אוניברסיטת
אריאל
בשומרון

פקולטה: מדעי הטבע

מחלקה: מדעי המחשב ומתמטיקה

שם הקורס: שפות תכנות

קוד הקורס: 2-7036010

תאריך בחינה: _14/06/2015_ 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.
- 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 — (17 נקודות):

הערה: שאלה זו קשורה לשאלה 4 בהמשך המבחן. בשאלה 4 נרצה להרחיב את השפה FLANG ולאפשר שימוש בביטויים ופעולות לוגיות (על ערכים בוליאניים). ביתר פירוט – נוסיף ביטויי **if** (על-פי תחביר מיוחד המודגם בביטויים מטה); אופרטורים בינאריים – **<**, **>**, **=**; אופרטורים אונארי – **not**; 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):

```
True
{not True}
{> 3 44}
{if {- 3 3} {then-do 4} {else-do 5}}
{with {x 8}
   {if {> x 0} {then-do {/ 2 x}} {else-do x}}}
{with {x 0}
   {if {> x 0} {then-do {/ 2 x}} {else-do x}}}
{if {> 2 1} {then-do True} {else-do {+ 2 2}}}
{with {c True} {if c {then-do {> 2 1}} {else-do 2}}}
{not False}
{not 2}
```

נדגיש כי then-do, else-do הינם חלקי תחביר קבועים בביטויי if חוקיים.

סעיף א' BNF (10 נקודות):

הרחיבו את הדקדוק של השפה FLANG בהתאם לדוגמאות מעלה (הוסיפו את הקוד הנדרש היכן שכתוב –«fill-in»–):

```
#|  The grammar:
    <FLANG> ::= <num>  ;; Rule 1
              | { + <FLANG> <FLANG> } ;; Rule 2
              | { - <FLANG> <FLANG> } ;; Rule 3
              | { * <FLANG> <FLANG> } ;; Rule 4
              | { / <FLANG> <FLANG> } ;; Rule 5
              | { with { <id> <FLANG> } <FLANG> } ;; Rule 6
              | <id> ;; Rule 7
              | { fun { <id> } <FLANG> } ;; Rule 8
              | { call <FLANG> <FLANG> } ;; Rule 9
              | –«fill-in 1»– ;; add rule for True  ;; Rule 10
              | –«fill-in 2»–                       ;; Rule 11
              | –«fill-in 3»– ;; add rule for =     ;; Rule 12
              | –«fill-in 4»–                       ;; Rule 13
              | –«fill-in 5»–                       ;; Rule 14
              | –«fill-in 6»–                       ;; Rule 15
              | –«fill-in 7»– ;; add rule 16 for (the above) if expressions
|#
```

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). אם תשובתכם שלילית, הסבירו מדוע לא ניתן לגזור את המילה.

1. #f
2. {+ 1 22}
3. {< 2 True}
4. {fun {x} {< x x}}
5. {with {foo {fun {x} {if {< x 2} {then-do x} {else-do {/ x 2}}}}}
   foo}
6. {call x y}

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)

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

א. שימוש ב - dynamic scoping מגדיל את הצורך של המשתמש לעמוד בהגדרות הממשק שקבע מייצר התוכנה.

ב. שימוש ב - dynamic scoping עשוי להיות מועיל במקרה הבא: תוכנה לניהול בנק שמקבלת כפרמטר סכום הדולרים שהוחלפו בבנק. הרעיון הוא שפרמטר זה משתנה בכל יום ולכן עדכון מקומי שלו, ישנה את תפקוד התוכנה כולה.

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

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

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

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

אילו מהמשפטים הבאים נכונים לגבי תהליך ה-Parsing?

א. לעיתים תהליך ה-Parsing צריך להצליח גם כאשר הקוד אינו תקין.
ב. במימוש שלנו, בסוף תהליך ה-Parsing קיבלנו FLANG שהוא בהכרח או ואריאנט Num או ואריאנט Fun.
ג. אם בכל ביטוי בשפה FLANG היינו מחליפים סוגר מסולסל בסוגר מעוגל (באותו כיוון) ומחליפים with ב-let ו-call במילה ריקה, ניתן להשתמש באותו עץ תחביר אבסטרקטי לכל מילה – לפני ואחרי השינוי.
ד. במימוש שלנו, תהליך ה-Parsing במודל הסביבות – זהה לתהליך ה-Parsing במודל ה-substitution cache.

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

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

א. בשפות כאלה יכולת הביטוי וההפשטה של המתכנת גבוהות ביותר ולכן ביכולתו לייצר קוד יעיל יותר (מבחינת זמן ריצה סופי של התוכנית) מאשר בשפות תחתיות.
ב. שפות אימפרטיביות (פקודתיות – השמות דגש על רצפים של פקודות) – בדרך כלל אינן יכולות להתייחס לפונקציות כ-first class, שכן, יש בהן נטייה גדולה לייצר side effects (תוצרי לוואי) ועל כן, התייחסות כזאת לפונקציות גוררת חוסר יעילות המבטלת את היתרונות של התייחסות לפונקציות כ-first class.
ג. בשפה כזו, יש תמיד static typing (הגדרת טיפוסים בזמן כתיבת התוכנית ולא בזמן ריצה) כי כל פונקציה צריכה לדעת מראש את טיפוסי הארגומנטים הפורמליים שלה.
ד. תמיכה בפונקציות כ- first class מהווה אתגר קשה ביותר עבור מתכנני השפה, שכן הטיפול בזיכרון המחסנית הופך מסורבל מאד. בפרט, בשונה מ-RACKET, השפה שאנחנו תכננו במהלך הקורס, אינה תומכת בהתייחסות לפונקציות כ-first class.

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 הסביבות, הטיפול של הפונקציה eval בבנאי ld שונה מהטיפול בבנאי ld במודל ה-substitution cache כי במודל הסביבות פונקציה היא אובייקט סגור ולכן כל מופע חופשי הוא שגיאה.
ב. השוני המרכזי בין מימושי האינטרפרטר שכתבנו עבור FLANG במודל ה-substitution cache ובמודל הסביבות קשור באופן הדוק להבדל בין lexical scoping ו-dynamic scoping.
ג. במודל הסביבות פונקציה היא אובייקט סגור ולכן מוגדרת בסביבה שבה הוגדרה. לעומת זאת, במודל ה-substitution cache פונקציה היא אובייקט פתוח ולכן מוגדרת בסביבה שבה היא נקראת.
ד. היינו יכולים לכתוב את הפונקציה eval כך שלא יהיה צורך בפונקציה parse. ההחלטה להפריד לשני חלקים שונים נבעה משיקולים של "תכנות נכון".
ה. 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.

אוניברסיטת
אריאל
בשומרון

שאלה 3 — (41 נקודות):

נתון הקוד הבא:

```
(run "{with {z 2}
        {with {f {fun {y} {+ k y}}}
          {with {k 4}
            {call f z}}}}")
```

סעיף א' (10 נקודות):

תארו את הפעלות הפונקציה eval בתהליך ההערכה 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)

AST_1 = (With x (Num 1) (Add (Id x) (Num 2)))
RES_1 = (Num 3)
AST_2 = (Num 1)
RES_2 = (Num 1)
AST_3 = (Add (Num 1) (Num 2))
RES_3 = (Num 3)
AST_4 = (Num 1)
RES_4 = (Num 1)
AST_5 = (Num 2)
RES_5 = (Num 2)

Final result: 3

סעיף ב' (12 נקודות):

תארו את הפעלות הפונקציה eval בתהליך ההערכה של הקוד מעלה במודל ה- environment (על-פי ה- interpreter התחתון מבין השניים המצורפים מטה) - באופן הבא – לכל הפעלה מספר *i* תארו את AST_i – הפרמטר האקטואלי הראשון בהפעלה מספר *i* *(עץ התחביר האבסטרקטי)*, *את* ENV_i – הפרמטר האקטואלי השני בהפעלה מספר *i* *(הסביבה)* *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)))
ENV_1 = (EmptyEnv)
RES_1 = (NumV 3)
AST_2 = (Num 1)
ENV_2 = (EmptyEnv)
RES_2 = (NumV 1)
AST_3 = (Add (Id x) (Num 2))
ENV_3 = (Extend x (NumV 1) (EmptyEnv))
RES_3 = (NumV 3)
AST_4 = (Id x)
ENV_4 = (Extend x (NumV 1) (EmptyEnv))
RES_4 = (NumV 1)
AST_5 = (Num 2)
ENV_5 = (Extend x (NumV 1) (EmptyEnv))
RES_5 = (NumV 2)

```
Final result: 3
```

סעיף ג' (4 נקודות):

ההרצות שביצעתם בסעיפים קודמים מובילות לשתי תוצאות שונות. הסבירו את מהות ההבדל והסיבות לו. כתבו תשובה מלאה המדגישה מהי התוצאה הרצויה. (לא יותר משלוש שורות הסבר).

סעיף ד' – Eliminating free instances (15 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 ה - substitution) – השלימו את הקוד במקומות החסרים. הניחו שהקוד נכתב כתוספת לאינטרפרטר הנ"ל ולכן הוא מכיר את כל הפונקציות והטיפוסים שבו (ובפרט את הפונקציה subst).

```
(: containsFreeInstance? : FLANG -> Boolean)
;; Scans FLANG expressions for free instances of identifiers
(define (containsFreeInstance? expr)
    (cases expr
      [(Num n) —«fill-in 1»— ]
      [(Add l r) (or —«fill-in 2»—)]
      [—«fill-in 3»— ]
      [—«fill-in 4»—]
      [—«fill-in 5»—]
      [(With bound-id named-expr bound-body
           (—«fill-in 6»—)]
      [(Id name) —«fill-in 7»—]
      [(Fun bound-id bound-body) —«fill-in 8»—]
      [(Call fun-expr arg-expr) —«fill-in 9»—]))
```

שימו לב: אם נגדיר את פונקציית המעטפת הבאה –

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

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)
```

אוניברסיטת
אריאל
בשומרון

שאלה 4 — הרחבת השפה — (35 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 שימוש בביטויים ופעולות לוגיות (על ערכים בוליאניים). ביתר פירוט – נוסיף ביטויי **if** (על-פי תחביר מיוחד המודגם בטסטים למטה); אופרטורים בינאריים – **<**, **>**, **=**; אופרטורים אונארי – **not**; וערכי **True** ו – **False** (כאן, לא נרשה #t, #f).

להלן דוגמאות לטסטים שאמורים לעבוד (בפרט, כל טסט שעבד לפני הרחבת השפה, ימשיך לפעול גם לאחר ההרחבה):

```
;; tests
(test (run "True") => true)
(test (run "{not True}") => false)
(test (run "{> 3 44}") => false)
(test (run "{if {- 3 3} {then-do 4} {else-do 5}}") => 4)
(test (run "{with {x 8}
              {if {> x 0} {then-do {/ 2 x}} {else-do x}}}") => 1/4)
(test (run "{with {x 0}
             {if {> x 0} {then-do {/ 2 x}} {else-do x}}}") => 0)
(test (run "{if {> 2 1} {then-do True} {else-do {+ 2 2}}}") => true)
(test (run "{with {c True} {if c {then-do {> 2 1}} {else-do 2}}}")
    => true)
(test (run "{with {foo {fun {x}
                         {if {< x 2} {then-do x} {else-do {/ x 2}}}}}
                foo}") => (Fun 'x (If (Smaller (Id 'x) (Num 2)) (Id 'x)
(Div (Id 'x) (Num 2)))))
```

סעיף א' הרחבת הטיפוס FLANG (5 נקודות):

בהמשך לשאלה 1 ולטסטים מעלה, הרחיבו את הטיפוס בהתאם. הוסיפו את הקוד הנדרש (היכן שכתוב –«fill-in»–) ל –

```
(define-type FLANG
    [Num  Number]

... ראו קוד ה- interpreter מטה                    ...

   [Call FLANG FLANG]
    [Bool  <--fill in 1 -->]
    [Bigger <--fill in 2 -->]
    [Smaller <--fill in 3 -->]
    [Equal <--fill in 4 -->]
    [Not <--fill in 5 -->]
    [If <--fill in 6 -->])
```

סעיף ב' parse (5 נקודות):

הרחיבו את parse-sexpr בהתאם. הוסיפו את הקוד הנדרש (היכן שכתוב –«fill-in»–) ל –

```
(: parse-sexpr : Sexpr -> FLANG)
```

```
  ;; to convert s-expressions into FLANGs
  (define (parse-sexpr sexpr)
    (match sexpr
      [(number: n)     (Num n)]
      ['True (Bool true)]
      ['False <--fill in 1-->)]
      [(symbol: name) (Id name)]

... מטה interpreter -ראו קוד ה              ...

      [(list 'call fun arg) (Call (parse-sexpr fun) (parse-sexpr arg))]
      [(list '= lhs rhs) (Equal  <--fill in 2 -->)]
      [(list '> lhs rhs) <--fill in 3 -->]
      [<--fill in 4 -->]
      [(list 'not exp) <--fill in 5 -->]
      [(list 'if  <--fill in 6 -->]
      [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

<u>סעיף ג' subst (5 נקודות):</u>

נתונות ההגדרות הפורמליות לביצוע החלפות בשפה.

```
Substitution 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}

      B[v/x]                  = B       ;; B is Boolean
      {= 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]}
      { not E}[v/x]           = {not E[v/x]}
      {if Econd {then-do Edo} {else-do Eelse}}[v/x]
                            = {if Econd[v/x] {then-do Edo[v/x]} {else-do
Eelse[v/x]}}
```

הוסיפו את הקוד הנדרש (היכן שכתוב –«fill-in»–) 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
... מטה interpreter -ראו קוד ה                ...

      [(Bool b) <--fill in 1 -->]
      [(Equal l r) <--fill in 2 -->]
      [<--fill in 3 -->]
      [<--fill in 4 -->]
      [<--fill in 5 -->]
      [<--fill in 6 -->]))
```

<u>סעיף ד' – פונקציות עזר (5 נקודות):</u>

נתונות הפונקציות הבאות:

```
;; The following function is used in multiple places below,
   ;; hence, it is now a top-level definition
   (: Num->number : FLANG -> Number)
   ;; gets a FLANG -- presumably a Num variant -- and returns the
   ;; unwrapped number
    (define (Num->number e)
      (cases e
        [(Num n) n]
        [else (error 'Num->number "expected a number, got: ~s" e)]))

  (: 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 (op (Num->number expr1) (Num->number expr2))))
```

ממשו בעזרתן את שתי פונקציות העזר הבאות. הוסיפו את הקוד הנדרש (היכן שכתוב –«fill-in»–) ל –

הראשונה – מפעילה פונקציה בוליאנית דו-מקומית (על שני מספרים) ומחזירה FLANG.

```
  (: logic-op : (Number Number -> Boolean) FLANG FLANG -> FLANG)
  ;; gets a Racket Boolean binary operator (on numbers), and applies it
  ;; to two `Num' wrapped FLANGs
  (define (logic-op op expr1 expr2)
    <--fill in 1 -->)
```

השנייה – מקבלת FLANG ומחזירה את הערך הבוליאני המתאים לו (על פי כללים דומים לאלו ש-ראקט משתמשת בהם).

```
  (: flang->bool : FLANG -> Boolean)
  ;; gets a Flang E (of any kind) and returns a its appropiate
  ;; Boolean value -- which is true if and only if E does not
  ;; represent false
  ;; Remark: the `flang->bool` function will also be top-level
  ;; since it's used in more than one place.
  (define (flang->bool e)
    (cases e
      [<--fill in 2 -->]
      [else <--fill in 3 -->]))
```

סעיף ה' (10 נקודות):

נתונות ההגדרות הפורמליות לסמנטיקה של השפה.

```
   eval:  Evaluation rules:
      eval(N)              = N    ;; N is an expression for a numeric value
      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

      eval(B)                 = B      ;; B is an expression for a Boolean
value
      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({not E})           = not(eval(E))         /same for E
      eval({if Econd {then-do Edo} {else-do Eelse}})
                              = eval(Edo) if eval(Econd) =/= false,
                                              eval(Eelse),    otherwise.
```

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

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

      [(Bool b) <--fill in 1 -->]
      [<--fill in 2 -->]
      [<--fill in 3 -->]
      [<--fill in 4 -->]
      [(If l m r)
       (let ([<--fill in 5 -->])
         (<--fill in 6 -->))]
      [(Not exp) [<--fill in 7 -->]]))
```

הערה: 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.

סעיף ו' (5 נקודות):

לבסוף, נרצה לאפשר לאינטרפרטר המורחב להחזיר ערך מכל אחד משלושת הטיפוסים הקיימים עתה בשפה. נגדיר מחדש את הפונקציה run.

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

```
(: run : String -> (U Number Boolean FLANG))
  ;; evaluate a FLANG program contained in a string
  (define (run str)
    (let ([result (eval (parse str))])
      (cases result
        [<--fill in 1 -->]
        [<--fill in 2 -->]
        [<--fill in 3 -->])))



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