עקרונות הקומפילציה – מטלה 1

מאיר גולדברג

הגשה: 29 נובמבר, 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 ל**תקנון 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 לממש 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 פרסר לשפת *infix*

במטלה זו אתם מתבקשים לממש, בשפת אוקמל, את הפרסר `nt_expr` לתחביר *infix* פשוט של ביטויים. עליכם לתמוך בצורות הבאות:

- מספרים שלמים, חיוביים ושליליים
- משתנים: מתחילים באות אנגלית, וכוללים את התווים `a` $\cdots$ `z`, `A` $\cdots$ `Z`, `0` $\cdots$ `9`, והתווים המיוחדים `$`, `_`
- פעולות אריתמטיות של חיבור $E+E$, חיסור $E-E$, כפל $E*E$, חילוק E/E, שארית $E \mod E$, וחזקה $E \hat{} E$
- סוגריים [עגולים] מקוננים במדה שרירותית
- פעולות של שלילה $(-E)$ והופכי $(/E)$
- קריאות לפונקציה $E(E_1, \ldots, E_n)$

- מציאת איבר במערך לפי אינדקס $E[E]$
- פעולות עם אחוזים: חיבור $x + y\%$, חיסור $x - y\%$, אחוז **מתוך** מספר $x * y\%$

את הפעולות האריתמטיות יש לקרוא לפי הקדימויות הרגילות הנוהגות במתמטיקה ובשפות תכנות רבות, לפיהן פעולות כפל תופסות לפני חיבור, פעולות חזקה לפני כפל, וכו'.
כל הפעולות האריתמטיות הן עם אסוציאציה שמאלית, למעט פעולת החזקה שיש לה אסוציאציה ימנית.
להלן מספר דוגמאות:

```
utop[5]> test_string nt_expr "1" 0;;
- : expr parsing_result = {index_from = 0; index_to = 1;
    found = Num 1}

utop[6]> test_string nt_expr "x" 0;;
- : expr parsing_result = {index_from = 0; index_to = 1;
    found = Var "x"}

utop[7]> test_string nt_expr "x + y" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 5; found = BinOp (Add, Var "
   x", Var "y")}

utop[8]> test_string nt_expr "x + y * z ^ t" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 13;
 found =
  BinOp (Add, Var "x", BinOp (Mul, Var "y", BinOp (Pow,
     Var "z", Var "t")))}

utop[9]> test_string nt_expr "2 * 3 + 4 * 5" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 13;
 found = BinOp (Add, BinOp (Mul, Num 2, Num 3), BinOp (
    Mul, Num 4, Num 5))}

utop[10]> test_string nt_expr "1 + 2 * 3 + 4" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 13;
 found = BinOp (Add, BinOp (Add, Num 1, BinOp (Mul, Num
    2, Num 3)), Num 4)}

utop[11]> test_string nt_expr "(1 + 2) * (3 + 4)" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 17;
 found = BinOp (Mul, BinOp (Add, Num 1, Num 2), BinOp (
    Add, Num 3, Num 4))}

utop[12]> test_string nt_expr "1 * 2 * 3 * 4 * 5" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 17;
 found =
  BinOp (Mul,
   BinOp (Mul, BinOp (Mul, BinOp (Mul, Num 1, Num 2),
      Num 3), Num 4),
```

```
   Num 5)}

utop[13]> test_string nt_expr "1 ^ 2 ^ 3 ^ 4 ^ 5" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 17;
 found =
  BinOp (Pow, Num 1,
   BinOp (Pow, Num 2, BinOp (Pow, Num 3, BinOp (Pow, Num
       4, Num 5))))}

utop[14]> test_string nt_expr "a(b, c)" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 7; found = Call (Var "a", [
   Var "b"; Var "c"])}

utop[15]> test_string nt_expr "f(x, 1, y, 2)" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 13;
 found = Call (Var "f", [Var "x"; Num 1; Var "y"; Num
    2])}

utop[16]> test_string nt_expr "f()" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 3; found = Call (Var "f",
   [])}

utop[17]> test_string nt_expr "f()()" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 5; found = Call (Call (Var "
   f", []), [])}

utop[18]> test_string nt_expr "f()(x)()(y)" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 11;
 found = Call (Call (Call (Call (Var "f", []), [Var "x"
    ]), []), [Var "y"])}

utop[19]> test_string nt_expr "A[i]" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 4; found = Deref (Var "A",
   Var "i")}

utop[20]> test_string nt_expr "A[3]" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 4; found = Deref (Var "A",
   Num 3)}

utop[21]> test_string nt_expr "A[3][4]" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 7; found = Deref (Deref (Var
    "A", Num 3), Num 4)}

utop[22]> test_string nt_expr "f(1, 2)[3](4)[5](6, 7, 8)
```

```
   " 0;;
- : expr parsing_result =
{index_from = 0; index_to = 25;
 found =
  Call
   (Deref (Call (Deref (Call (Var "f", [Num 1; Num 2]),
      Num 3), [Num 4]),
     Num 5),
   [Num 6; Num 7; Num 8])}

utop[23]> test_string nt_expr "a[1] + a[2] * a[3] ^ a[4]
   " 0;;
- : expr parsing_result =
{index_from = 0; index_to = 25;
 found =
  BinOp (Add, Deref (Var "a", Num 1),
   BinOp (Mul, Deref (Var "a", Num 2),
    BinOp (Pow, Deref (Var "a", Num 3), Deref (Var "a",
       Num 4))))}

utop[24]> test_string nt_expr "1 + (2) * ((3)) ^ (((4)))
   " 0;;
- : expr parsing_result =
{index_from = 0; index_to = 25;
 found = BinOp (Add, Num 1, BinOp (Mul, Num 2, BinOp (
    Pow, Num 3, Num 4)))}

utop[25]> test_string nt_expr "1 + -1" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 6; found = BinOp (Add, Num
   1, Num (-1))}

utop[26]> test_string nt_expr "1 + -1 - -3" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 11;
 found = BinOp (Sub, BinOp (Add, Num 1, Num (-1)), Num
    (-3))}

utop[27]> test_string nt_expr "1 + -1 - -3 + (- x)" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 19;
 found =
  BinOp (Add, BinOp (Sub, BinOp (Add, Num 1, Num (-1)),
     Num (-3)),
   BinOp (Sub, Num 0, Var "x"))}

utop[28]> test_string nt_expr "(/ x) * (- y)" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 13;
 found =
  BinOp (Mul, BinOp (Div, Num 1, Var "x"), BinOp (Sub,
     Num 0, Var "y"))}
```

```
utop[15]> test_string nt_expr "a mod b" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 7; found = BinOp (Mod, Var "
   a", Var "b")}

utop[16]> test_string nt_expr "m mod b" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 7; found = BinOp (Mod, Var "
   m", Var "b")}

utop[17]> test_string nt_expr "mod mod b" 0;;
Exception: PC.X_no_match.

utop[18]> test_string nt_expr "mod + b" 0;;
Exception: PC.X_no_match.

utop[19]> test_string nt_expr "modest" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 6; found = Var "modest"}

utop[20]> test_string nt_expr "mode" 0;;
- : expr parsing_result = {index_from = 0; index_to = 4;
    found = Var "mode"}

utop[21]> test_string nt_expr "mod" 0;;
Exception: PC.X_no_match.

utop[22]> test_string nt_expr "a + b mod c" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 11;
 found = BinOp (Add, Var "a", BinOp (Mod, Var "b", Var "
    c"))}

utop[23]> test_string nt_expr "a * b mod c" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 11;
 found = BinOp (Mod, BinOp (Mul, Var "a", Var "b"), Var
    "c")}

utop[24]> test_string nt_expr "mod mod mod" 0;;
Exception: PC.X_no_match.

utop[25]> test_string nt_expr "mode mod mo" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 11; found = BinOp (Mod, Var
   "mode", Var "mo")}

utop[26]> test_string nt_expr "mode mod mod" 0;;
- : expr parsing_result = {index_from = 0; index_to = 5;
    found = Var "mode"}
```

2.1 פעולות עם אחוזים

שפת הנוסחאות בגליונות אלקטרוניים מנגישה פונקציות שמחשבות חישובים עם אחוזים. גם מחשבונים פשוטים ומדעיים תומכים לרוב בפעולות אריתמטיות עם אחוזים, אבל לרוב עם כללי רישום יחודיים. עד כמה שאני יודע, אין אף שפת תכנות כללית שתומכת באופרטור של אחוזים (%) באופן אלגברי, כלומר באופן שבו רושמים על נייר ביטויים עם אחוזים.
הפרסר שלכם צריך לתמוך בשלוש פעולות עם אחוזים:

- חיבור של אחוזים $\mathscr{E}_1+\mathscr{E}_2\%$. הביטויים $\mathscr{E}_1$, $\mathscr{E}_2$ הם ביטויים כלליים, שיכולים, כל אחד מהם, לכלול גם שימוש באופרטור האחוזים!
- חיסור של אחוזים $\mathscr{E}_1-\mathscr{E}_2\%$. הביטויים $\mathscr{E}_1$, $\mathscr{E}_2$ הם ביטויים כלליים, שיכולים, כל אחד מהם, לכלול גם שימוש באופרטור האחוזים!
- כפל של אחוזים $\mathscr{E}_1*\mathscr{E}_2\%$. הכוונה כאן ל־$\mathscr{E}_2$ אחוזים מתוך $\mathscr{E}_1$, שזה כמובן אותו הדבר כמו $\mathscr{E}_1$ אחוזים מתוך $\mathscr{E}_1$. הביטויים $\mathscr{E}_1$, $\mathscr{E}_2$ הם ביטויים כלליים, שיכולים, כל אחד מהם, לכלול גם שימוש באופרטור האחוזים!

מה שמורכב בביטויים עם אחוזים זה להבין את **הקדימות** של ביטויים כאלה ביחס לביטויים עם אופרטורים אחרים. מחשבונים לא עוזרים בכך: כל מחשבון ממש באופן ייחודי קדימויות של ביטויים עם אחוזים, ואין סטנדרט מקובל בנושא הזה. כדי להבין מהי התנהגות סבירה, צריך לזכור שלמרות שביטוי כמו $\mathscr{E}_1\pm\mathscr{E}_2\%$ **נראה כמו ביטוי חיבור או חיסור**, הרי שמבחינה מתמטית הוא ביטוי **כפל** שערכו $\left[\mathscr{E}_1 \cdot \left(1 \pm \frac{\mathscr{E}_2}{100}\right)\right]$, ולמרות שאנחנו **לא נמיר** ביטויי חיבור או חיסור של אחוזים לביטויי כפל, ויש לנו מבנים מיוחדים עבור ביטויים כאלה בדקדוק שלנו, אנחנו כן נתייחס אליהם כסוג של ביטויי כפל מבחינת הקדימות. באופן דומה, ערכו של הביטוי $\mathscr{E}_1*\mathscr{E}_2\%$ הוא $\left(\frac{\mathscr{E}_1\cdot\mathscr{E}_2}{100}\right)$, ואנחנו נתייחס גם לביטוי הזה כסוג של ביטוי כפל.
להלן מספר דוגמאות של ביטויים עם אחוזים:

```
utop[29]> test_string nt_expr "2 * 3 + 4%" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 10;
 found = BinOp (Mul, Num 2, BinOp (AddPer, Num 3, Num 4)
    )}

utop[30]> test_string nt_expr "2 / 3 + 4%" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 10;
 found = BinOp (Div, Num 2, BinOp (AddPer, Num 3, Num 4)
    )}

utop[31]> test_string nt_expr "2 / 3 + (2 + 5%)%" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 17;
 found =
  BinOp (Div, Num 2, BinOp (AddPer, Num 3, BinOp (AddPer
     , Num 2, Num 5)))}

utop[32]> test_string nt_expr "2 + 50%" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 7; found = BinOp (AddPer,
   Num 2, Num 50)}

utop[33]> test_string nt_expr "5 * 2 + 50%" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 11;
```

```
 found = BinOp (Mul, Num 5, BinOp (AddPer, Num 2, Num
    50))}

utop[34]> test_string nt_expr "5 * 2 + (50)%" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 13;
 found = BinOp (Mul, Num 5, BinOp (AddPer, Num 2, Num
    50))}

utop[35]> test_string nt_expr "5 * 2 + (50 - 25%)%" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 19;
 found =
  BinOp (Mul, Num 5, BinOp (AddPer, Num 2, BinOp (SubPer
     , Num 50, Num 25)))}

utop[36]> test_string nt_expr "5 * 5%" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 6; found = BinOp (PerOf, Num
    5, Num 5)}

utop[37]> test_string nt_expr "6 + 5 * 5%" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 10;
 found = BinOp (Add, Num 6, BinOp (PerOf, Num 5, Num 5))
    }

utop[38]> test_string nt_expr "7 / 6 + 5 * 5%" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 14;
 found = BinOp (Add, BinOp (Div, Num 7, Num 6), BinOp (
    PerOf, Num 5, Num 5))}

utop[39]> test_string nt_expr "7 / (6 + 5 * 5%)" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 16;
 found = BinOp (Div, Num 7, BinOp (Add, Num 6, BinOp (
    PerOf, Num 5, Num 5)))}

utop[40]> test_string nt_expr "10 / 5 - 20%" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 12;
 found = BinOp (Div, Num 10, BinOp (SubPer, Num 5, Num
    20))}

utop[41]> test_string nt_expr "10 / 5 - 20% + 100%" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 19;
 found =
  BinOp (Div, Num 10, BinOp (AddPer, BinOp (SubPer, Num
     5, Num 20), Num 100))}

utop[42]> test_string nt_expr "10 / 5 - 20% + 100% - 6%"
```

```
    0;;
- : expr parsing_result =
{index_from = 0; index_to = 24;
 found =
  BinOp (Div, Num 10,
   BinOp (SubPer, BinOp (AddPer, BinOp (SubPer, Num 5,
      Num 20), Num 100),
    Num 6))}
```

כפי שלבטח שמתם לב, אין כאן תיאור פורמאלי ומדוייק של הדקדוק, וחלק מהעבודה היא להגדיר את הדיוק המתאים בהתאם לדוגמאות שאתם רואים כאן. כדאי לכם לחשוב על החלק הזה לפני שאתם ניגשים לממש את העבודה!

2.2 דוגמאות נוספות עם שלילה והופכי

חברי הכתה (בקבוצת הוואצאפ) בקשו להפיק עוד דוגמאות לשלילה והופכי, אז לבקשתם, הנה כמה דוגמאות. יתכן והן תראנה לכם פחות טבעיות, אבל זכרו־נא ששלילה והופכי הם אופרטורים מסוג unary operators in prefix notation, ולכן הקדימויות הגיוניות (וגם פשוטות יותר למימוש!).

```
utop[3]> test_string nt_expr "(- a + b)" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 9;
 found = BinOp (Sub, Num 0, BinOp (Add, Var "a", Var "b"
    ))}
utop[4]> test_string nt_expr "(- a / b)" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 9;
 found = BinOp (Sub, Num 0, BinOp (Div, Var "a", Var "b"
    ))}
utop[5]> test_string nt_expr "(- a mod b)" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 11;
 found = BinOp (Sub, Num 0, BinOp (Mod, Var "a", Var "b"
    ))}
utop[6]> test_string nt_expr "(- a - b)" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 9;
 found = BinOp (Sub, Num 0, BinOp (Sub, Var "a", Var "b"
    ))}
utop[7]> test_string nt_expr "((- a) - b)" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 11;
 found = BinOp (Sub, BinOp (Sub, Num 0, Var "a"), Var "b
    ")}
utop[8]> test_string nt_expr "(/ a + b%)" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 10;
 found = BinOp (Div, Num 1, BinOp (AddPer, Var "a", Var
    "b"))}
utop[9]> test_string nt_expr "(/ a * b%)" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 10;
 found = BinOp (Div, Num 1, BinOp (PerOf, Var "a", Var "
    b"))}
```

```
utop[10]> test_string nt_expr "(/ a - b%)" 0;;
- : expr parsing_result =
{index_from = 0; index_to = 10;
 found = BinOp (Div, Num 1, BinOp (SubPer, Var "a", Var
    "b"))}
```

3 כיצד להתחיל

1. אתם מקבלים את הקובץ `hw1.ml`, שמכיל את השלד של הקוד, טיפוסים והגדרות שונות. אין לשנות את הטיפוסים והחתימות בקובץ, משום שזה ישבור את קוד הבדיקה האוטומטית.

2. נתון לכם המודול `InfixParser : INFIX_PARSER`. מופיעה שם ההגדרה

זוהי הגדרה זמנית, וכשתטענו את הקוד יזרק לכם החריג:
שפירושו שעליכם להגדיר מחדש את המשתנה `nt_expr` להיות ה־*parser* לביטויים ב־*infix* עם אחוזים.

3.1 היכן ללמוד עוד על חבילת ה־*parsing combinators*

תהיינה הרצאות בכתה שתעסוקנה בחבילת ה־*parsing combinators*, וגם תרגולים בנושא.
העליתי סרטונים ליוטיוב שעוסקים בשימוש בחבילה לכתיבת פרסרים שונים, כולל כאלו עם זיהוי מספרים שונים, ביטויים בתחביר *infix*, וכו'. לנוחותכם, אני מצרף קישורים לסרטונים רלוונטיים:

1. מבוא כללי לחבילת ה־*parsing combinators*

2. דוגמאות לזיהוי של סוגים שונים של מספרים באמצעות חבילת ה־*parsing combinators*

3. דוגמאות ל־*parsing* של ביטויי *infix* עם קדימויות, באמצעות חבילת ה־*parsing combinators*

4 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 `hw1`

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!

מבנה הקובץ ה־*zip* וקבצים שבו הוא כדלקמן:

- `123456789_987654321.zip`
  - `hw1`
    - `hw1.ml`
    - `pc.ml`
    - `readme.txt`