

פקולטה: מדעי הטבע

מחלקה: מדעי המחשב

שם הקורס: שפות תכנות

קוד הקורס: 2-7036010

תאריך בחינה: 09/02/2023 סמ' א' מועד א'

משך הבחינה: שעתיים וחצי

שם המרצה: סעיד עסלי

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 100 נקודות במבחן.
- לנוחיותכם מצורפים שני קטעי קוד עבור ה- interpreter של FLANG בסוף טופס המבחן. הראשון – במודל הסביבות והשני – במודל ה-Substitution-cache.

## שאלה 1 — BNF — (20 נקודות):

### סעיף א' (6 נקודות):

כתבו דקדוק עבור שפת המחרוזות הסופיות (באורך גדול מ-0) של אפסים ואחדות. להלן דוגמאות למילים חוקיות בשפה. הראו גזירה באורך ארבעה תווים.

```
1 0 0 0
1 0 1 0 1 0 1 0
1 1
```

פתרון מלא מקובל:

$\langle \text{BIN} \rangle := \langle \text{digit} \rangle$  (1)  
 $|\langle \text{BIN} \rangle \langle \text{BIN} \rangle$  (2)  
 Where  $\langle \text{digit} \rangle$  can be 0/1

נגזור את 1001:

$\langle \text{BIN} \rangle = 2 \langle \text{BIN} \rangle \langle \text{BIN} \rangle = 2 \langle \text{BIN} \rangle (\langle \text{BIN} \rangle \langle \text{BIN} \rangle) = 2 \langle \text{BIN} \rangle \langle \text{BIN} \rangle (\langle \text{BIN} \rangle \langle \text{BIN} \rangle)$   
 $= 1(4 \text{ times}) \langle \text{BIN} \rangle = 1001$

הערה: הסוגריים אומרים שפיתחתי את הביטוי מצד ימין.  
 סעיף ב' (7 נקודות):

הסבירו את מושג ה-ambiguity ביחס לדקדוק שהגדרתם. בפרט הראו שני דקדוקים לאותה שפה (של סעיף א). אחד שמתקיימת בו התכונה ואחר שאינה מתקיימת בו. הסבירו את תשובתכם.

דו משמעות: שפה שסובלת מדו משמעות הינה שפה שקיימת בה מילה שיש לה לפחות 2 עצי גזירה שונים.

לדוגמה, בסעיף א', המילה 1001 ניתנת גם כן לפיתוח מצד שמאל.

ע"מ לפתור בעיה זו נגדיר:

$\langle \text{BIN} \rangle := \langle \text{digit} \rangle$  (1)  
 $|\langle \text{digit} \rangle \langle \text{BIN} \rangle$  (2)  
 Where  $\langle \text{digit} \rangle$  can be 0/1

### סעיף ג' (7 נקודות):

הסבירו את מושג ה-compositionality ביחס לדקדוק שהגדרתם. בפרט, הראו בחירה של סמנטיקה ושני דקדוקים לאותה שפה (של סעיף א). אחד שמתקיימת בו התכונה ואחר שאינה מתקיימת בו. הסבירו את תשובתכם.

בומל והועבר לסעיפים א' ו-ב' (כך שכל סעיף עכשיו שוקל 10 נק') – בכל מקרה, הפתרון וההסבר די דומה למה שעשינו בכיתה! -כדאי לכם לנסות לפתור.

- הערות: מי שהראה בסעיף א' דקדוק שלא סובל מדו משמעות ובסעיף ב' הראה דקדוק שכן סובל מדו משמעות חושב לו ציון מלא.

## שאלה 2 — With vs. Call — (20 נקודות):

בכיתה דיברנו על כך שכל ביטוי FLANG יכול להיות מוחלף בביטוי FLANG שקול, אשר אינו מכיל הפעלות של בנאי With (מחליף אותן בעזרת בנאים Fun-ו Call).

סעיף א' – מקרה פרטי – (10 נקודות): נתון ה- FLANG הבא

```
(With t
  (Add (Num 11) (Num 25))
  (With u
    (Mul (Id t) (Id t))
    (With w
      (Num 2022)
      (Add (Id w) (Id u))))))
```

מצאו עבורו ביטוי שקול שאינו מכיל הפעלות של בנאי With (כמוסבר למעלה).

**פתרון:**

```
(Call (Fun 't
  (Call (Fun 'u
    (Call (Fun 'w
      (Add (Id 'w) (Id 'u)))
      (Num 2022)))
    (Mul (Id 't)
      (Id 't))))
  (Add (Num 11) (Num 25))))
```

סעיף ב' בעמוד הבא..

סעיף ב' – פתרון כללי – (10 נקודות):

כתבו פונקציה `SynSugarWith`, אשר מקבלת כקלט עץ דקדוק אבסטרקטי FLANG עוברת עליו ומחזירה ביטוי FLANG שקול שאינו מכיל הפעלות של `With` (כמוסבר למעלה).

פתרון:

---

```
(: SynSugarWith : FLANG -> FLANG)

(define (SynSugarWith T)

  (cases T

    [(Num n) (Num n)]

    [(Add l r) (Add (SynSugarWith l) (SynSugarWith r))]

    [(Sub l r) (Sub (SynSugarWith l) (SynSugarWith r))]

    [(Mul l r) (Mul (SynSugarWith l) (SynSugarWith r))]

    [(Div l r) (Div (SynSugarWith l) (SynSugarWith r))]

    [(With id named body) (Call (Fun id (SynSugarWith body)) (SynSugarWith named))]

    [(Id name) (Id name)]

    [(Fun bound-id bound-body)

      (Fun bound-id (SynSugarWith bound-body))]

    [(Call fun-expr arg-expr)

      (Call (SynSugarWith fun-expr) (SynSugarWith arg-expr))]))
```

---

שאלה 3 — (30 נקודות): נתון הקוד הבא:

```
(run "{with {foo {fun {x} {* x x}}}  
      {call {with {foo {fun {y} {- y 1}}}  
              {fun {x} {call foo x}}}  
            4}}")
```

סעיף א' (15 נקודות):

תארו את הפעולות הפונקציה **eval** בתהליך ההערכה של הקוד מעלה במודל ה-**substitution-cache** (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  $Cache_i$  – הפרמטר האקטואלי השני בהפעלה מספר  $i$  (רשימת ההחלפות) ואת  $RES_i$  – הערך המוחזר מהפעלה מספר  $i$ . הסבירו בקצרה כל מעבר. ציינו מהי התוצאה הסופית.

תוצאה: 16

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).

תוצאה: 3

סעיף ג' (10 נקודות):

האינטרפרטר שכתבנו במודל ה-**substitution-cache** מממש **dynamic-scoping** ואילו האינטרפרטר שכתבנו במודל הסביבות מממש **static-scoping**. בשאלה זאת, עליכם לשנות את הקוד של האינטרפרטר שכתבנו במודל הסביבות (העליון מבין השניים המופיעים מטה) – על מנת להמירו למימוש של **dynamic-scoping**. עליכם לשנות מעט ככל שתוכלו מהקוד.

העתיקו למחברת רק את שורות הקוד שברצונכם לשנות וציינו בבירור מהו השינוי הנדרש. ככל שתשובתכם תכלול פחות שינויים לקוד המקורי, היא תזכה אתכם ביותר ניקוד. הסבירו בבירור מדוע השינויים שביצעתם נדרשים ומדוע הם מספיקים. תשובה ללא הסבר, לא תזכה אתכם בניקוד.

משנים את **f-env** ל- **env** בשורה:

(Extend bound-id (eval arg-expr env) f-env)

רמז: תשובה טובה תהיה קצרה מאד.

הערה: מי שנתן תשובות הפוכות עבור א' ו-ב' ניתן לו ציון חלקי עליהם (10 עבור א' ו-3 עבור ב') – כל מי שקיבל פחות אז זה תלוי בפתרון שלו.

#### שאלה 4 – הרחבת השפה FLANG מודל הסביבות – (30 נקודות):

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

נרצה להרחיב את השפה **FLANG** ולאפשר חישוב לולאות while פשוט עם פעולת חיבור. כלומר, נאפשר חזרה על קטע קוד, עם מונה (counter), כך שבתוך קטע הקוד מותר להשתמש בערך המונה (קטע הקוד עשוי להיות כל ביטוי על פי התחביר בשפה). הערך המוחזר מחישוב כל הלולאה יהיה סכום כל הערכים המוחזרים מכל החישובים בכל האיטרציות של הלולאה.

להלן דוגמה לביטוי אפשרי:

```
"{ while { i } = 1 <= 12 do {+ 1 i} }"
```

כאן,  $i$  הוא שם המונה, 1 הוא ערך התחלתי למונה ו-12 הוא גבול עליון,  $\{+ 1 i\}$  הוא גוף הלולאה. לשם פשטות, בהמשך ניתן תמיד להניח שערך הגבול העליון למונה גדול ממש מן הערך ההתחלתי עבור המונה וכי שניהם מספרים שלמים.

להלן דוגמאות למסמכים שאמורים לעבוד:

```
(test (run "{ while { i } = 1 <= 12 do 1 }")
      => 12)
(test (run "{ while { i } = {+ 1 0} <= 5 do i }")
      => 15)
(test (run "{with {x { while { i } = 1 <= 5 do i }}
            {- x 10}}")
      => 5)

(test (run "{ while { i } = 1 <= 4 do i }")
      => 24)

(test (run
      "{with {factorial {fun {n} { while { i } = 1 <= n do i }}}
        {call factorial 4}}")
      => 24)

(test (run "{ while { i } = {with {x {- 5 3}} x} <= {- 7 4} do
            {with {sqr {fun {x} {* x x}}}
              {call sqr i}}}")
      => 13)

(test (run
      "{ + 5 { while { i } = {with {x {- 5 3}} x} <= {- 7 4} do
            {with {sqr {fun {x} {* x x}}}
              {call sqr i}}}}")
```

=> 18)

```
(test (run "{ while i = 1 <= 12 do 1 }")
      =error> "parse-sexpr: bad `while' syntax in")
```

הערה: השתמשו בדוגמאות אלו בכדי להבין את דרישות התחביר, את אופן הערכת הקוד וכן את הודעות השגיאה שיש להדפיס במקרים המתאימים.

סעיף א' (הרחבת הדקדוק) (6 נקודות):

כתבו מהו כלל הדקדוק שיש להוסיף לדקדוק הקיים.

```
-«fill-in 1»-: { while { <id> } = <FLANG> <= <FLANG> do <FLANG> }
```

סעיף ב' (הרחבת הטיפוס FLANG) (3 נקודות):

הוסיפו את הקוד הנדרש (יש להוסיף בנאי יחיד) ל –

```
(define-type FLANG
...
[WHILE -«fill-in 2»-]:[While Symbol FLANG FLANG FLANG]
...)
```

סעיף ג' (parsing) (10 נקודות):

השתמשו בדוגמאות המסמכים מעלה כדי להבין אילו הודעות שגיאה רצויות. הוסיפו את הקוד הנדרש בתוך הסוגריים המרובעים – 7 השלמות סה"כ לסעיף זה) ל –

```
(: parse-sexpr : Sexpr -> FLANG)
;; to convert s-expressions into FLANGs
(define (parse-sexpr sexpr)
  (match sexpr
    ...
    [(cons -«fill-in 3»-) : 'while more (2 pts)
      (match -«fill-in 4»-: sexpr (1 pts)
        [(list 'while -«fill-in 5»-) : (list (symbol: name)) '= from '<= to 'do do (3 pts)
          -«fill-in 6»-]:
          (While name (parse-sexpr from) (parse-sexpr to) (parse-sexpr do)) (3 pts)
          [else -«fill-in 7»-]]):
      (error 'parse-sexpr "bad `while' syntax in ~s" sexpr) (1 pts)
    ...
    [else (error 'parse-sexpr "bad syntax in ~s"
      sexpr) ]))
```

### סעיף ד' (evaluation) (11 נקודות):

בסעיף זה נשלים את הקוד שיאפשר הערכה של ביטויי לולאה כפי שהגדרנו בסעיפים הקודמים. ראשית, נהפוך את הפרוצדורה NumV->number זמינה ומוכרת (עד כה הייתה פנימית ל - arith-op).

```
(: NumV->number : VAL -> Number)
(define (NumV->number v)
  (cases v
    [(NumV n) n]
    [else (error ' NumV->number "expected a number, got: ~s" v)]))
(: 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 (op (NumV->number val1) (NumV->number val2))))
```

עתה נכתוב פונקציה שתקרא ע"י eval ותבצע את האיטרציות של הלולאה והחלת פעולת החיבור על כל תוצאות החישובים האיטרטיביים. השלימו את הקוד החסר:

```
(: eval-while-loop : Symbol FLANG FLANG FLANG ENV -> VAL)
(define (eval-while-loop cnt-name from-exp to-exp body env)
  (: loop-eval : Number Number -> VAL)
  (define (loop-eval from to)
    (if -«fill-in 8»- : (> from (- to 1)) (1 pts)
        (eval -«fill-in 9»-):
        body (Extend cnt-name (NumV from) env) (2 pts)
        (arith-op -«fill-in 10»-):
    + (eval body (Extend cnt-name (NumV from) env)) (loop-eval (+ from 1) to) (3 pts)
    (loop-eval -«fill-in 11»-)):
  (NumV->number (eval from-exp env)) (NumV->number (eval to-exp env)) (2 pts))
```

הדרכה: כדי להשלים את #9 – חישבו מהו תנאי העצירה המתאים במקרה שאמורה להיות איטרציה אחת.

כדי להשלים את #10 – זכרו שבתוך גוף הלולאה ייתכנו מופעים חופשיים של שם המונה.

הערה: לשם פשטות, ניתן להניח שערך הגבול העליון למונה גדול ממש מן הערך ההתחלתי עבור המונה וכי שניהם מספרים שלמים.



לבסוף, השלימו את שורת הקוד המתאימה בפונקציה eval:

```
(: eval : FLANG ENV -> VAL)
;; evaluates FLANG expressions by reducing them to values
(define (eval expr env)
  (cases expr
    ...
    [(While -«fill-in 12»-) : cnt-name from to body (1 pts)
     (-«fill-in 13»-)])) : (eval-while-loop cnt-name from to body env) (2 pts)
```

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