עקרונות הקומפילציה – מטלות 2-3

מאיר גולדברג

הגשה: יום ג׳, 31 דצמבר, 2024, בשעה 12:00 **בצהרים**

1 הנחיות כלליות

- העבודה היא ביחידים או בזוגות. צירוף לקבוצות גדולות יותר אסור, עבודות בשלושה שותפים או יותר לא תבדקנה, וּכְאִלּוּ לֹא הֻגְּשׁוּ!
- אין להשתמש בקוד של תלמידים או קבוצות אחרות, לא מִשָּׁנָה זו, לא משנים קודמות, ולא בקוד אחר שנמצא באינטרנט. הדברים נוגעים בהעתקה של קוד, בשלמותו או בחלקו, עם או בלי נסיונות להסתיר את ההעתקה. מקרים שיתגלו יועברו לטיפולה של **ועדת משמעת**. אין זו הדרך בה אנחנו, סגל הקורס, מעוניינים להתייחס לתלמידים שלנו, אבל העתקות הן בעיה חמורה ונפוצה, ואנחנו ננהג בהתאם **לתקנון משמעת הסטודנטים**.
- אתם אחראים בלעדית לבדוק את העבודה שלכם בטרם ההגשה:
  - אנחנו נשחרר דוגמאות להרצת הקוד, שמטרתן להדגים את הממשק ולספק מספר קטן של דוגמאות לקלט ולפלט תקינים. **הדוגמאות הללו לא תמצנה את בדיקת הנכונות**, ועליכם להצמד לטכסט של המטלה על מנת למנע אותה נכון!
  - **פלט מיותר הוא פלט שגוי**. ודאו שהקוד שלכם לא מייצר פלט שלא נתבקשתם ליצר (שְׁאֵרִיּוֹת של קוד ל־*debug* ו־*trace*, וכו׳). ככלל, סגנון התכנות בקורס הוא פונקציונאלי, כלומר ללא *side effects*, ועל־כן אינכם אמורים להדפיס דבר. **החריג היחיד לכך הוא בפרוייקט הסופי**, לכשתתבקשו לכתוב לקבצים קוד באסמבלי. עד אז, מה שעשוי להראות כפלט הוא בפועל ההדפסה האוטומטית של הערך המוחזר מקריאה לפונקציה.
- אנא קראו את המסמך הזה בזהירות מתחילתו ועד סופו, וּוַדְּאוּ שאתם מבינים מה אתם מתבקשים לעשות בטרם תתחילו לתכנת!

2 שתי מטלות משולבות

המסמך הזה מתאר שתי מטלות:

- פרסר לשפת הקוד של סקים
- מנתח סמנטי לשפת סקים

על מנת לחסוך בזמן, שלבנו את שתי המטלות ואתם תגישו אותן יחדיו.
בקוד תמצאו הערות בסגנון:

```
(* add support for שם של מבנה שעליכם לתמוך בו *)
```

מחקו נא את ההערה (או החליפו אותה באחרת!) וכתבו את הקוד שתומך במבנה שנתבקשתם לתמוך בו. זו תהיה צורה תחבירית שנתמכת בליבת הקומפיילר, ושתהיה מוגדרת בטיפוסים `expr`, `'expr`.

- אנא התחילו לעבוד על המטלות הללו מוקדם, ואל תחכו לרגע האחרון!

- אל תתחילו לכתוב קוד מיד, אלא עברו קודם על כל המטלות, הבינו היטב למה אתם מתבקשים.
- עברו על הקובץ `compiler.ml`, סִפְרוּ בכמה מקומות עליכם להשלים קוד עבור כל מטלה, עברו על הפונקציות והטיפוסים השונים, חשבו טוב על הקשר בין הפונקציות הנתונות לכם לבין מה שאתם צריכים לכתוב, ונסו לתכנן את העבודה שלכם.
- שִׁמְרוּ את הקובץ `compiler.ml`, עם הקוד שכתבתם במטלות. הקובץ הזה והקוד שלכם יִהְוּוּ את הבסיס לפרוייקט הסופי.

העבודות שתעשו במסגרת הקורס הזה קשורות קשר הדוק לשאלות שתראו בבחינות. קל לנו מאד לשכנע את עצמנו שאנחנו מבינים את הקוד כשאנחנו מסתכלים על פתרון של מישהו אחר. אבל זו לא רק הונאה לפי **תקנון המשמעת**, זו קודם כל **הונאה־עצמית**: בבחינות אתם תתקלו בבעיות שדומות, אך לא בהכרח זהות, לאלו שראיתם בעבודות. אותם תלמידים שיכתבו את הקוד בעצמם, שיבינו כל בורג בקוד שלהם, יוכלו להתמודד בהצלחה עם האתגרים בבחינה. אלו שיסתפקו "בְּלְהָבִין" קוד של אחרים במקום ליצור אותו בעצמם, לא יהיו מסוגלים לאלתר פתרונות בזמן הבחינה באותה מדה של אפקטיביות.

3 מטלה 2: פרסר לשפת הקוד של סקים

- המטלה עוסקת בהשלמת הקוד החסר במודול `module Tag_Parser : TAG_PARSER`.
- אתם מקבלים מן המוכן פרסר לשפת ה־$data$ של סקים. אתם יכולים (וכדאי לכם!) להעזר בו בכתיבת הפרסר לשפת הקוד של סקים.
- אתם מקבלים שלד של פרסר לשפת הקוד של סקים. השלד כולל חלקים נרחבים מאד מהפרסר ומהרחבות המקרו.
- כפי שצויין, עליכם להשלים את החלקים החסרים בקוד, בהתאם להערות.

להלן דוגמאות להרצה של הפרסר, ולפלט המוחזר:

```
utop[12]> parse;;
- : string -> expr = <fun>

utop[13]> parse "496351";;
- : expr = ScmConst (ScmNumber (ScmInteger 496351))

utop[14]> parse "a";;
- : expr = ScmVarGet (Var "a")

utop[15]> parse " 'a ";;
- : expr = ScmConst (ScmSymbol "a")

utop[16]> parse " '234 ";;
- : expr = ScmConst (ScmNumber (ScmInteger 234))

utop[17]> parse " (if a b c) ";;
- : expr =
ScmIf (ScmVarGet (Var "a"), ScmVarGet (Var "b"),
   ScmVarGet (Var "c"))

utop[18]> parse " (if (< a b) (if (< a c) b c) (+ a b c)
   ) ";;
- : expr =
ScmIf
```

```
 (ScmApplic (ScmVarGet (Var "<"), [ScmVarGet (Var "a");
    ScmVarGet (Var "b")]),
 ScmIf
  (ScmApplic (ScmVarGet (Var "<"),
    [ScmVarGet (Var "a"); ScmVarGet (Var "c")]),
  ScmVarGet (Var "b"), ScmVarGet (Var "c")),
 ScmApplic (ScmVarGet (Var "+"),
  [ScmVarGet (Var "a"); ScmVarGet (Var "b"); ScmVarGet (
     Var "c")]))

utop[19]> parse " (set! a 3) ";;
- : expr = ScmVarSet (Var "a", ScmConst (ScmNumber (
   ScmInteger 3)))

utop[20]> parse " (define a 3) ";;
- : expr = ScmVarDef (Var "a", ScmConst (ScmNumber (
   ScmInteger 3)))

utop[21]> parse " (begin (a b) (c d) (e f)) ";;
- : expr =
ScmSeq
 [ScmApplic (ScmVarGet (Var "a"), [ScmVarGet (Var "b")])
    ;
  ScmApplic (ScmVarGet (Var "c"), [ScmVarGet (Var "d")])
     ;
  ScmApplic (ScmVarGet (Var "e"), [ScmVarGet (Var "f")])
     ]

utop[22]> parse " (or (a b) (c d) (e f)) ";;
- : expr =
ScmOr
 [ScmApplic (ScmVarGet (Var "a"), [ScmVarGet (Var "b")])
    ;
  ScmApplic (ScmVarGet (Var "c"), [ScmVarGet (Var "d")])
     ;
  ScmApplic (ScmVarGet (Var "e"), [ScmVarGet (Var "f")])
     ]

utop[23]> parse " (lambda (a b c) (+ a b c)) ";;
- : expr =
ScmLambda (["a"; "b"; "c"], Simple,
 ScmApplic (ScmVarGet (Var "+"),
  [ScmVarGet (Var "a"); ScmVarGet (Var "b"); ScmVarGet (
     Var "c")]))

utop[24]> parse " (lambda (a b . c) (list a b c)) ";;
- : expr =
ScmLambda (["a"; "b"], Opt "c",
 ScmApplic (ScmVarGet (Var "list"),
  [ScmVarGet (Var "a"); ScmVarGet (Var "b"); ScmVarGet (
     Var "c")]))

utop[25]> parse " (lambda a (list a b c)) ";;
```

```
- : expr =
ScmLambda ([], Opt "a",
 ScmApplic (ScmVarGet (Var "list"),
  [ScmVarGet (Var "a"); ScmVarGet (Var "b"); ScmVarGet (
     Var "c")]))

utop[26]> parse " (lambda () (list a b c)) ";;
- : expr =
ScmLambda ([], Simple,
 ScmApplic (ScmVarGet (Var "list"),
  [ScmVarGet (Var "a"); ScmVarGet (Var "b"); ScmVarGet (
     Var "c")]))

utop[39]> parse " `(a ,b ,@c d) ";;
- : expr =
ScmApplic (ScmVarGet (Var "cons"),
 [ScmConst (ScmSymbol "a");
  ScmApplic (ScmVarGet (Var "cons"),
   [ScmVarGet (Var "b");
    ScmApplic (ScmVarGet (Var "append"),
     [ScmVarGet (Var "c");
      ScmApplic (ScmVarGet (Var "cons"),
       [ScmConst (ScmSymbol "d"); ScmConst ScmNil])])])
          ])

utop[40]> parse "

(cond ((f? x) (f y) (f z))
      ((g? x) (g y) (g z))
      (else '()))

";;
- : expr =
ScmIf (ScmApplic (ScmVarGet (Var "f?"), [ScmVarGet (Var
   "x")]),
 ScmSeq
  [ScmApplic (ScmVarGet (Var "f"), [ScmVarGet (Var "y")
     ]);
   ScmApplic (ScmVarGet (Var "f"), [ScmVarGet (Var "z")
      ])],
 ScmIf (ScmApplic (ScmVarGet (Var "g?"), [ScmVarGet (Var
     "x")]),
  ScmSeq
   [ScmApplic (ScmVarGet (Var "g"), [ScmVarGet (Var "y")
      ]);
    ScmApplic (ScmVarGet (Var "g"), [ScmVarGet (Var "z")
       ])],
  ScmConst ScmNil))

utop[4]> parse "(let () (+ a b))";;
- : expr =
ScmApplic
 (ScmLambda ([], Simple,
```

```
   ScmApplic (ScmVarGet (Var "+"),
    [ScmVarGet (Var "a"); ScmVarGet (Var "b")])),
 [])

utop[1]> parse "(let ((a 2) (b 3)) (+ a b))";;
- : expr =
ScmApplic
 (ScmLambda (["a"; "b"], Simple,
   ScmApplic (ScmVarGet (Var "+"),
    [ScmVarGet (Var "a"); ScmVarGet (Var "b")])),
 [ScmConst (ScmNumber (ScmInteger 2)); ScmConst (
    ScmNumber (ScmInteger 3))])

utop[13]> parse "
(let ((a 2) (b 3) (c 5) (d 8))
  (+ a b c d e f))
";;
- : expr =
ScmApplic
 (ScmLambda (["a"; "b"; "c"; "d"], Simple,
   ScmApplic (ScmVarGet (Var "+"),
    [ScmVarGet (Var "a"); ScmVarGet (Var "b"); ScmVarGet
        (Var "c");
     ScmVarGet (Var "d"); ScmVarGet (Var "e"); ScmVarGet
         (Var "f")])),
 [ScmConst (ScmNumber (ScmInteger 2)); ScmConst (
    ScmNumber (ScmInteger 3));
  ScmConst (ScmNumber (ScmInteger 5)); ScmConst (
     ScmNumber (ScmInteger 8))])

utop[11]> parse "
(let ((a 2) (b 3) (t 'moshe!))
  (set! t (+ a b))
  (set! a b)
  (set! b t)
  (list a b))
";;
- : expr =
ScmApplic
 (ScmLambda (["a"; "b"; "t"], Simple,
   ScmSeq
    [ScmVarSet (Var "t",
      ScmApplic (ScmVarGet (Var "+"),
       [ScmVarGet (Var "a"); ScmVarGet (Var "b")]));
     ScmVarSet (Var "a", ScmVarGet (Var "b"));
     ScmVarSet (Var "b", ScmVarGet (Var "t"));
     ScmApplic (ScmVarGet (Var "list"),
      [ScmVarGet (Var "a"); ScmVarGet (Var "b")])]),
 [ScmConst (ScmNumber (ScmInteger 2)); ScmConst (
    ScmNumber (ScmInteger 3));
  ScmConst (ScmSymbol "moshe!")])

utop[3]> parse "(let* () (+ a b))";;
```

```
- : expr =
ScmApplic
 (ScmLambda ([], Simple,
   ScmApplic (ScmVarGet (Var "+"),
    [ScmVarGet (Var "a"); ScmVarGet (Var "b")])),
 [])

utop[2]> parse "(let* ((a 2) (b 3)) (+ a b))";;
- : expr =
ScmApplic
 (ScmLambda (["a"], Simple,
   ScmApplic
    (ScmLambda (["b"], Simple,
      ScmApplic (ScmVarGet (Var "+"),
       [ScmVarGet (Var "a"); ScmVarGet (Var "b")])),
    [ScmConst (ScmNumber (ScmInteger 3))])),
 [ScmConst (ScmNumber (ScmInteger 2))])

utop[41]> parse "

(letrec ((fact
           (lambda (n)
             (if (zero? n)
                 1
               (* n (fact (- n 1)))))))
  (+ (fact 5) (fact 55)))

";;
- : expr =
ScmApplic
 (ScmLambda (["fact"], Simple,
   ScmSeq
    [ScmVarSet (Var "fact",
      ScmLambda (["n"], Simple,
       ScmIf (ScmApplic (ScmVarGet (Var "zero?"), [
          ScmVarGet (Var "n")]),
        ScmConst (ScmNumber (ScmInteger 1)),
        ScmApplic (ScmVarGet (Var "*"),
         [ScmVarGet (Var "n");
          ScmApplic (ScmVarGet (Var "fact"),
           [ScmApplic (ScmVarGet (Var "-"),
             [ScmVarGet (Var "n"); ScmConst (ScmNumber (
                ScmInteger 1))])])])))));
     ScmApplic (ScmVarGet (Var "+"),
      [ScmApplic (ScmVarGet (Var "fact"),
        [ScmConst (ScmNumber (ScmInteger 5))]);
       ScmApplic (ScmVarGet (Var "fact"),
        [ScmConst (ScmNumber (ScmInteger 55))])])]),
 [ScmConst (ScmSymbol "whatever")])

utop[42]> parse "

(cond ((f x) => (g y))
```

```
      ((f y) => (g x))
      ((p? x y) => (+ x y))
      (else (+ (* x x) (* y y))))

";;
- : expr =
ScmApplic
 (ScmLambda (["value"; "f"; "rest"], Simple,
   ScmIf (ScmVarGet (Var "value"),
    ScmApplic (ScmApplic (ScmVarGet (Var "f"), []),
     [ScmVarGet (Var "value")]),
    ScmApplic (ScmVarGet (Var "rest"), []))),
 [ScmApplic (ScmVarGet (Var "f"), [ScmVarGet (Var "x")])
    ;
  ScmLambda ([], Simple,
   ScmApplic (ScmVarGet (Var "g"), [ScmVarGet (Var "y")
      ]));
  ScmLambda ([], Simple,
   ScmApplic
    (ScmLambda (["value"; "f"; "rest"], Simple,
      ScmIf (ScmVarGet (Var "value"),
       ScmApplic (ScmApplic (ScmVarGet (Var "f"), []),
        [ScmVarGet (Var "value")]),
       ScmApplic (ScmVarGet (Var "rest"), []))),
    [ScmApplic (ScmVarGet (Var "f"), [ScmVarGet (Var "y"
       )]);
     ScmLambda ([], Simple,
      ScmApplic (ScmVarGet (Var "g"), [ScmVarGet (Var "x
         ")]));
     ScmLambda ([], Simple,
      ScmApplic
       (ScmLambda (["value"; "f"; "rest"], Simple,
         ScmIf (ScmVarGet (Var "value"),
          ScmApplic (ScmApplic (ScmVarGet (Var "f"), [])
             ,
           [ScmVarGet (Var "value")]),
          ScmApplic (ScmVarGet (Var "rest"), []))),
       [ScmApplic (ScmVarGet (Var "p?"),
         [ScmVarGet (Var "x"); ScmVarGet (Var "y")]);
        ScmLambda ([], Simple,
         ScmApplic (ScmVarGet (Var "+"),
          [ScmVarGet (Var "x"); ScmVarGet (Var "y")]));
        ScmLambda ([], Simple,
         ScmApplic (ScmVarGet (Var "+"),
          [ScmApplic (ScmVarGet (Var "*"),
            [ScmVarGet (Var "x"); ScmVarGet (Var "x")]);
           ScmApplic (ScmVarGet (Var "*"),
            [ScmVarGet (Var "y"); ScmVarGet (Var "y")])
               ]))]))]))])

utop[44]> parse "

(if (zero? x) (display \"just lovely!\"))
```

```
";;
- : expr =
ScmIf (ScmApplic (ScmVarGet (Var "zero?"), [ScmVarGet (
   Var "x")]),
 ScmApplic (ScmVarGet (Var "display"), [ScmConst (
    ScmString "just lovely!")]),
 ScmConst ScmVoid)
```

4 מטלה 3: מנתח סמנטי לסקים

- המטלה עוסקת בהשלמת הקוד החסר במודול
module Semantic_Analysis : SEMANTIC_ANALYSIS.
- אתם מקבלים את כל פרוצדורות העזר הדרושות להשלמת המודול.
- כפי שצויין, עליכם להשלים את החלקים החסרים בקוד, בהתאם להערות.

להלן דוגמאות להרצה של המנתח הסמנטי, ולפלט המוחזר:

```
utop[27]> sem;;
- : string -> expr' = <fun>

utop[28]> sem " 234  ";;
- : expr' = ScmConst' (ScmNumber (ScmInteger 234))

utop[29]> sem " a  ";;
- : expr' = ScmVarGet' (Var' ("a", Free))

utop[30]> sem " 'a  ";;
- : expr' = ScmConst' (ScmSymbol "a")

utop[31]> sem " ''a  ";;
- : expr' =
ScmConst' (ScmPair (ScmSymbol "quote", ScmPair (
   ScmSymbol "a", ScmNil)))

utop[32]> sem " (+ a b c)  ";;
- : expr' =
ScmApplic' (ScmVarGet' (Var' ("+", Free)),
 [ScmVarGet' (Var' ("a", Free)); ScmVarGet' (Var' ("b",
    Free));
  ScmVarGet' (Var' ("c", Free))],
 Non_Tail_Call)

utop[33]> sem " (lambda (a) (lambda (b c) (+ a b c)))  "
   ;;
- : expr' =
ScmLambda' (["a"], Simple,
 ScmLambda' (["b"; "c"], Simple,
  ScmApplic' (ScmVarGet' (Var' ("+", Free)),
   [ScmVarGet' (Var' ("a", Bound (0, 0))); ScmVarGet' (
      Var' ("b", Param 0));
```

```
    ScmVarGet' (Var' ("c", Param 1))],
   Tail_Call)))

utop[12]> sem "
(let ((a 2) (b 3) (t 'moshe!))
  (set! t (+ a b))
  (set! a b)
  (set! b t)
  (list a b))
";;
- : expr' =
ScmApplic'
 (ScmLambda' (["a"; "b"; "t"], Simple,
   ScmSeq'
    [ScmVarSet' (Var' ("t", Param 2),
      ScmApplic' (ScmVarGet' (Var' ("+", Free)),
       [ScmVarGet' (Var' ("a", Param 0)); ScmVarGet' (
          Var' ("b", Param 1))],
       Non_Tail_Call));
     ScmVarSet' (Var' ("a", Param 0), ScmVarGet' (Var' (
        "b", Param 1)));
     ScmVarSet' (Var' ("b", Param 1), ScmVarGet' (Var' (
        "t", Param 2)));
     ScmApplic' (ScmVarGet' (Var' ("list", Free)),
      [ScmVarGet' (Var' ("a", Param 0)); ScmVarGet' (Var
         ' ("b", Param 1))],
      Tail_Call)]),
 [ScmConst' (ScmNumber (ScmInteger 2)); ScmConst' (
    ScmNumber (ScmInteger 3));
  ScmConst' (ScmSymbol "moshe!")],
 Non_Tail_Call)

utop[34]> sem " (lambda (a) (lambda (b c) (+ (* a a) (*
   b b) (* c c))))  ";;
- : expr' =
ScmLambda' (["a"], Simple,
 ScmLambda' (["b"; "c"], Simple,
  ScmApplic' (ScmVarGet' (Var' ("+", Free)),
   [ScmApplic' (ScmVarGet' (Var' ("*", Free)),
     [ScmVarGet' (Var' ("a", Bound (0, 0)));
      ScmVarGet' (Var' ("a", Bound (0, 0)))],
     Non_Tail_Call);
    ScmApplic' (ScmVarGet' (Var' ("*", Free)),
     [ScmVarGet' (Var' ("b", Param 0)); ScmVarGet' (Var'
         ("b", Param 0))],
     Non_Tail_Call);
    ScmApplic' (ScmVarGet' (Var' ("*", Free)),
     [ScmVarGet' (Var' ("c", Param 1)); ScmVarGet' (Var'
         ("c", Param 1))],
     Non_Tail_Call)],
   Tail_Call)))

utop[35]> sem "
```

```
(let ((a 0))
  (list (lambda () a)
        (lambda (v) (set! a v))))

";;
- : expr' =
ScmApplic'
 (ScmLambda' (["a"], Simple,
   ScmSeq'
    [ScmVarSet' (Var' ("a", Param 0), ScmBox' (Var' ("a"
       , Param 0)));
     ScmApplic' (ScmVarGet' (Var' ("list", Free)),
      [ScmLambda' ([], Simple, ScmBoxGet' (Var' ("a",
         Bound (0, 0))));
       ScmLambda' (["v"], Simple,
        ScmBoxSet' (Var' ("a", Bound (0, 0)),
         ScmVarGet' (Var' ("v", Param 0))))],
      Tail_Call)]),
 [ScmConst' (ScmNumber (ScmInteger 0))], Non_Tail_Call)

utop[36]> sem "
(+ (let () (* a a)) (let () (* b b)))

";;
- : expr' =
ScmApplic' (ScmVarGet' (Var' ("+", Free)),
 [ScmApplic'
   (ScmLambda' ([], Simple,
     ScmApplic' (ScmVarGet' (Var' ("*", Free)),
      [ScmVarGet' (Var' ("a", Free)); ScmVarGet' (Var' (
         "a", Free))],
      Tail_Call)),
   [], Non_Tail_Call);
  ScmApplic'
   (ScmLambda' ([], Simple,
     ScmApplic' (ScmVarGet' (Var' ("*", Free)),
      [ScmVarGet' (Var' ("b", Free)); ScmVarGet' (Var' (
         "b", Free))],
      Tail_Call)),
   [], Non_Tail_Call)],
 Non_Tail_Call)

utop[14]> sem "
(let ((a 2) (b 3) (c 5) (d 8))
  (+ a b c d e f))
";;
- : expr' =
ScmApplic'
 (ScmLambda' (["a"; "b"; "c"; "d"], Simple,
   ScmApplic' (ScmVarGet' (Var' ("+", Free)),
    [ScmVarGet' (Var' ("a", Param 0)); ScmVarGet' (Var'
       ("b", Param 1));
     ScmVarGet' (Var' ("c", Param 2)); ScmVarGet' (Var'
```

```
        ("d", Param 3));
     ScmVarGet' (Var' ("e", Free)); ScmVarGet' (Var' ("f
        ", Free))],
    Tail_Call)),
 [ScmConst' (ScmNumber (ScmInteger 2)); ScmConst' (
    ScmNumber (ScmInteger 3));
  ScmConst' (ScmNumber (ScmInteger 5)); ScmConst' (
     ScmNumber (ScmInteger 8))],
 Non_Tail_Call)

utop[37]> sem "

(cond ((f? x) (f y) (f z))
      ((g? x) (g y) (g z))
      (else '()))

";;
- : expr' =
ScmIf'
 (ScmApplic' (ScmVarGet' (Var' ("f?", Free)),
   [ScmVarGet' (Var' ("x", Free))], Non_Tail_Call),
 ScmSeq'
  [ScmApplic' (ScmVarGet' (Var' ("f", Free)),
    [ScmVarGet' (Var' ("y", Free))], Non_Tail_Call);
   ScmApplic' (ScmVarGet' (Var' ("f", Free)),
    [ScmVarGet' (Var' ("z", Free))], Non_Tail_Call)],
 ScmIf'
  (ScmApplic' (ScmVarGet' (Var' ("g?", Free)),
    [ScmVarGet' (Var' ("x", Free))], Non_Tail_Call),
  ScmSeq'
   [ScmApplic' (ScmVarGet' (Var' ("g", Free)),
     [ScmVarGet' (Var' ("y", Free))], Non_Tail_Call);
    ScmApplic' (ScmVarGet' (Var' ("g", Free)),
     [ScmVarGet' (Var' ("z", Free))], Non_Tail_Call)],
  ScmConst' ScmNil))
```

5 הוראות הגשה

1. אם אתם מגישים בזוגות, רק אחד משני השותפים צריך להגיש את המטלה בשם הקבוצה
2. עליכם להגיש קובץ *zip* ששמו מורכב ממספרי תעודות הזהות של השותפים, מופרדים על ידי קו־תחתון, או מספר תעודת הזהות במקרה של מגיש יחיד, והסיומת *zip*
3. קובץ ה־*zip* צריך ליצור תיקיה בשם hw23
4. אנא צרפו קובץ readme.txt שיכיל עבור כל אחד מהשותפים את השם המלא ואת מספר תעודת הזהות
5. כל הקבצים הרלוונטיים צריכים להמצא בתיקיה
6. בבקשה השאירו זמן פנוי לבדוק את קובץ ה־*zip* בטרם תגישו אותו: פתחו אותו בתיקיה חדשה, וודאו שכל הקבצים הדרושים שם!

7. אני מזכיר לכם שוב לשמור את הקובץ `compiler.ml`, משום שהוא ישמש אתכם בפרוייקט הסופי!

מבנה הקובץ ה־zip וקבצים שבו הוא כדלקמן:

- `123456789_987654321.zip`
  - `hw23`
    - `pc.ml`
    - `compiler.ml`
    - `readme.txt`