

מושגים בשפות תכנות

תרגיל 4

להגשה עד 06/06/2019

בתרגיל זה נבנה Parser ו Interpreter ל λ -calculus בשפת OCaml.

הנחיות כלליות:

- בתרגיל נעשה שימוש ב OCaml בגרסת 4.02. כל הפתרונות צריכים לרוץ עם גרסה זו ולהתקמפל ללא שגיאות עם הפקודות שמפורטות ב build.txt.
- התרגיל אינו תלוי בספריית Core, וניתן לפתור אותו בסביבת Windows. הקובץ utils.ml מכיל פונקציות שימושיות שאינן חלק מהספרייה הסטנדרטית של OCaml.
- בתרגיל זה כל הקוד צריך להיות "טהור", כלומר ללא side-effects, למעט הדפסות פלט.
- סגנון התכנות צריך להיות בהתאם למה שנלמד בשיעורים ובתרגול: שימוש נרחב ב pattern-matching ופונקציות רקורסיביות (במקום ביטויי תנאי ולולאות).
- כל השינויים בקבצים צריכים להיות במקומות המסומנים בהם. אין לשנות בקבצים דבר מלבד במקומות אלה.
- באחריותכם לבדוק את הקוד שכתבת על דוגמאות נוספות ולוודא את נכונותו.
- מומלץ לקרוא את התרגיל עד סופו לפני שמתחילים לפתור אותו.

1. בשאלה זו נבנה Parser ל λ -calculus מורחב שכולל גם let expressions. התחביר הקונקרטי (concrete syntax) מוגדר ע"י הדקדוק הבא. אנו משתמשים בסמל \ במקום λ :
$$t ::= id \mid (\lambda id. t) \mid (t1\ t2) \mid (t) \mid \text{let } id=t1 \text{ in } t2$$

שימו לב שהדקדוק מחייב סוגריים מסביב ל λ -abstractions ו applications. לדוגמה, המחרוזת הבאה היא מילה חוקית בשפה:

```
let tru = ( $\lambda t. (\lambda f. t)$ ) in
let fls = ( $\lambda t. (\lambda f. f)$ ) in
let and = ( $\lambda b. (\lambda c. ((b\ c)\ fls))$ ) in
((and\ tru)\ fls)
```

הייצוג הפנימי לביטויים בשפה הוא AST, שניתן ע"י התחביר האבסטרקטי (abstract syntax) הבא:

```
term ::= id \mid \lambda id.term \mid term1\ term2
```

הקובץ lexer.ml מכיל את ה Lexer המלא עבור שפה זו (אין לשנות קובץ זה), ומגדיר את הטיפוס token. הקובץ parser.ml מגדיר את הטיפוס הבא, שמשמש לייצוג ה AST (אין לשנות טיפוס זה):

```
type term = Variable of string
          | Abstraction of string * term
```

| Application of term * term

בקובץ parser.ml עליכם לממש את הפונקציות הבאות:

```
parse_term : token list -> term * token list
parse : string -> term
format_term : term -> string
```

- הפונקציה parse_term מקבלת רשימה של tokens ומחזירה term ורשימה של tokens שנשארו, בדומה לדוגמה שראינו בתרגול עבור ביטויים רגולריים. הפונקציה צריכה לזרוק SyntaxError במידה וה parsing נכשל. הפונקציה מטפלת בביטויים מהצורה let x=t1 in t2 ע"י ייצוגם בתחביר האבסטרקטי כך:

$(x. t2) t1$

(השתכנעו שזהו אכן ייצוג שמשמר את המשמעות של let expressions כפי שאנו מכירים אותם).

- הפונקציה parse מקבלת מחרוזת ומחזירה את ה term שהיא מייצגת, או זורקת SyntaxError במידה והמחרוזת אינה מכילה מילה בשפה (לפי הדקדוק של התחביר הקונקרטי).

- הפונקציה format_term מקבלת term ומחזירה ייצוג שלו באמצעות מחרוזת (לדוגמה לצורך הדפסה). הייצוג צריך להיות מילה חוקית בשפה - כך שהפעלה של parse על התוצאה של format_term תחזיר term זהה ל term המקורי.

בקובץ reducer.ml נבנה interpreter עבור λ -calculus בשלבים, בשאלות הבאות. בקובץ זה נשתמש במודול StringSet מהקובץ utils.ml כדי לייצג קבוצות של מחרוזות. המודול מכיל פונקציות עבור פעולות נפוצות על קבוצות (איחוד, הוספת איבר, הוצאת איבר, וכו'), והתיעוד שלו זמין ב: <http://caml.inria.fr/pub/docs/manual-ocaml/libref/Set.S.html>. הקובץ utils.ml גם מכיל פונקציה נוחה להדפסת קבוצות של מחרוזות.

2. הוסיפו לקובץ reducer.ml את הפונקציה:

`fv : term -> StringSet.t`

שמקבלת `term` ומחזירה את קבוצת המשתנים החופשיים בו. את הקבוצה יש לייצג באמצעות המודול `StringSet` (שמגיע מ `utils.ml`). כזכור, את קבוצת המשתנים החופשיים ניתן להגדיר באופן אינדוקטיבי כך:

$$FV(x) = \{x\}$$

$$FV(\lambda x. t) = FV(t) \setminus \{x\}$$

$$FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$$

3. לצורך מימוש `alpha-renaming`, אנו זקוקים לפונקציה שתחזיר שם של משתנה חדש שאינו בקבוצה של משתנים בשימוש. לצורך כך הקובץ `reducer.ml` מכיל את הערך `possible_variables : string list`, שמכיל רשימה של שמות משתנים אפשריים. הוסיפו לקובץ `reducer.ml` את הפונקציה:

`fresh_var : StringSet.t -> string`

הפונקציה צריכה לקבל קבוצה של שמות משתנים בשימוש, ולהחזיר שם חדש מתוך הרשימה `possible_variables`. במידה וכל השמות ברשימה בשימוש, הפונקציה צריכה לזרוק `OutOfVariablesError`.

4. הוסיפו לקובץ `reducer.ml` את הפונקציה:

`substitute : string -> term -> term -> term`

על פונקציה זו לממש החלפה (`substitution`) כולל `alpha-renaming` במקרה הצורך. סדר הפרמטרים הוא כזה שהביטוי `t1 t2 "x"` `substitute` יחזיר את:

`[x ↦ t1] t2`

כלומר `t2` כאשר כל המופעים של המשתנה `x` הוחלפו ב `t1` (ולא להיפך!).

הפונקציה צריכה לבצע את ההחלפה בכל מקרה, תוך שהיא מבצעת `alpha-renaming` במקרה הצורך. את פעולת הפונקציה ניתן להגדיר אינדוקטיבית באופן הבא:

`[x ↦ s] x = s`

`[x ↦ s] y = y` if `y ≠ x`

`[x ↦ s] (t1 t2) = ([x ↦ s] t1) ([x ↦ s] t2)`

`[x ↦ s] (λx. t) = λx. t`

`[x ↦ s] (λy. t) = λy. ([x ↦ s] t)` if `y ≠ x` and `y ∉ FV(s)`

`[x ↦ s] (λy. t) = λz. ([x ↦ s] ([y ↦ z] t))` if `y ≠ x` and `y ∈ FV(s)`,
when `z ∉ FV(s) ∪ FV(t) ∪ {x}`

כאשר המקרה האחרון מבצע `alpha-renaming` (השתכנעו שזוהי אכן הגדרה נכונה להחלפה).

5. הוסיפו לקובץ reducer.ml את הפונקציה:

`reduce_strict : term -> term option`

על פונקציה זו לממש צעד אחד של חישוב (reduction) לפי סמנטיקת call-by-value. הפונקציה מחזירה ערך מטיפוס term option, כיוון שלא על כל term ניתן לבצע reduction. משמעות ערך החזרה היא:

$(\text{reduce_strict } t) = \text{Some } t'$ if $t \Rightarrow t'$ in call-by-value

$(\text{reduce_strict } t) = \text{None}$ if $t \not\Rightarrow$, i.e. t is not reducible in call-by-value

הפונקציה צריכה לממש את הכללים שנלמדו בשיעור ובתרגול, כאשר הערכים היחידים הם abstractions.

6. הוסיפו לקובץ reducer.ml את הפונקציה:

`reduce_lazy : term -> term option`

על פונקציה זו לממש צעד אחד של חישוב (reduction) לפי סמנטיקת lazy-evaluation. הפונקציה מחזירה ערך מטיפוס term option, ומשמעות ערך החזרה לפי ההסבר בשאלה 5, הפעם עבור סמנטיקת lazy-evaluation. הפעם עבור סמנטיקת lazy-evaluation מוגדרת ע"י הכללים הבאים:

$$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2} \quad (\text{E-APP1})$$

$$(\lambda x. t_{12}) t_2 \rightarrow [x \mapsto t_2] t_{12} \quad (\text{E-APPABS})$$

7. הוסיפו לקובץ reducer.ml את הפונקציה:

`reduce_normal : term -> term option`

על פונקציה זו לממש צעד אחד של חישוב (reduction) לפי סמנטיקת normal-order. הפונקציה מחזירה ערך מטיפוס term option, ומשמעות ערך החזרה לפי ההסבר בשאלה 5, הפעם עבור סמנטיקת normal-order. סמנטיקת normal-order מוגדרת ע"י הכללים הבאים לפי הסדר, כלומר הכלל העליון ביותר מופעל במקרה בו ניתן להפעיל יותר מכלל אחד (בניסוח אחר, תמיד בוחרים את ה-redex החיצוני ביותר והשמאלי ביותר):

$$(\lambda x. t_{12}) t_2 \rightarrow [x \mapsto t_2] t_{12} \quad (\text{E-APPABS})$$

$$\frac{t_1 \rightarrow t'_1}{\lambda x. t_1 \rightarrow \lambda x. t'_1} \quad (\text{E-ABS})$$

$$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2} \quad (\text{E-APP1})$$

$$\frac{t_2 \rightarrow t'_2}{t_1 t_2 \rightarrow t_1 t'_2} \quad (\text{E-APP2})$$

8. הקובץ tests.ml מכיל את הפונקציה הבאה:

```
evaluate : verbose:bool -> (term -> term option) -> term -> term
```

פונקציה זו מקבלת את אחת הפונקציות שמימשת בשאלות 5-7, ומקבלת term, ומחשבת אותו איטרטיבית עד לצורה שהיא irreducible תוך שימוש בפונקציה הנתונה. אם הפרמטר verbose הוא true, הפונקציה גם מדפיסה את תהליך החישוב. הקובץ tests.ml גם מכיל קלטי בדיקה ראשוניים והרצות בדיקה בסמנטיקות השונות. הרחיבו את הקובץ כדי שכלול בדיקות נוספות, והשתמשו בו במהלך הפיתוח של כל השאלות הקודמות כדי לבדוק את המימוש.

9. בנוס א'

הוסיפו לקובץ parser.ml את הפונקציות הבאות:

```
parse_term_conv : token list -> term * token list  
parse_conv : string -> term
```

הפונקציות צריכות לתפקד כמו הפונקציות משאלה 1, אבל לתמוך במוסכמות התחביריות (syntax conventions) שהגדרנו בכיתה, כלומר לאפשר להשמיט סוגריים, כך שלדוגמה הביטוי הבא יעבור parsing בהצלחה:

```
\x. \y. \z. x y z (* parses like: (\x (\y. (\z. ((x y) z)))) *)
```

10. בנוס ב'

הוסיפו לקובץ parser.ml את הפונקציה:

```
format_term_conv : term -> string
```

שתפעל בדומה ל format_term משאלה 1, אבל לא תדפיס סוגריים מיותרים, כשהיא לוקחת בחשבון את המוסכמות התחביריות שהגדרנו בכיתה.

בהצלחה!