פקולטה: מדעי הטבע

בית ספר: מדעי המחשב

שם הקורס: שפות תכנות

קוד הקורס: 7036010

תאריך בחינה: _09/04/2024_ סמ' _א'___מועד __א'_

משך הבחינה: שעתיים וחצי

שם המרצה: ד"ר סעיד עסלי

שם מתרגל: מר' בנימין סאלדמן

חומר עזר: אסור

שימוש במחשבון: לא

הוראות כלליות:

- **קראו היטב את הוראות המבחן.**
- 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 (אין זה תקף לחלקי סעיף).
- אפשר להסתמך על סעיפים קודמים גם אם לא עניתם עליהם.
- יש לענות על כל השאלות.
- בכל מקום הדורש תכנות, מותר להגדיר פרוצדורות עזר (אלא אם נאמר אחרת במפורש).
- לנוחיותכם מצורף קטע קוד עבור ה- interpreter של FLANG בסוף טופס המבחן.

בהצלחה!

שאלה 1 — מופעים חופשיים של שמות מזהים — (25 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. למעשה, שגיאה כזאת יכולה להתגלות עוד לפני הפעלת ה-eval.

סעיף א' – free-instance – (5 נקודות):

יהא E ביטוי (תכנית בשפה FLANG). הסבירו מהו מופע חופשי בביטוי E וכיצד עשוי מופע של x להיות לא-חופשי (קשור) בביטוי E כולו, אך חופשי בתת-ביטוי של E. תנו תשובה קצרה (לא יותר ארבע שורות). השתמשו במילים scope, binding-instance.

סעיף ב' – count-free-instances – (5 נקודות):

נתון הקוד הבא בשפה FLANG:

```
"{with {x {fun {y} {- x y}}}
    {+ {with {x {- x 3}} {+ x y}}
        y}}"
```

מהם המופעים החופשיים בקוד זה (רק כאלה שהינם חופשיים בביטוי כולו)? הסבירו לגבי כל אחד שקבעתם שהוא חופשי – מדוע הוא אכן כזה.

תשובה: 3

סעיף ג' – CFI – (10 נקודות):

בסעיף זה תכתבו פונקציה נומרית CFI (קיצור עבור CountFreeInstance) המקבלת ביטוי (תכנית) בצורת FLANG ומחזירה את המספר (הטבעי) של מופעים חופשיים של שמות מזהים, אשר מכיל הביטוי – כך שהפונקציה אינה מבצעת שום הערכה סמנטית של התכנית.

בסעיפים הבאים תכתבו קוד כתוספת לאינטרפרטר של FLANG במודל ההחלפות. הניחו, אם כן, כי כל הפונקציות והטיפוסים שבו מוגדרים לכם.

הקוד הבא מבוסס על הפונקציה eval (באינטרפרטר של FLANG במודל ההחלפות – הקוד עבור פונקציה זאת ועבור subst מופיע בתחתית טופס המבחן) – השלימו את הקוד במקומות החסרים.

```
(: CFI : FLANG -> Natural)
;; Scans FLANG expressions to count free instances of identifiers
(define (CFI expr)
    (cases expr
      [(Num n) —«fill-in 1»— ]0
      [(Add l r) —«fill-in 2»—](+ (CFI l) (CFI r))
      [(Sub l r) —«fill-in 3»— ] (+ (CFI l) (CFI r))
      [(Mul l r) —«fill-in 4»—](+ (CFI l) (CFI r))
      [(Div l r) —«fill-in 5»—](+ (CFI l) (CFI r))
      [(With bound-id named-expr bound-body)
         (+ —«fill-in 6»—)]
(+ (CFI named-expr)
```

```
(CFI (subst bound-body bound-id (Num 0))))
[(Id name) —«fill-in 7»—] 1
[(Fun bound-id bound-body) —«fill-in 8»—]
(CFI (subst bound-body bound-id (Num 0)))
[(Call fun-expr arg-expr) —«fill-in 9»—])) (+ (CFI fun-expr) (CFI arg-expr))
```

הדרכה:

1. תפקידכם למנות מופעים חופשיים ולא להעריך את הביטוי. כך, הטיפול בכל הבנאים יהיה שונה מהטיפול ב-`eval`. אל תפעילו את `eval` עצמה בקוד שלכם. מותר לכם להשתמש בפונקציות אחרות, כגון `subst`.

כשאתם מחליטים כיצד יש לטפל בבנאי מסוים, מומלץ לצייר את מבנהו ולשים לב לכל תת ביטוי שעשוי להכיל מופעים חופשיים. מעבר לכך, יתכן שצריך לבצע פעולה כלשהי על תת הביטוי לפני שבודקים אותו.

<u>בדקו את עצמכם</u>: אם נגדיר את פונקציית המעטפת הבאה –

```
(: count-free : String -> Natural)
(define (count-free str)
  (CFI (parse str)))
```

כל הטסטים הבאים צריכים לעבוד –

```
;tests
(test (count-free "{+ z z}") => 2)
(test (count-free "{call {fun {x} {/ x 0}} 4}") => 0)
(test (count-free "{call {fun {y} {+ x 0}} 4}") => 1)
(test (count-free "{fun {y} {- x z}}") => 2)
(test (count-free "{with {foo {fun {y} {- y r}}}
                    {call foo {* c foo}}}") => 2)
(test (count-free "{call foo {+ t r}}") => 3)
(test (count-free "{fun {x} {+ x {/ 5 0}}}") => 0)
```

<u>סעיף ב' – (5 נקודות)</u>: תארו מה יקרה בהרצת שניים מהקודים למעלה (אחד שתשובתו 0 והשני לא) – על פי הבניה שלכם עבור הפונקציה `CFI?` בסעיף א'. האם התוצאה המתקבלת הינה התוצאה הרצויה?

<u>שאלה 2 — שאלות כלליות — (20 נקודות):</u>

<u>סעיף א'</u>: ביטוי cond הוא syntactic sugar בשפה ראקט.

מהו syntactic sugar באופן כללי? הסבירו מה הסיבה להוסיפו לשפה. מה הוא אינו נותן? הדגימו דבריכם לגבי ביטויי cond. (כתבו שלוש שורות לכל היותר!)

<u>סעיף ב'</u>:

מהו דקדוק דו-משמעי? הסבירו מה הבעיה בדקדוק כזה ביחס לשפת תכנות. תנו דוגמה קצרה (אין צורך בדוגמה מלאה, ניתן להזכיר דוגמאות שנתנו בקורס).

(כתבו שלוש שורות לכל היותר, מעבר לדוגמה.)

שאלה 3 — שאלות כלליות (בחירה) — (20 נקודות):

לפניכם מספר שאלות פשוטות. עליכם לבחור את התשובות הנכונות לכל סעיף (ייתכנו מספר תשובות נכונות – סמנו את כולן).

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

נתון הקוד הבא ב-Racket?

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

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

סעיף ב' (7 נקודות): (סמנו את כל התשובות הנכונות)

אילו מהמשפטים הבאים נכונים לגבי תהליך ה-Parsing?

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

סעיף ג' (7 נקודות): (סמנו את כל התשובות הנכונות)

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

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

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

ג. ישנן שפות כנ"ל שאינן מאפשרות להגדיר פונקציה ללא מתן שם (בזמן הגדרתה), אולם מאוחר יותר ניתן לשלוח את הפונקציה כפרמטר לפונקציה כלשהי ואף להחזיר פונקציה כערך מוחזר של חישוב כלשהו.

ד. בשפות אלו פונקציה מקבלות את הפרמטר דרך רגיסטרים, והפונקציה עצמה הינה כתובת בזיכרון.

שאלה 4 — הרחבת השפה FLANG מודל ההצבה — (35 נקודות):

לצורך פתרון שאלה זו נעזר בקוד ה- interpreter של FLANG במודל ההצבה, המופיע בסוף טופס המבחן.

נרצה להרחיב את השפה FLANG ולאפשר חישוב לולאות for עם פעולת חיבור או עם פעולת כפל. 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 (אם הביטוי משתמש באופרטור +, ראו דוגמאות מטה) או מכפלת כל הערכים הללו (אם הביטוי משתמש באופרטור *).

להלן דוגמה לביטוי אפשרי:

```
"{ for + { i } = 1 to 12 do {+ 1 i} }"
```

כאן, i הוא שם המונה, 1 הוא ערך התחלתי למונה ו-12 הוא גבול עליון, {+ 1 i} הוא גוף הלולאה והפעולה היא פעולת החיבור (זאת על-פי השימוש באופרטור + שבהתחלה). לשם פשטות, בהמשך ניתן תמיד להניח שערך הגבול העליון למונה גדול ממש מן הערך ההתחלתי עבור המונה וכי שניהם מספרים שלמים.

להלן דוגמאות לטסטים שאמורים לעבוד:

```
(test (run "{ for + { i } = 1 to 12 do 1 }")
        => 12)
```

בדוגמה למעלה, הלולאה רצה מ-1 עד – 12, וכל פעם מוסיפה (בגלל פעולת הפלוס) 1 לתשובה – לכן, מחשבת את 1+1+1..1 (12 פעמים) = 12

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

(test (run "{ for * { i } = 1 to 4 do i }")
        => 24)

(test (run
    "{with {factorial {fun {n} { for * { i } = 1 to n do i }}}
         {call factorial 4}}")
```

```
      => 24)

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


(test (run
     "{ + 5 { for + { i } = {with {x {- 5 3}} x} to {- 7 4} do
                {with {sqr {fun {x} {* x x}}}
                   {call sqr i}}}}")
      => 18)

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

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

<u>סעיף א' (הרחבת הדקדוק) (4 נקודות):</u> כתבו מהם שני כללי הדקדוק שיש להוסיף לדקדוק הקיים.

—«fill-in 1»— { for + { <id> } = <FLANG> to <FLANG> do <FLANG> }\

—«fill-in 2»— { for * { <id> } = <FLANG> to <FLANG> do <FLANG> }

<u>סעיף ב' (הרחבת הטיפוס FLANG) (4 נקודות):</u> הוסיפו את הקוד הנדרש (יש להוסיף בנאי יחיד).

```
(define-type FLANG
…
[For —«fill-in 3»—]   [For (Number Number -> Number) Symbol FLANG FLANG FLANG])
…)
```

הדרכה: הארגומנט הראשון צריך להיות זה שיגדיר את הפעולה המתאימה (חיבור או כפל).

<u>סעיף ג' (parsing) (7 נקודות):</u>

השתמשו בדוגמאות הטסטים 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 4»—)   (cons 'for more)
       (match —«fill-in 5»— match sexpr
```

```
          [(list 'for —«fill-in 6»—)
(list 'for '+ (list (symbol: name)) '= from 'to to 'do do)

            —«fill-in 7»—]
(For + name (parse-sexpr from) (parse-sexpr to) (parse-sexpr do))
 [(list 'for —«fill-in 8»—)
(list 'for '* (list (symbol: name)) '= from 'to to 'do do)
            —«fill-in 9»—]
(For * name (parse-sexpr from) (parse-sexpr to) (parse-sexpr do))
          [else —«fill-in 10»—])]
(error 'parse-sexpr "bad `for' syntax in ~s" sexpr)])
       ….
       [else (error 'parse-sexpr "bad syntax in ~s"
          sexpr)]))
```

<u>סעיף ד' (substitution) (10 נקודות):</u>

בסעיף זה נשלים את הקוד שיאפשר הצבה של שם מזהה (סימבול) בביטוי אחר בתוך ביטוי מסוים כפי שהגדרנו בכיתה.

```
(: subst : FLANG Symbol FLANG -> FLANG)
(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))]
          [(With name named-expr body)
           (With name
                 (subst named-expr from to)
                 (if (eq? name from)
                     body
                     (subst body from to))
                 )
           ]
          [(Id name) (if (eq? from name) to expr)]
          [(Fun name body) (Fun name
                            (if (eq? name from)
                                body
                                (subst body from to)))]
          [(Call fun-expr arg-expr) (Call (subst fun-expr from to)
(subst arg-expr from to))]
          [(For —«fill-in 11»—) (For op cnt-name i_from i_to body)
```

```
—«fill-in 12»—
(For op cnt-name (subst i_from from to) (subst i_to from to)

                    (if (eq? cnt-name from) body

                    (subst body from to)))


]
```

סעיף ה' (evaluation) (10 נקודות):

בסעיף זה נשלים את הקוד שיאפשר הערכה של ביטויי לולאה כפי שהגדרנו בסעיפים הקודמים.

```
  (: Num->number : FLANG -> Number)
(define (Num->number arg)
  (cases arg
    [(Num n) n]
    [else (error 'Num->number "excpected a number, got: ~s" arg)]))



(: arith-op : (Number Number -> Number) FLANG FLANG -> FLANG)

(define (arith-op op arg1 arg2)
  (Num (op (Num->number arg1) (Num->number arg2))))
```

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

```
(: eval-for-loop : (Number Number -> Number) Symbol FLANG FLANG
FLANG -> FLANG)
 (define (eval-for-loop op cnt-name from-exp to-exp body)
   (: loop-eval :  Number Number -> FLANG)
   (define (loop-eval from to)
     (if —«fill-in 13»— (> from (- to 1))
         (eval —«fill-in 14»—) (eval (subst body cnt-name (Num from)))

         (arith-op —«fill-in 15»—)
(arith-op op

         (eval (subst body cnt-name (Num from)))

         (loop-eval (+ from 1) to))


   (loop-eval —«fill-in 16»—)) (loop-eval (Num->number (eval from-exp)) (Num-
>number (eval to-exp)))
```

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
     …
      [(For —«fill-in 17»—)  (For op cnt-name from to body)

       (—«fill-in 18»—)]))  (eval-for-loop op cnt-name from to body)
```

```
--<<<FLANG-SUBST>>>-----------------------------------------------------
  ;; The Flang interpreter, using substitution
#lang pl
#| BNF:
<FLANG> :=
     <num>
    |{+ <FLANG> <FLANG>}
    |{- <FLANG> <FLANG>}
    |{* <FLANG> <FLANG>}
    |{/ <FLANG> <FLANG>}
    |{with {<id> <FLANG>} <FLANG>}
    |<id>
    |{fun {<id>} <FLANG>}
    |{call <FLANG> <FLANG>}

-------

Formal specifications for subst (e[v/i]):

1. N[v/x] = N
2. {+ E1 E2}[v/x] = {+ E1[v/x] E2[v/x]}
3. {- E1 E2}[v/x] = {- E1[v/x] E2[v/x]}
4. {* E1 E2}[v/x] = {* E1[v/x] E2[v/x]}
5. {/ E1 E2}[v/x] = {/ E1[v/x] E2[v/x]}
6. y[v/x] = y
7. x[v/x] = v (this is the only place where we actually substitute)
8. {with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]}
9. {with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}
10. {call E1 E2}[v/x] = {call E1[v/x] E2[v/x]}
11. {fun {y} E1}[v/x] = {fun {y} E1[v/x]}
12. {fun {x} E1}[v/x] = {fun {x} E1}

-------
Formal specifications for eval:

1. eval(N) = N
2. eval({+ E1 E2}) = if eval(E1), eval(E2) return numbers:
                          eval(E1) + eval(E2)
                     else: error
3. eval({- E1 E2}) =if eval(E1), eval(E2) return numbers:
                           eval(E1) - eval(E2)
4. eval({* E1 E2}) = if eval(E1), eval(E2) return numbers:
```

```
                          eval(E1) * eval(E2)
5. eval({/ E1 E2}) =if eval(E1), eval(E2) return numbers:
                           if eval(E2) != 0:
                            eval(E1) / eval(E2)
                     else: Error / by 0
6. eval(id) = Error!
7. eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
8. eval({fun {x} E1})= "a cloud" [arg, body]
9. eval({call E1 E2}) = if eval(E1) -> {fun {x} Ef}
                        eval(Ef[eval(E2)/x])
                       else: Error - expected a function got: ...
-------

|#


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


(: parse-sexpr : Sexpr -> FLANG)

(define (parse-sexpr sxpr)
  (match sxpr
    [(number: n) (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
     (match sxpr
     [(list 'with (list (symbol: name) named-expr) body)
     (With name (parse-sexpr named-expr) (parse-sexpr body))]
     [else (error 'parse-sexpr "bad with syntax in ~s" sxpr)])]


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

    [(list 'call fun-expr arg-expr)
     (Call (parse-sexpr fun-expr) (parse-sexpr arg-expr))]

    [(list '+ l r) (Add (parse-sexpr l) (parse-sexpr r))]
    [(list '- l r) (Sub (parse-sexpr l) (parse-sexpr r))]
    [(list '* l r) (Mul (parse-sexpr l) (parse-sexpr r))]
    [(list '/ l r) (Div (parse-sexpr l) (parse-sexpr r))]
    [else (error 'parse-sexpr "bad syntax in ~s" sxpr)]
   ))
```

```
(: parse : String -> FLANG)
(define (parse code)
  (parse-sexpr (string->sexpr code)))

(: subst : FLANG Symbol FLANG -> FLANG)
(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))]



          [(With name named-expr body)
           (With name
                 (subst named-expr from to)
                 (if (eq? name from)
                     body
                     (subst body from to))
                 )
           ]
          [(Id name) (if (eq? from name) to expr)]
          [(Fun name body) (Fun name
                            (if (eq? name from)
                                body
                                (subst body from to)))]
          [(Call fun-expr arg-expr) (Call (subst fun-expr from to) (subst
arg-expr from to))]
          ))

(: Num->Number : FLANG -> Number)
(define (Num->Number arg)
  (cases arg
    [(Num n) n]
    [else (error 'Num->Number "excpected a number, got: ~s" arg)]))

(: arith-op : (Number Number -> Number) FLANG FLANG -> FLANG)
(define (arith-op op arg1 arg2)
  (Num (op (Num->Number arg1) (Num->Number arg2))))

(: eval : FLANG -> FLANG)
(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) (if (zero? (Num->Number (eval r)))
                   (error 'eval "Dont Devide by 0 again pls!")
                   (arith-op / (eval l) (eval r)))]

    [(With name named body) (eval (subst body name (eval named)))]
    [(Id name) (error 'eval "free identfier ~s" name)]
    [(Fun name body) expr]
```

```
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr)])
       (cases fval
         [(Fun name body)
          (eval (subst body
                       name
                       (eval arg-expr)))]
         [else (error 'eval "expected a function, got: ~s" fval)]))]
   ))

(: run : String -> Number)
(define (run code)
  (let ([res (eval (parse code))])
    (cases res
      [(Num n) n]
      [else (error 'run "evaluation returned a non-number: ~s" res)])))
```