שכל הקריאות הרקורסיביות **הן קריאות זנב**.

להלן דוגמאות לטסטים שאמורים לעבוד:

```
(: factorial : Number -> Number)
(: fact-helper : Number Number -> Number)

(define (factorial n)
  (if (< n 0)
      (error 'factorial "expects a non-negative number,
got: ~s" n)
      (fact-helper (round n) 1)))
(define (fact-helper n acc)
  (if (zero? n)
      acc
      (fact-helper (- n 1)
                   (* acc  n))))
```

```
(test (factorial 1) => 1)
(test (factorial 2) => 2)
(test (factorial 3) => 6)
```

סעיף ב' (2 נקודות):

הרחיבו את הדקדוק בהתאם (הוסיפו את הקוד הנדרש היכן שכתוב –«fill-in»):

הערה: שימו לב לשוני בין השמות fact ו- factorial.

```
#|  The grammar:
    <FLANG> ::= <num>
              | { + <FLANG> <FLANG> }
              | { - <FLANG> <FLANG> }
              | { * <FLANG> <FLANG> }
              | { / <FLANG> <FLANG> }
              | { with { <id> <FLANG> } <FLANG> }
              | <id>
              | { fun { <id> } <FLANG> }
              | { call <FLANG> <FLANG> }
                  | { fact <FLANG> } ; Add
|#
```

סעיף ג' (2 נקודות):

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

```
(define-type FLANG
... מטה interpreter -ה קוד ראו      ...
    [Fact FLANG ]) ; Add
```

סעיף ד' (3 נקודות):

הוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים) ל –

```
(: parse-sexpr : Sexpr -> FLANG)
  ;; to convert s-expressions into FLANGs
  (define (parse-sexpr sexpr)
    (match sexpr
... מטה interpreter -ה קוד ראו      ...
      [(list 'fact  arg) (Fact (parse-sexpr arg))]     ; Add
      [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

סעיף ה' (4 נקודות):

הוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים) ל –

```
(: 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
... ראו קוד ה- interpreter מטה                ...

      [(Fact a) (Fact (subst a from to))])) ; Add
```

סעיף ו' (8 נקודות):

עתה נרצה לאפשר לפונקציה eval לטפל במקרה שנוסף עבור פעולת **fact**.

הערה: אינכם צריכים לטפל במקרה שבו ערך הביטוי עליו מופעלת **fact** הינו מספר לא טבעי. אולם, עליכם עדיין לוודא שאינו פונקציה.

לצורך כך נגדיר פונקצית עזר: **הוסיפו את הקוד הנדרש במקום המתאים**

```
(: fact-op : FLANG -> FLANG)
  (define (fact-op expr)
    (: Num->number : FLANG -> Number)
    (define (Num->number e)
      (cases e
        [(Num n)  n]
        [else (error 'fact-op "expects a number, got: ~s" e)]))
    (Num (factorial (Num->number expr))))  ; Add
```

הוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים) ל –

```
(: eval : FLANG -> FLANG)
  ;; evaluates FLANG expressions by reducing them to *expressions*
  (define (eval expr)
    (cases expr
... ראו קוד ה- interpreter מטה                ...

      [(Fact arg) (fact-op (eval arg))]))   ; Add
```

הדרכה: השתמשו בפונקציה שלכם מסעיף א'.

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)
```