

פקולטה: מדעי הטבע

מחלקה: מדעי המחשב ומתמטיקה

שם הקורס: שפות תכנות

קוד הקורס: 2-7036010

תאריך בחינה: 08/07/2015 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 (במחברת התשובות בלבד ולא על גבי טופס המבחן עצמו).
- בכל שאלה או סעיף (שבהם נדרשת כתיבה, מעבר לסימון תשובות נכונות), ניתן לכתוב – לא יודעת/ (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 109 נקודות במבחן.
- לנוחיותכם מצורפים שני קטעי קוד עבור ה- interpreter של FLANG בסוף טופס המבחן. הראשון – במודל ה-substitution והשני במודל הסביבות.

שאלה 1 — BNF — (20 נקודות):

הערה: שאלה זו קשורה לשאלה 3 בהמשך המבחן. בשאלה 3 נכתוב את שפת הרגיסטרים ROL המאפשרת פעולות לוגיות (על רגיסטרים, שהם למעשה סדרות של אפסים ואחדים). בשאלה זו, עליכם לכתוב תחביר עבור השפה, על-פי הכללים הבאים ועל-פי הדוגמאות מטה (למעשה, הדוגמאות מספיקות).

- כל ביטוי בשפה, הוא מהצורה "{ reg-len = len A }", כאשר len הוא מספר טבעי ו-A הוא ביטוי המתאר סדרת פעולות על רגיסטרים. ביטוי כזה מקיים את הכללים הבאים:
 - סדרה של אפסים ואחדות – עטופים בסוגריים מסולסלים (בהמשך, נחשוב על סדרה כזו כייצוג של ערך נתון של רגיסטר) – היא חוקית כסדרת פעולות על רגיסטרים.
 - אם B, A סדרות פעולות על רגיסטרים, אז גם הביטוי המתקבל ע"י שימוש באופרטור and או אופרטור or כאשר B, A הם האופרנדים, ועטיפת הביטוי כולו בסוגריים מסולסלים – הוא חוקי כסדרת פעולות על רגיסטרים. גם הביטוי המתקבל ע"י שימוש באופרטור shl כאשר A הוא האופרנד, ועטיפת הביטוי כולו בסוגריים מסולסלים – הוא חוקי כסדרת פעולות על רגיסטרים.
 - ביטויי with וביטויי fun הם חוקיים בדומה מאוד למה שעשינו עבור השפה FLANG (כמובן שתתי הביטויים שבעזרתם יוצרים ביטוי חדש, הם עתה סדרת פעולות על רגיסטרים, במקום ביטויים בשפה FLANG).

להלן דוגמאות לביטויים חוקיים בשפה:

```
"{ reg-len = 4  {1 0 0 0} }"
"{ reg-len = 4  {shl {1 0 0 0}} }"
"{ reg-len = 4  {and {shl {1 0 1 0}} {shl {1 0 1 0}}} }"
"{ reg-len = 4  { or {and {shl {1 0 1 0}} {shl {1 0 0 1}}} {1 0 1 0}}} }"
"{ reg-len = 2  { or {and {shl {1 0}} {1 0}} {1 0}}} }"
"{ reg-len = 2  { with {x { or {and {shl {1 0}} {1 0}} {1 0}}}
                    {shl x}}} { reg-len = 3
                    {with {identity {fun {x} x}}
                    {with {foo {fun {x} {or x {1 1 0}}}}
                    {call {call identity foo} {0 1 0}}}}} }"
"{ reg-len = 4  {or {1 1 1 1} {0 1 1}}} "
```

סעיף א' BNF (13 נקודות):

כתבו דקדוק עבור השפה ROL בהתאם להגדרות ולדוגמאות מעלה. תוכלו להשתמש ב- $\langle \text{num} \rangle$ בדומה לשימוש שלו כ-FLANG (אל תשתמשו בו עבור ביטים). אתם מוזמנים להשתמש בשלד הדקדוק הבא.

#| BNF for the ROL language:

$\langle \text{ROL} \rangle ::=$

$\langle \text{RegE} \rangle ::=$

$\langle \text{Bits} \rangle ::=$

|#

סעיף ב' (7 נקודות):

עבור כל אחת מן המילים הבאות, קבעו האם ניתן לייצר את המילה מהדקדוק שכתבתם בסעיף א'. אם תשובתכם חיובית, הראו תהליך גזירה עבורה (ניתן להראות עץ גזירה – שימו לב שזהו אינו עץ תחביר אבסטרקטי – בפרט, לא אמורים להופיע בתוכו פונקציות של RACKET). אם תשובתכם שלילית, הסבירו מדוע לא ניתן לגזור את המילה.

1. $\{ \text{reg-len} = 1 \quad \{ + \ 1 \ 22 \} \}$
2. $\{ \text{reg-len} = 3 \quad \{ \text{or} \ \{ 1 \ 1 \ 0 \} \ \{ 0 \ 1 \} \} \}$
3. $\{ \text{reg-len} = 3 \quad \{ \text{with} \ \{ f1 \ \{ \text{fun} \ \{ x \} \ \{ \text{shl} \ \{ \text{and} \ x \ y \} \} \} \} \} \}$
 $\{ \text{call} \ \{ \text{call} \ f1 \ \{ 0 \ 0 \ 1 \} \} \ \{ 0 \ 1 \ 0 \} \} \}$
4. $\{ \text{reg-len} = 3 \quad \{ \text{call} \ x \ z \} \}$

שאלה 2 — (40 נקודות):

נתון הקוד הבא:

```
(run "{call {with {f {fun {x} x}} f}
      {with {z 2}
        {with {f {fun {y} {* 6 y}}}
          {call f z}}}}")
```

סעיף א' (14 נקודות):

תארו את הפעולות הפונקציה eval בתהליך ההערכה (יש 15 הפעולות) של הקוד מעלה במודל ההחלפות (על-פי ה-*interpreter* העליון מבין השניים המצורפים מטה) - באופן הבא – לכל הפעלה מספר i תארו את הפרמטר האקטואלי ה- i (AST_i) וכן את הערך המוחזר מהפעלה זו (RES_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)))
RES1 = (Num 3)
AST2 = (Num 1)
RES2 = (Num 1)
AST3 = (Add (Num 1) (Num 2))
RES3 = (Num 3)
AST4 = (Num 1)
RES4 = (Num 1)
AST5 = (Num 2)
RES5 = (Num 2)
```

Final result: 3

סעיף ב' (4 נקודות):

מה לדעתכם היה קורה לו היינו מבצעים את ההערכה במודל substitution-cache? האם התשובה הייתה זהה או שונה? מדוע? (אין צורך לבצע הערכה; כתבו עד שלוש שורות)

סעיף ג' – Eliminating free instances במודל הסביבות (22 נקודות):

קוד בשפה שכתבנו FLANG המכיל מופעים חופשיים של שמות מזהים הוא קוד שגוי – עליו יש להוציא הודעת שגיאה. באינטרפרטר שכתבנו במודל ההחלפות, הודעת השגיאה ניתנת בזמן הערכת התוכנית. ברצוננו לכתוב 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.

הקוד הבא מבוסס על הפונקציה `eval` (באינטרפרטר של FLANG במודל הסביבות) – השלימו את הקוד במקומות החסרים. הניחו שהקוד נכתב כתוספת לאינטרפרטר הנ"ל ולכן הוא מכיר את כל הפונקציות והטיפוסים שבו. את הפונקציה `lookup` נצטרך לשנות כדי שתחזיר גם היא ערך בוליאני.

```
(: lookup : Symbol ENV -> Boolean)
(define (lookup name env)
  (cases env
    [(EmptyEnv) <--fill in 1-->]
    [(Extend id val rest-env)
     (if (eq? id name) <--fill in 2-->)])))
```

```
(: containsFreeInstance? : FLANG ENV -> Boolean)
;; checks for free instances -- without evaluating the
expression
(define (containsFreeInstance? expr env)
  (cases expr
    [(Num n) <--fill in 3-->]
    [(Add l r) (or <--fill in 4-->)]
    [(Sub l r) <--fill in 5-->]
    [(Mul l r) <--fill in 6-->]
    [(Div l r) <--fill in 7-->]
    [(With bound-id named-expr bound-body)
     (or <--fill in 8-->)]
    [(Id name) <--fill in 9-->]
    [(Fun bound-id bound-body)
     <--fill in 10-->]
    [(Call fun-expr arg-expr)
     <--fill in 11-->]))
```

שימו לב: אם נגדיר את פונקציית המעטפת הבאה –

```
(: check-code : String -> Boolean)
(define (check-code str)
  (containsFreeInstance? (parse str) (EmptyEnv)))
```

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 3 – Interpreter עבור השפה ROL – (49 נקודות):

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

בהמשך לשאלה ראשונה, נרצה לממש Interpreter עבור השפה ROL במודל הסביבות ולאפשר שימוש בביטויים ופעולות על רגיסטרים.

להלן דוגמאות לטסטים שאמורים לעבוד:

```
:: tests
(test (run "{ reg-len = 4 {1 0 0 0}}") => '(1 0 0 0))
(test (run "{ reg-len = 4 {shl {1 0 0 0}}}") => '(0 0 0 1))
(test (run "{ reg-len = 4 {and {shl {1 0 1 0}}{shl {1 0 1 0}}}") => '(0 1 0 1))
(test (run "{ reg-len = 4 { or {and {shl {1 0 1 0}} {shl {1 0 0 1}} {1 0 1 0}}}") => '(1 0 1 1))
(test (run "{ reg-len = 2 { or {and {shl {1 0}} {1 0}} {1 0}}}") => '(1 0))
(test (run "{ reg-len = 4 {with {x {1 1 1 1}} {shl y}}}") =error> "no binding for")
(test (run "{ reg-len = 2 { with {x { or {and {shl {1 0}} {1 0}} {1 0}} {shl x}}}") => '(0 1))
(test (run "{ reg-len = 4 {or {1 1 1 1} {0 1 1 1}}}") =error> "wrong number of bits in")
(test (run "{ reg-len = 0 {}") =error> "Register length must be at least 1")
(test (run "{ reg-len = 3
  {with {identity {fun {x} x}}
    {with {foo {fun {x} {or x {1 1 0}}}}
      {call {call identity foo} {0 1 0}}}}")
  => '(1 1 0))
(test (run "{ reg-len = 3
  {with {x {0 0 1}}
    {with {f {fun {y} {and x y}}}
      {with {x {0 0 0}}
        {call f {1 1 1}}}}}")
  => '(0 0 1))
```

לצורך כך נגדיר טיפוס של ביט וטיפוס של רשימה של ביטים.

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

סעיף א' הטיפוס RegE (8 נקודות):

כיוון שהחלק המרכזי בניתוח הסינטקטי הוא החלק של RegE, נממש רק אותו. בהמשך לשאלה 1 ולטסטים מעלה, כתבו את הטיפוס בהתאם. הוסיפו את הקוד הנדרש (היכן שכתוב «fill-in») – ל

```
;; RegE abstract syntax trees
(define-type RegE
  [Reg <--fill in 1-->]
  [And <--fill in 2-->]
  [Or <--fill in 3-->]
  [Shl <--fill in 4-->]
  [Id <--fill in 5-->]
  [With <--fill in 6-->]
  [Fun <--fill in 7-->]
  [Call <--fill in 8-->])
```

סעיף ב' parse (15 נקודות):

כתבו את הפונקציה parse-sexpr בהתאם. בפונקציה זו, עליכם גם לבדוק שאורך כל רגיסטר הוא בהתאם למה שכתוב בקוד וכן שהוא לפחות 1 (אסור לכתוב תכנית עם רגיסטרים ריקים). הפונקציה הבאה, הינה פונקציית עזר טכנית. השתמשו בה.

```
;; 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)))]))
```

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

```
(: parse-sexpr : Sexpr -> RegE)
;; to convert s-expressions into RegEs
(define (parse-sexpr sexpr)
  (match sexpr
    [(list 'reg-len '= (number: len) reg-sexpr)
     (if <--fill in 1--> ;; we do not allow this
         (error <--fill in 2-->
               <--fill in 3-->)
         [else (error 'parse-sexpr "bad syntax in ~s" sexpr)])])
```

```
(: parse-sexpr-RegL : Sexpr Number -> RegE)
;; to convert s-expressions into RegEs
(define (parse-sexpr-RegL sexpr reg-len)
  (match sexpr
    [(list (and a (or 1 0)) ... )
     (if <--fill in 4--> ;; verifying length
       (<--fill in 5-->)
       (error <--fill in 6-->)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
     (match sexpr
       [(list 'with (list (symbol: name) named) body)
        <--fill in 7-->]
       [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
    [(list 'and lreg rreg) <--fill in 8-->]
    [(list 'or lreg rreg) <--fill in 9-->]
    [(list 'shl reg) <--fill in 10-->]
    [(cons 'fun more)
     (match sexpr
       [(list 'fun (list (symbol: name)) body)
        <--fill in 11-->]
       [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(list 'call fun arg) (Call <--fill in 12-->)]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

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

סעיף ג' – פונקציות עזר (6 נקודות):

כדי להגדיר את eval נזדקק לפונקציות העזר הבאות:

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

```
;; 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--> ;; using logical and

(: bit-or : BIT BIT -> BIT) ;; Aithmetic or
(define (bit-or a b)
  <--fill in 2--> ;; Using logical or
```


סעיף ד' eval (20 נקודות):

נתונים המיפוסים הנדרשים ונתונות ההגדרות הפורמליות לסמנטיקה של השפה. שימו לב שאנחנו במודל הסביבות.

```
;; Types for environments, values, and a lookup function

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

(define-type VAL
  [RegV Bit-List]
  [FunV Symbol RegE ENV])

(: lookup : Symbol ENV -> VAL)
(define (lookup name env)
  (cases env
    [(EmptyEnv) (error 'lookup "no binding for ~s" name)]
    [(Extend id val rest-env)
     (if (eq? id name) val (lookup name rest-env))]))

|# Formal specs for `eval':
    Evaluation rules:
    eval(Reg,env) = Reg
    eval({and E1 E2},env) =
      (<x1 bit-and y1> <x2 bit-and y2> ... <xk bit-and yk>)
      where eval(E1,env) = (x1 x2 ... xk) and eval(E2,env) = (y1 y2 ... yk)
    eval({or E1 E2},env) =
      (<x1 bit-or y1> <x2 bit-or y2> ... <xk bit-or yk>)
      where eval(E1,env) = (x1 x2 ... xk) and eval(E2,env) = (y1 y2 ... yk)
    eval({shl E},env) = (x2 ... xk x1), where eval(E,env) = (x1 x2 ... xk)
    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

|#

(: RegV->bit-list : VAL -> Bit-List)
(define (RegV->bit-list v)
  (cases v
    [(RegV n) n]
    [else (error 'RegV->bit-list "expects a bit-list, got: ~s" v)]))
```

עתה נרצה לאפשר לפונקציה eval לטפל בביטויים לוגיים ע"פ הגדרות אלו והטסטים מעלה. הוסיפו את הקוד הנדרש (היכן שכתוב «fill-in»)- ל

```
(: reg-arith-op : (BIT BIT -> BIT) VAL VAL -> VAL)
;; 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 (reg-arith-op op reg1 reg2)
  (: bit-arith-op : Bit-List Bit-List -> Bit-List)
  ;; Consumes two bit-lists and uses the binary bit operation 'op'.
  ;; It returns the bit-list obtained by applying op on the
  ;; i'th bit of both registers for all i.
  (define (bit-arith-op bl1 bl2)
    (if <--fill in 1-->
        null
        (cons <--fill in 2-->
              (RegV <--fill in 3-->))))
```

הפונקציה הבאה מממשת shift-left על רשימה של BIT. כלומר הזזה מחזורית שמאלה.

```
(: shift-left : Bit-List -> Bit-List)
;; Shifts left a list of bits (once)
(define (shift-left bl)
  (if (null? bl)
      <--fill in 4-->
      <--fill in 5-->))

(: eval : RegE ENV -> VAL)
;; evaluates RegE expressions by reducing them to bit-lists
(define (eval expr env)
  (cases expr
    [(Reg n) <--fill in 6-->]
    [(And l r) <--fill in 7-->]
    [(Or l r) <--fill in 8-->]
    [(Shl reg) (RegV (shift-left (RegV->bit-list <--fill in 9-->)))]
    [(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)
     <--fill in 10-->]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr env)])
       (cases fval
         [<--fill in 11-->]
         [else (error 'eval "`call' expects a function, got: ~s"
                       fval)])))]))
```

```

---<<<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) named) body)
        (Fun name (parse-sexpr named) (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(cons 'call more)
     (match sexpr
       [(list 'call (list (symbol: name) named) body)
        (Call name (parse-sexpr named) (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `call' syntax in ~s" sexpr)])]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

```
(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)])))

```