בס"ד

שפות תכנות – תרגיל 1

תאריך הגשה: 20.11.2022

הוראות הגשה: ההגשה בזוגות דרך מערכת הסאבמיט. כל זוג נדרש לחשוב, לפתור ולכתוב את התרגיל בעצמו. יש לקרוא הוראות אלא בקפידה. הגשה שלא על פי הוראות אלה תוביל להורדת ניקוד.

בתרגילי תכנות יש להגיש קובץ ml. עם השם של הפונקציה שאותה יש לממש (לדוגמא עבור הפונקציה insert_at שאתם צריכים לממש הקובץ שבו נמצאת הפונקציה יקרא insert_at.ml). יש להגיש קובץ עבור כל פונקציה שנדרשתם לכתוב גם אם הם באותה שאלה בסעיפים שונים. במידה ויש הגדרות של טיפוסים בשאלה צריך להוסיף אותם לקובץ, אין להדפיס למסך שום דבר שאינו נדרש בתרגיל עצמו. מותר ואף מומלץ להשתמש בפונקציות עזר שהפונקציה שאתם כותבים תקרא להם (פונקציות עזר יהיו באותו קובץ של הפונקציה הראשית).

בחלק ב כל ההוכחות באינדוקציה יהיו באינדוקציה מבנית בלבד. הוכחות באינדוקציה מתמטית לא יתקבלו. ניתן לייצג את הדקדוקים ואת הכללים בהוכחות לפי איזה דרך שתרצו שנלמדה בכיתה (כללי גזירה, BNF, כללי יצירה).

חומר עזר מומלץ: כדאי להבין היטב את הרצאות ותרגולים מספר 2-1. קישור לתרגולים (המצגות נמצאות בתיאור הסרטון):
https://www.youtube.com/watch?v=1kRdE90lB34&list=PLaMkJ2Pfx92I7DbMteYLYmMDyn3N0dDIT

התקנת סביבת עבודה בפעם הראשונה, מומלץ לעקוב אחרי ההנחיות שבתרגול הראשון (למידע נוסף ניתן להסתכל ב: https://dev.realworldocaml.org/install.html).

מודול של רשימות: https://v2.ocaml.org/api/List.html

קומפיילר של ocaml אונליין: https://www.jdoodle.com/compile-ocaml-online/

סקריפט בדיקה: מצורף לתרגיל הסקריפט (שימו לב שהסקריפט עובד רק בסביבת עבודה של linux/mac, בווידוס ניתן להתקין wsl) שאיתו אני אבדוק את החלק התכנותי של התרגיל. מומלץ מאוד להשתמש בבדיקות שיש שם (כמובן שאני אריץ גם עם בדיקות נוספות). כדי להריץ את הסקריפט יש לשים אותו ואת תיקיית test בתיקייה של קבצי הקוד שלכם ולהריץ את הפקודה

```
bash test.sh <some argument>
```

הפלט של הסקריפט הוא כמה שגיאות יש בכל טסט. אם כתוב 0 כל הכבוד הצלחתם. במידה ולא כתוב 0, הסקריפט יראה את השורה שבה הייתה שגיאה (הראשון זה מה שיש בפלט הטסט האמיתי והשני זה מה שהיה אצלכם). לשם הנוחות תוכלו לראות את התוצאות של הקוד שלכם על הטסטים בתיקיית results שיצר הסקריפט. כדי לדעת בכל טסט מה הבדיקה שבוצעה יש להסתכל בנספח לתרגיל הזה שמופיע בעמוד האחרון של קובץ זה.

הארגומנט לסקריפט הוא השם של הפונקציה שאתם רוצים לבדוק. נניח ואתם רוצים לבדוק את הקובץ table.ml תריצו את הסקריפט עם הארגומנט table:

```
bash test.sh table
```

אם תרצו לבדוק את כל הפונקציות שלכם בפעם אחת ניתן להריץ את הסקריפט עם הארגומנט all.

```
bash test.sh all
```

בס"ד

<u>מה להגיש</u>: רשימה של הקבצים שיש להגיש:

Insert_at.ml

insert_none_at.ml

add_to_search_tree.ml

table_two.ml

table.ml

ex1.pdf – עם הפתרונות של החלק התיאורטי

id.txt – קובץ עם השם משתמש בסבמיט ות.ז של כל אחד מהמגישים (כל מגיש בשורה בנפרד)

כל הקבצים צריכים להיות בקובץ zip אחד בשם: ex1.zip.

בס"ד

- **<u>חלק א: Ocaml</u>**

1. <u>חימום - הכנסת ערך לרשימה במיקום מסויים:</u>

א. כתבו פונקציה בשם insert_at המקבלת איבר אינדקס ורשימה ומכניסה את האיבר לרשימה באינדקס הנתון.

דוגמאות להרצה:

```
─( 13:16:48 )─< command 0 >──────────────────────────────────────────────────
utop # <--please insert your function here-->
─( 13:16:48 )─< command 1 >──────────────────────────────────────────────────
utop # insert_at "e" 1 ["a";"b";"c";"d"];;
- : string list = ["a"; "e"; "b"; "c"; "d"]
─( 13:16:57 )─< command 2 >──────────────────────────────────────────────────
utop # insert_at "e" 3 ["a";"b";"c";"d"];;
- : string list = ["a"; "b"; "c"; "e"; "d"]
─( 13:17:23 )─< command 3 >──────────────────────────────────────────────────
utop # insert_at "e" 4 ["a";"b";"c";"d"];;
- : string list = ["a"; "b"; "c"; "d"; "e"]
─( 13:17:29 )─< command 4 >──────────────────────────────────────────────────
utop # insert_at "e" 0 ["a";"b";"c";"d"];;
- : string list = ["e"; "a"; "b"; "c"; "d"]
```

<u>הנחת קלט</u>: אנחנו מניחים שברשימה l יש לפחות n איברים, כאשר n הוא האינדקס הנבחר להכנסה.

ב. במפעל מסוים רצו ליצור פונקציה שמכניסה אחרי None במקומות מסוימים ברשימות שלהם. כתבו פונקציה בשם insert_none_at שתקבל אינדקס להכנסה ורשימה ותכניס את הערך "None" ברשימה באינדקס שהתקבל. שימו לב הפונקציה החדשה צריכה לקרוא לפונקציה מסעיף א.
דוגמאות להרצה:

```
─( 13:16:48 )─< command 0 >──────────────────────────────────────────────────
utop # <--please insert your function here-->
─( 14:01:10 )─< command 2 >──────────────────────────────────────────────────
utop # insert_none_at 2 ["a";"b";"c";"d";"e"];;
- : string list = ["a"; "b"; "None"; "c"; "d"; "e"]
─( 14:01:14 )─< command 3 >──────────────────────────────────────────────────
utop # insert_none_at 0 ["Not None";"Not None";"Not None"];;
- : string list = ["None"; "Not None"; "Not None"; "Not None"]
```

<u>הנחת קלט</u>: אנחנו מניחים שברשימה l יש לפחות n איברים, כאשר n הוא האינדקס הנבחר להכנסה.

2. <u>עץ חיפוש בינארי</u>

תזכורת: בתרגול ראינו שאפשר להגדיר עץ בינארי בצורה הבאה:

```
# type 'a binary_tree =
      | Empty
      | Node of 'a * 'a binary_tree * 'a binary_tree;;
type 'a binary_tree = Empty | Node of 'a * 'a binary_tree * 'a binary_tree
```

א. כתבו פונקציה add_to_search_tree המקבלת עץ חיפוש בינארי של מספרים שלמים ומספר שלם, ומחזירה עץ חיפוש חדש שמתקבל מהוספת המספר לעץ החיפוש הבינארי שהתקבל בקלט. עץ חיפוש בינארי הוא עץ בינארי כך שכל הערכים הקטנים מהערך של הקודקוד נמצאים בתת העץ השמאלי של הקודקוד וכל הקודקודים בעלי ערך גבוה יותר נמצאים בתת העץ הימני של הקודקוד.

דוגמא להרצה:

```
─( 14:05:42 )─< command 0 >──────────────────────────────────────────────────
utop # type 'a binary_tree =
      | Empty
      | Node of 'a * 'a binary_tree * 'a binary_tree;;
type 'a binary_tree = Empty | Node of 'a * 'a binary_tree * 'a binary_tree
─( 14:05:42 )─< command 1 >──────────────────────────────────────────────────
utop # <--please insert your function here-->
```

```
─( 10:58:03 )─< command 3 >──────────────────────────────{ counter: 0 }─
utop # let tree = add_to_search_tree Empty 5;;
val tree : int binary_tree = Node (5, Empty, Empty)
─( 10:58:47 )─< command 4 >──────────────────────────────{ counter: 0 }─
utop # let tree = add_to_search_tree tree 3;;
val tree : int binary_tree = Node (5, Node (3, Empty, Empty), Empty)
─( 10:59:27 )─< command 5 >──────────────────────────────{ counter: 0 }─
utop # let tree = add_to_search_tree tree 6;;
val tree : int binary_tree =
  Node (5, Node (3, Empty, Empty), Node (6, Empty, Empty))
─( 10:59:37 )─< command 6 >──────────────────────────────{ counter: 0 }─
utop # let tree = add_to_search_tree tree 21;;
val tree : int binary_tree =
  Node (5, Node (3, Empty, Empty), Node (6, Empty, Node (21, Empty, Empty)))
─( 10:59:40 )─< command 7 >──────────────────────────────{ counter: 0 }─
utop # let tree = add_to_search_tree tree 1;;
val tree : int binary_tree =
  Node (5, Node (3, Node (1, Empty, Empty), Empty),
   Node (6, Empty, Node (21, Empty, Empty)))
```

<u>הנחה על הקלט</u>: - בעץ החיפוש הבינארי לא יהיו שני מספרים בעלי ערך זהה בכל קריאה לפונקציה.

- המספר שמתקבל לפונקציה לא יהיה שווה לאחד מאיברי העץ.

ב. מה הייתם משנים בפונקציה שכתבתם בסעיף א כדי שתתמוך גם בעץ חיפוש בינארי שכל איבריו הם מספרים של נקודה צפה (float).
<u>הערה</u>: תשובה לסעיף ב צריכה להתוסף לקובץ התשובות pdf .

3. <u>לוגיקה וטבלאות אמת</u>

נגדיר את ה-variant type הבא שיעזור לנו לייצג ביטויים לוגיים עם משתנים בוליאניים:

```
type bool_expr =
  |   Var of string
  |   Not of bool_expr
  |   And of bool_expr * bool_expr
  | Or of   bool_expr * bool_expr;;
```

ניתן לייצג ביטויים לוגים בעזרת ה-type החדש. לדוגמא הביטוי הלוגי $(a \vee b) \wedge (a \wedge b)$, ייוצג באופן הבא:

```
# And(Or(Var "a", Var "b"), And(Var "a", Var "b"));;
- : bool_expr = And (Or (Var "a", Var "b"), And (Var "a", Var "b"))
```

א. הגדירו פונקציה table_two שבהינתן שני משתנים בוליאניים וביטוי לוגי (ללא משתנים נוספים חוץ מהשניים שהוגדרו לפני) מחזירה את טבלת האמת של הביטוי הלוגי. טבלת האמת תודפס למסך כרשימה של tuple שבכל איבר יש שלוש ערכים בוליאניים. הראשון הוא הערך של a, השני של b והשלישי הוא הערך של הביטוי הלוגי בהינתן הערכים של a ו-b.

דוגמא להרצה:

```
-( 23:00:24 )-< command 0 >-------------------------------------------------
utop # type bool_expr =
    | Var of string
    | Not of bool_expr
    | And of bool_expr * bool_expr
    | Or of bool_expr * bool_expr;;
type bool_expr =
    Var of string
  | Not of bool_expr
  | And of bool_expr * bool_expr
  | Or of bool_expr * bool_expr
-( 23:00:24 )-< command 1 >-------------------------------------------------
utop # <--please insert your function here-->
-( 23:01:35 )-< command 3 >-------------------------------------------------
utop # table_two "a" "b" (And(Var "a", Or(Var "a", Var "b")));;
- : (bool * bool * bool) list =
[(true, true, true); (true, false, true); (false, true, false);
 (false, false, false)]
```

נשים לב שיש ברשימה ארבעה איברים. לשם האחידות נגדיר את הסדר של האיברים כך:

a = true, b = true

a = true, b = false

a = false, b = true

a = false, b = false

זה הסדר שמופיע בדוגמא להרצה.

ב. הגדירו פונקציה table שתעשה טבלת אמת כמו סעיף א אך הפעם היא יכולה לקבל מספר שונה של משתנים בכל פעם. המשתנים יועברו לפונקציה כרשימה של משתנים. הפעם כל איבר ברשימה המייצגת טבלת אמת יהיה tuple עם שני איברים, האיבר הראשון יהיה רשימה שכל איבר בו הוא tuple של שם המשתנה והערך שניתן לו (true/false) והערך השני של ה-tuple הוא הערך שך הביטוי הלוגי.

שתי דוגמאות להרצה:

```
-( 23:00:24 )-< command 0 >-------------------------------------------------------
utop # type bool_expr =
    | Var of string
    | Not of bool_expr
    | And of bool_expr * bool_expr
    | Or of bool_expr * bool_expr;;
type bool_expr =
    Var of string
  | Not of bool_expr
  | And of bool_expr * bool_expr
  | Or of bool_expr * bool_expr
-( 23:00:24 )-< command 1 >-------------------------------------------------------
utop # <--please insert your function here-->
-( 23:11:01 )-< command 7 >-------------------------------------{ counter: 0 }
utop # table ["a"; "b"] (And(Var "a", Or(Var "a", Var "b")));;
- : ((string * bool) list * bool) list =
[([("a", true); ("b", true)], true); ([("a", true); ("b", false)], true);
 ([("a", false); ("b", true)], false); ([("a", false); ("b", false)], false)]
-( 23:11:07 )-< command 8 >-------------------------------------{ counter: 0 }
utop # let a = Var "a" and b = Var "b" and c = Var "c" in
  table ["a"; "b"; "c"] (Or(And(a, Or(b,c)), Or(And(a,b), And(a,c))));;
- : ((string * bool) list * bool) list =
[([("a", true); ("b", true); ("c", true)], true);
 ([("a", true); ("b", true); ("c", false)], true);
 ([("a", true); ("b", false); ("c", true)], true);
 ([("a", true); ("b", false); ("c", false)], false);
 ([("a", false); ("b", true); ("c", true)], false);
 ([("a", false); ("b", true); ("c", false)], false);
 ([("a", false); ("b", false); ("c", true)], false);
 ([("a", false); ("b", false); ("c", false)], false)]
```

סדר האיברים יהיה עבור האיבר הראשון יופיע כל האפשרויות שלו ל-true ולאחר מכן של false. עבור שאר המשתנים חוץ מהראשון הסדר יהיה הופעה של כל האיברים שבהם האיבר הראשון (מתוך ה-n-1 איברים הוא true ואחר כך כל האיברים בהם הוא false כך יהיה סדר הטבלה באופן רקורסיבי (כמו שמופיע בדוגמא להרצה).

בס"ד

- **<u>חלק ב: אינדוקציה מבנית</u>**

<u>שאלה 1:</u>

א. להלן דקדוק חסר הקשר:

E1 ::= ε | id | (E1)

הסימן ε מייצג את המחרוזת הריקה, ו-id היא מחרוזת שמייצגת מספר זהות כלשהו.
הוכיחו שבכל ביטוי בשפה של E1, מספר הסוגריים הפותחים "(" שווה למספר הסוגריים הסוגרים ")".

ב. להלן דקדוק חסר הקשר:

E2 ::= ε | id | (R
R ::=) | E2)

הוכיחו שבכל ביטוי בשפה של E2, מספר הסוגריים הפותחים "(" שווה למספר הסוגריים הסוגרים ")".

ג. הוכיחו ששני הדקדוקים שבסעיפים א ו-ב מגדירים את אותה שפה. כלומר L(E1)=L(E2).

<u>שאלה 2:</u>

להלן דקדוק חסר הקשר לאובייקט משפת JSON (string ו-int מקבלים את המשמעות המקובלת שלהם, כלומר מחרוזות ומספרים):

obj ::= {} | {member}

member ::= keyvalue | member, member

keyvalue ::= string : value

value ::= string | int | obj

א. האם המילים הבאות נמצאות בשפת הדקדוק הנ"ל? אם כן, הראו עץ גזירה שמוכיח זאת. אם לא, הוכיחו שאין עץ גזירה כזה:
   a. {“course” : “concept in PL”, “ex” : 1, “grade” : 100}
   b. {“course” : “concept in PL”, “ex” : 1, “grade” : {100}}

ב. הראו שהדקדוק הוא רב משמעי ע"י מציאת שני עצי גזירה שונים לאותה מילה.

בס"ד

<u>שאלה 3:</u>

<u>תזכורת</u>: בחלק א ראינו את הטיפוס הבא:

```
type bool_expr =
  |   Var of string
  |   Not of bool_expr
  |   And of bool_expr * bool_expr
  | Or of   bool_expr * bool_expr;;
```

נכתוב את הפונקציות הבאות:

```
let rec num_of_vars = fun exp -> match exp with
                        Var(_) -> 1
                      | And(x,y) -> (num_of_vars x) + (num_of_vars y)
                      | Or (x,y) -> (num_of_vars x) + (num_of_vars y)
                      | Not(x) -> (num_of_vars x);;


let rec num_of_connectives = fun exp -> match exp with
                        Var(_) -> 0
                      | And(x,y) -> (num_of_connectives x) + (num_of_connectives y) + 1
                      | Or(x,y) -> (num_of_connectives x) + (num_of_connectives y) + 1
                      | Not(x) -> (num_of_connectives x) + 1;;
```

א. הוכיחו באינדוקציה מבנית או הפריכו באמצעות דוגמא נגדית: לכל ביטוי exp מטיפוס bool_expr מתקיים:

$$num_of_vars(exp) = num_of_connectives(exp) + 1$$

ב. הוכיחו באינדוקציה מבנית או הפריכו באמצעות דוגמא נגדית: לכל ביטוי exp מטיפוס bool_expr בו לא מופיע Not מתקיים:

$$num_of_vars(exp) = num_of_connectives(exp) + 1$$

<u>שאלה 4 (רשות, בונוס 10 נקודות):</u>

נרצה להגדיר דקדוק עבור מולטי קבוצות סופיות מעל המספרים הטבעיים. כזכור, במולטי קבוצה יש חשיבות למספר הפעמים שאיבר מופיע אבל לא לסדר. למשל הביטוי {} מייצג את הקבוצה הריקה, {5,2,2} מייצג את הקבוצה שמכילה את המספר 5 ופעמיים את המספר 2, והקבוצה {1 ,{2,3} ,4} מייצגת את הקבוצה שמכילה את 1 ו-4 וקבוצה שמכילה את 2 ו-3. שימו לב כי {} היא הקבוצה הריקה, ולעומת זאת {{}} היא הקבוצה שמכילה את הקבוצה הריקה.

א. כתבו דקדוק חסר הקשר שגוזר משתנה בשם H לקבוצת הביטויים הנ"ל. מומלץ להשתמש במשתנה נוסף.

ב. כתבו הגדרה של הקבוצה H ע"י הגדרה אינדוקטיבית בעזרת כללי היסק.

ג. הגדירו בהגדרה אינדוקטיבית פונקציה שסוכמת את כל האיברים שמופיעים במולטי-קבוצה בסדר כלשהו. למשל:

sum({1, {2, 2}, {{3}}}) = 1 + 2 + 2 + 3 = 8

sum({{}, {{}}}) = 0

ד. הגדירו בהגדרה אינדוקטיבית פונקציה שסופרת את מספר האיברים במולטי-קבוצה בקינון כלשהו. למשל:

sum({1, {2, 2}, {{3}}}) = 4

sum({{}, {{}}}) = 0

ה. הגדירו בהגדרה אינדוקטיבית בעזרת כללי היסק את קבוצת המולטי-קבוצות כנ"ל שבהן לא מופיעים מספרים טבעיים פרט ל-1, לקבוצה זו נקרא בשם J.

ו. הוכיח באינדוקציה שלכל קבוצה a ב-J מתקיים: count(a) = sum(a).

נספח – בדיקות לקטעי קוד:

insert_at_test1: insert_at ”5” 2 [”0”;”1”;”2”;”3”];;

insert_at_test2: insert_at ”Ocaml” 0 [”Java”; ”C”;];;

insert_at_test3: insert_at ”Empty List” 0 [];;

insert_none_at_test1: insert_none_at 0 [”0”;”1”;”2”;”3”];;

insert_none_at_test2: insert_none_at 0 [];;

add_to_search_tree_test1_test1: add_to_search_tree Empty 5;;

add_to_search_tree_test1_test2: add_to_search_tree (Node(5, Empty, Empty)) 1;

add_to_search_tree_test1_test3: add_to_search_tree (Node(5, Node(1, Empty, Empty), Empty)) 6;;

table_two_test1: table_two ”a” ”b” (And(Var ”a”, Or(Var ”a”, Var ”b”)));;

table_two_test2: table_two ”x” ”y” (Or(And(Var ”y”, Var ”y”),(And(Var ”x”, Var ”x”)));;

table_two_test3: table_two ”@” ”$” (Or(And(Var ”@”, Not(Var ”$”)), And(Var ”$”, Not(Var ”@”))));;

table_test1: table [”a”; ”b”; ”c”] (Or(And(Var ”a”, Or(Var ”b”, Var ”c”)), Or(And(Var ”a”, Var ”b”), And(Var ”a”, Var ”c”))));;

table_test2: table [”a”; ”b”] (And(Var ”a”, Or(Var ”a”, Var ”b”)));;