פקולטה: מדעי הטבע

מחלקה: מדעי המחשב ומתמטיקה

שם הקורס: שפות תכנות

קוד הקורס: 2-7036010

תאריך בחינה: 29/06/2016 סמ' ב' מועד א'

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 יודע/ת** (מבלי להוסיף דבר מעבר לכך) – ולקבל 20% מהניקוד על השאלה או הסעיף.
- אפשר להסתמך על סעיפים קודמים גם אם לא עניתם עליהם.
- ניתן להשיג עד 109 נקודות במבחן. לצורך כך, יש לענות על כל השאלות.
- לנוחיותכם מצורפים שלושה קטעי קוד עבור ה- interpreter של FLANG בסוף טופס המבחן. הראשון – במודל ה-substitution, השני במודל הסביבות והשלישי במודל ה-Substitution-cache.

אוניברסיטת
אריאל
בשומרון

שאלה 1 — BNF — (20 נקודות):

הערה: שאלה זו קשורה לשאלה 4 בהמשך המבחן. בשאלה 4 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). השאלה מתחלקת לשני חלקים. בחלק ראשון, נתמקד בגרסה בסיסית יותר שכבר פגשתם בעבר. בהמשך נרחיב את השפה.

תיאור הגרסה הבסיסית: כל ביטוי בשפה, הוא מהצורה "{ reg-len = `len` A}", באשר 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 A, B 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 מאד 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 הבסיסית:

```
"{ 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 (10 נקודות):

כתבו דקדוק עבור השפה ROL בהתאם להגדרות ולדוגמאות מעלה. תוכלו להשתמש ב-<num> בדומה לשימוש שלו כ-FLANG (אל תשתמשו בו עבור ביטים). אתם מוזמנים להשתמש בשלד הדקדוק הבא.

מספרו כל כלל שאתם מגדירים.

```
#| BNF for the ROL language:

    <ROL>  ::=

    <RegE> ::=


    <Bits> ::=

  |#
```

סעיף ב' (6 נקודות):

נרצה להרחיב את השפה ROL ולאפשר שימוש בביטויים ופעולות לוגיות (על ערכים בוליאניים). ביתר פירוט – נוסיף ביטויי **if** (על-פי תחביר מיוחד המודגם בביטויים מטה); אופרטור בינארי – **geq?**, אופרטור אונארי – **maj?** וערכי **true** ו – **false** (כאן, לא נרשה #t, #f).

להלן דוגמאות לביטויים חוקיים בשפה המורחבת (בפרט, כל ביטוי חוקי בשפה לפני הרחבת השפה, ימשיך להיות חוקי גם לאחר הרחבת התחביר):

```
"{ reg-len =  4  true }"
"{ reg-len = 3  {if {geq? {1 0 1} {1 1 1}} {0 0 1} {1 1 0}}}"
"{ reg-len = 4  {if {maj? {0 0 1 1}} {shl {1 0 1 1}} {1 1 0 1}}}"
"{ reg-len = 4  {if false {shl {1 0 1 1}} {1 1 0 1}}}"
"{ reg-len = 4 {if true {0 1 0 1} false }}
```

הרחיבו את הדקדוק של השפה ROL, שכתבתם בסעיף א', בהתאם לדוגמאות מעלה (הוסיפו את הכללים הנדרשים על מנת לאפשר ביטויי: **(true, false, if, maj?, geq?.**

מספרו כל כלל.

סעיף ג' (4 נקודות):

כתבו מילה חוקית בשפה אשר מכילה: לפחות ביטוי if אחד (שחלק התנאי בו אינו מילה אחת – בפרט, לא true ולא false) ולפחות שלושה אופרטורים שונים (חלקם לוגיים).

הראו את תהליך הגזירה עבור המילה שבחרתם (מספרו את הכללים בדקדוק וכתבו מעל כל גזירה באיזה כלל השתמשתם).

שאלה 2 (כללי) — (21 נקודות):

סעיף א' — (6 נקודות):

מנו (ותארו במשפט קצר) לפחות שלושה יתרונות חשובים שדנו בהם כאשר הוספנו שמות מזהים לשפה שלנו. מנו (ותארו במשפט קצר) לפחות שלושה יתרונות חשובים שדנו בהם כאשר הוספנו פונקציות לשפה שלנו.

סעיף ב' – תכנות בסיסי – (15 נקודות):

עליכם לכתוב מספר פרוצדורות בסיסיות בשפה pl. **השתמשו בקריאות זנב (ברקורסיה) בלבד. ניתן להגדיר פרוצדורות עזר. נתון הקוד החלקי הבא. השלימו אותו על-פי ההוראות מטה** (היכן שכתוב – «fill-in»–):

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

1. הפרוצדורה הבאה מממשת הזזה מחזורית שמאלה על רשימה של ביטים. כלומר, כל ביט מוזז מיקום אחד שמאלה (השמאלי ביותר הופך ימני).

```
(: shift-left : Bit-List -> Bit-List)
    ;; Shifts left a list of bits (once)
  (define(shift-left bl)  <--fill in 1-->)
```

2. הפרוצדורה הבאה בודקת האם לפחות מחצית הביטים ברשימה של ביטים הם אחדות.

```
(: majority? : Bit-List -> Boolean)
    ;; Consumes a list of bits and checks whether the
    ;; number of 1's are at least as the number of 0's.
(define(majority? bl)  <--fill in 2-->)
```

3. הפרוצדורה הבאה בודקת האם רשימה אחת של ביטים גדולה (כערך בייצוג בינארי) מרשימה שנייה.

```
(: geq-bitlists? : Bit-List Bit-List -> Boolean)
   ;; Consumes two bit-lists and compares them. It returns true if the
   ;; first bit-list is larger or equal to the second.
(define (geq-bitlists? bl1 bl2) <--fill in 3-->)
```

אוניברסיטת
אריאל
בשומרון

שאלה 3 — (28 נקודות):

נתון הקוד הבא:

```
(run "{with {+ {fun {x} {fun {y} {* x y}}}}
        {with {x 4}
          {call {call + {+ 1 2}} 6}}}")
```

סעיף א' (15 נקודות):

תארו את הפעלות הפונקציה **eval** בתהליך ההערכה של הקוד מעלה במודל ה-**substitution-cache** (על-פי ה-interpreter התחתון מבין השלושה המצורפים מטה) - באופן הבא – לכל הפעלה מספר *i* תארו את AST_i – הפרמטר האקטואלי הראשון בהפעלה מספר *i* *(עץ התחביר האבסטרקטי)*, *את* $Cache_i$ – הפרמטר האקטואלי השני בהפעלה מספר *i* *(רשימת ההחלפות)* *ואת* RES_i – הערך המוחזר מהפעלה מספר *i*.

הסבירו בקצרה כל מעבר. ציינו מהי התוצאה הסופית.

דוגמת הרצה: עבור הקוד

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

היה עליכם לענות (בתוספת הסברים)

AST_1 = (With x (Num 1) (Add (Id x) (Num 2)))
$Cache_1$ = '()
RES_1 = (Num 3)
AST_2 = (Num 1)
$Cache_2$ = '()
RES_2 = (Num 1)
AST_3 = (Add (Id x) (Num 2))
$Cache_3$ = '((x (Num 1))
RES_3 = (Num 3)
AST_4 = (Id x)
$Cache_4$ = '(x (Num 1))
RES_4 = (Num 1)
AST_5 = (Num 2)
$Cache_5$ = '((x (Num 1))
RES_5 = (Num 2)

```
Final result: 3
```

סעיף ב' (5 נקודות):

מה היה קורה לו היינו מבצעים את ההערכה במודל הסביבות? מהי התשובה הרצויה? מדוע? (אין צורך לבצע הערכה). תשובה מלאה לסעיף זה לא תהיה ארוכה משלוש שורות (תשובה ארוכה מדי תקרא חלקית בלבד).

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

אילו מהמשפטים הבאים נכונים?

א. בשפה **FLANG** שכתבנו בקורס, ישנם שני טיפוסים בלבד. אחד מהם הוא טיפוס פונקציה. **FLANG** מתייחסת לפונקציות כ-first class.

ב. הפעולות +, -, *, /, הן מטיפוס פונקציה בשפה **FLANG** שכתבנו בקורס.

ג. כאשר שפה מחושבת בצורת **dynamic scoping** תתכן דריסה של אובייקט מטיפוס פונקציה בתוך פרוצדורה מוגדרת. בפרט, ייתכן שנכתוב פרוצדורה foo המפעילה את הפרוצדורה g -- אשר, בזמן הגדרת foo, מוגדרת להעלות ארגומנט יחיד בחזקת 4 – וכשנריץ את foo, נקבל הודעת שגיאה האומרת כי g מצפה לשלושה ארגומנטים.

ד. במימוש במודל הסביבות של השפה **FLANG** שכתבנו בקורס, מימשנו **lexical scoping**, לכן, אם בתחילת התכנית הגדרנו שפונקציה f מעלה מספר בריבוע, לא ייתכן מקרה שבו (בהמשך התכנית) נקרא ל-f על 3 ונקבל 81.

שאלה 4 — Interpreter עבור השפה ROL המורחבת — (40 נקודות):

לצורך פתרון שאלה זו נעזר בקוד ה- interpreter של FLANG במודל הסביבות, המופיע בסוף טופס המבחן (האמצעי מבין השלושה המופיעים שם). בהמשך לשאלה ראשונה, נרצה לממש Interpreter עבור השפה ROL (המורחבת) 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}}}") =error>
                                        "wrong number of bits in (0 1 1)")
  (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))
(test (run "{ reg-len = 3
                 {if {geq? {1 0 1} {1 1 1}} {0 0 1} {1 1 0}}}")
        => '(1 1 0))
(test (run "{ reg-len = 4
                 {if {maj? {0 0 1 1}} {shl {1 0 1 1}} {1 1 0 1}}}")
        => '(0 1 1 1))
(test (run "{ reg-len = 4
                 {if false {shl {1 0 1 1}} {1 1 0 1}}}")
        => '(1 1 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, הרחיבו 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-->]
    [Bool <--fill in 9-->]
    [Geq <--fill in 10-->]
    [Maj <--fill in 11-->]
    [If <--fill in 12-->])
```

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-->) ; מהי ההודעה המתאימה?
         <--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-->] ; מהי ההודעה המתאימה?
    ['true <--fill in 7-->]
    [<--fill in 8-->]
    [(symbol: name) <--fill in 9-->]
    [(cons 'with more)
     (match sexpr
       [(list 'with (list (symbol: name) named) body)
        <--fill in 10-->]
       [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
    [(list 'and lreg rreg  <--fill in 11-->]
    [<--fill in 12-->]
    [<--fill in 13-->]
    [(cons 'fun more)
     (match sexpr
       [(list 'fun (list (symbol: name)) body)
        <--fill in 14-->]
       [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(list 'call fun arg) (Call <--fill in 15-->]
    [(list 'if <--fill in 16-->) <--fill in 17-->]
    [<--fill in 18-->]
    [<--fill in 19-->]
    [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' – eval (17 נקודות):

השלימו את הקוד החסר להגדרת הטיפוסים הנדרשים (בהמשך נתונות 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])
```

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

```
 (define-type VAL
   [RegV Bit-List]
   [FunV <--fill in 1-->]
   [BoolV <--fill in 2-->)

 (: 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))]))
```

השתמשו בהגדרות הבאות, הנותנות ניסוח פורמלי לאופן הרצוי להערכת קוד בשפה המורחבת.

```
#| Formal specs for `eval':
      eval(Reg,env)     = Reg
      eval(bl)          = bl
      eval(true)        = true
      eval(false)       = false
      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
   eval({if E1 E2 E3},env)
            = eval(E3, env)     if eval(E1,env) = false
            = eval(E2, env)     otherwise
```

```
    eval({maj? E},env) = true if x1+x2+...+xk >= k/2, and false otherwise,
                                        where eval(E,env) = (x1 x2 ... xk)
    eval({geq? E1 E2},env) = true if x_i >= y_i,
            where eval(E1,env) = (x1 x2 ... xk)
                  and eval(E2,env) = (y1 y2 ... yk)
                  and i is the first index s.t. x_i and y_i are not equal
                     (or i =k if all are equal)
   eval({if Econd  Edo Eelse}, env)
                       = eval(Edo, env) if eval(Econd, env) =/= false,
                       = eval(Eelse, env),    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 בשפה 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 3-->
          null
          (cons    <--fill in 4-->
  (RegV    <--fill in 5-->)))
```

<u>הדרכה</u>: בהשלימכם את הקוד מטה, ודאו שאתם מקפידים על הטיפוס הנכון של אובייקט הנשלח כארגומנט לפרוצדורה אחרת או כערך מוחזר לחיוב הנוכחי. השתמשו בפרוצדורות שכתבתם בסעיף ב' של שאלה 2.

נתונות לכם הפרוצדורות הבאות:

```
;; Defining functions for dealing with arithmetic operations
;; on the above types
  (: bit-and : BIT BIT -> BIT) ;; Arithmetic and
  (define(bit-and a b)
    (if (and (= a 1) (= b 1)) 1 0)) ;; using logical and

  (: bit-or : BIT BIT -> BIT) ;; Aithmetic or
  (define(bit-or a b)
    (if (or (= a 1) (= b 1)) 1 0)) ;; Using logical or
```

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

```
(: eval : RegE ENV -> VAL)
  ;; evaluates RegE expressions by reducing them to bit-lists
  (define (eval expr env)
    (cases expr
      [(Reg n)   <--fill in 6-->]
      [(Bool b) <--fill in 7-->]
      [(And l r)  <--fill in 8-->]
      [(Or l r)   <--fill in 9-->]
      [(Shl reg) (<--fill in 10-->)]
      [(With bound-id named-expr bound-body)
       (eval bound-body
             (Extend bound-id (eval named-expr env) env))]
      [(Id name) <--fill in 11-->]
      [(Fun bound-id bound-body)
       <--fill in 12-->]
      [(Call fun-expr arg-expr)
       (let ([fval (eval fun-expr env)])
         (cases fval
           [<--fill in 13-->]
           [else (error 'eval "`call' expects a function, got: ~s"
                                fval)]))]
       [(If cond-term do-term else-term)
        (let ([condval <--fill in 14-->]
           <--fill in 15-->)]
      [(Maj reg) <--fill in 16-->]
      [(Geq  regl regr) <--fill in 17-->]))))
```

<u>הערה:</u> שימו לב כי בטיפול בביטוי לוגי, עליכם להסיר את המעטפת של הביטוי בכדי להעריך את אמיתותו.
כמו כן, בביטוי if תמיד יוערך רק אחד משני ביטויים אפשריים.

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

```
--<<<FLANG-Substitution-cache>>>-----------------------------------------
  ;; The Flang interpreter, using Substitution-cache
#lang pl

  (define-type FLANG
    [Num  Number]
    [Add  FLANG FLANG]
    [Sub  FLANG FLANG]
    [Mul  FLANG FLANG]
    [Div  FLANG FLANG]
    [Id   Symbol]
    [With Symbol FLANG FLANG]
    [Fun  Symbol FLANG]
    [Call FLANG FLANG])

  (: parse-sexpr : Sexpr -> FLANG)
  ;; to convert s-expressions into FLANGs
  (define (parse-sexpr sexpr)
    (match sexpr
      [(number: n)     (Num n)]
      [(symbol: name)  (Id name)]
      [(cons 'with more)
       (match sexpr
         [(list 'with (list (symbol: name) named) body)
          (With name (parse-sexpr named) (parse-sexpr body))]
         [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
      [(cons 'fun more)
       (match sexpr
         [(list 'fun (list (symbol: name)) body)
          (Fun name (parse-sexpr body))]
         [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
      [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list 'call fun arg) (Call (parse-sexpr fun) (parse-sexpr arg))]
      [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

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

  ;; a type for substitution caches:
  (define-type SubstCache = (Listof (List Symbol FLANG)))

  (: empty-subst : SubstCache)
  (define empty-subst null)
```

```
(: extend : Symbol FLANG SubstCache -> SubstCache)
(define (extend name val sc)
  (cons (list name val) sc))

(: lookup : Symbol SubstCache -> FLANG)
(define (lookup name sc)
  (let ([cell (assq name sc)])
    (if cell
      (second cell)
      (error 'lookup "no binding for ~s" name))))

(: arith-op : (Number Number -> Number) FLANG FLANG -> FLANG)
;; gets a Racket numeric binary operator, and uses it within a FLANG
;; `Num' wrapper
(define (arith-op op expr1 expr2)
  (: Num->number : FLANG -> Number)
  (define (Num->number e)
    (cases e
      [(Num n) n]
      [else (error 'arith-op "expects a number, got: ~s" e)]))
  (Num (op (Num->number expr1) (Num->number expr2))))
(: eval : FLANG SubstCache -> FLANG)
;; evaluates FLANG expressions by reducing them to expressions
(define (eval expr sc)
  (cases expr
    [(Num n) expr]
    [(Add l r) (arith-op + (eval l sc) (eval r sc))]
    [(Sub l r) (arith-op - (eval l sc) (eval r sc))]
    [(Mul l r) (arith-op * (eval l sc) (eval r sc))]
    [(Div l r) (arith-op / (eval l sc) (eval r sc))]
    [(With bound-id named-expr bound-body)
     (eval bound-body
           (extend bound-id (eval named-expr sc) sc))]
    [(Id name) (lookup name sc)]
    [(Fun bound-id bound-body) expr]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr sc)])
       (cases fval
         [(Fun bound-id bound-body)
          (eval bound-body
                (extend bound-id (eval arg-expr sc) sc))]
         [else (error 'eval "`call' expects a function, got: ~s"
                      fval)]))]))
(: run : String -> Number)
;; evaluate a FLANG program contained in a string
(define (run str)
  (let ([result (eval (parse str) empty-subst)])
    (cases result
      [(Num n) n]
      [else (error 'run
                   "evaluation returned a non-number: ~s" result)])))
```