שפות תכנות 236319

תרגיל בית 5 – אביב תשפ"ג

חלק יבש

שם סטודנט	ת.ז
עדן סרחאן	324849256
מורן עאמר	322730789

שאלה 1

המתרגלים בקורס שפות תכנות רצו לכתוב שאלות למבחן בקורס, אך מרוב שהשקיעו בשאלות שיעורי הבית נגמרו להם הרעיונות הטובים לשאלות.

הציעו 2 שאלות בְּרֵרָה למבחן (שאלות אמריקאית) על אודות חומר הלימוד שהוצג בהרצאות או בשקפים. על כל שאלת בררה להציע בדיוק 4 תשובות אפשריות, שמתוכן על הנבחן לברור תשובה אחת, הנכונה ביותר. לכל שאלה ציינו מה התשובה הנכונה ונמקו.

שאלה 1:

מה תדפיס הפונקציה הבאה ב-SML:

```
fun fofo a b c = fn d => (a + b  ; c^d; fn e => b);
```

בחר אחת התשובות הבאות:

A. val fofo = fn: ∀ 'a 'b . int → int → 'a → 'a → 'b → int;
B. val fofo = fn: int → int → string → string → int → int;
C. val fofo = fn: ∀ 'a . int → int → string → string → 'a → int;
D. val fofo = fn: ∀ 'a . int → int → char → char → 'a → int;

התשובה הנכונה היא C.

שאלה 2:

מה מייחד את שפת SML משפות אחרות בתחום התכנות הפונקציונלי?

א) SML מספקת תמיכה מובנית בשימוש בלולאות ותנאים לביצוע קוד מחדש ומורכב

ב) SML מציעה אפשרות לשימוש באובייקטים ומחלקות כחלק מהתכנות הפונקציונלי.

ג) SML מתירה את השימוש בטיפוסים משלמים ושברים באותו הקוד .

ד) SML מאפשרת הגדרת פונקציות רקורסיביות ושימוש בסיבובי עץ ביטויים (pattern matching).

ה) SML מספקת כלים מתקדמים לניתוח ובדיקת טיפוסים בשלב הקומפילציה.

תשובה נכונה: ד SML מאפשרת הגדרת פונקציות רקורסיביות ושימוש בסיבובי עץ ביטויים.

שאלה 2

המתרגל האחראי בקורס שפות תכנות קצת שכח איך לעבוד עם מספרים בפרולוג (מבוסס על סיפור אמיתי). בשאלה זו נעזור לו לחדד את ההבדלים בין אופרטורים שונים בשפה עבור מספרים.

ענו על כל אחד מהסעיפים והסבירו בקצרה:

1. בתרגול ראינו את האופרטורים =, is ו- =:=.
   1. איזה אופרטור מבין אלו אינו סימטרי (כלומר, לא ניתן להחליף את סדר הארגומנטים ולקבל תוצאה זהה)?

אופרטור 'is' אינו סימטרי, הוא אופרטור שעושה שערוך לצד ימין ואז עושה השמה לתוצאת השערוך לתוך צד שמאל אם המשתנה בצד שמאל אינו קשור, אחרת עושה השוואה.
נביט בקטעי הקוד הבאים להבנה יותר עמוקה:

```
?- X is 2 + 1, Y = 1 + 2, X is Y.

X = 3,

Y = 1+2.
```

כי קודם הוא עשה שערוך ל-X ושם בתוכן 3, ואז עשה שערוך ל-Y וקיבל 3, ואז השווה ביניהם ולכן התוצאה הייתה true כאשר X ו-Y הם אלה שנתן, מצד שני:

```
?- X is 2 + 1, Y = 1 + 2, Y is X.

false.
```

כי קודם הוא עשה שערוך ל-X וקיבל 3, ואז השווה בין Y שהוא (1, 2)+ והשווה בין 3 ו-(1,2)+ שהם לא שווים לכן החזיר false.

```
?- X = 2 + 1, Y = 1 + 2, X is Y.

false.
```

כי קודם הוא עשה שערוך ל-Y וקיבל 3, ואז השווה בין X שהוא (2, 1)+ והשווה בין 3 ו-(2,1)+ שהם לא שווים לכן החזיר false.

2. תנו דוגמה לשימוש ב- =:= בה לא ניתן להשתמש באופרטור = במקום.

אם נרצה להבדיל בין הטיפוסים השונים (כי ב-prolog אין טיפוסים):

```
X = 3, Y = 3.0, X = Y. -> false
X = 3, Y = 3.0, X =:=Y -> true
```

במקרה זה אופרטור = מבדיל בין שני הטיפוסים כי הם structurally שונים, לעומת =:=.

3. תנו דוגמה לשימוש ב- = בה לא ניתן להשתמש באופרטור =:= במקום.

אם נרצה להשוות ערכים:

```
X = 1 + 2, Y = 3 + 0, X =:= Y. -> true.
X = 1 + 2, Y = 3 + 0, X = Y. -> false.
```

במקרה זה אופרטור =:= משערך קודם את שני האגפים ואז מבצע השוואה ביניהם, בניגוד ל =.

2. בתרגול ראינו את האופרטור =# מהספרייה .CLPFD עבור אילו מקרים אופרטור זה הוא הכללה של שלושת האופרטורים מהסעיף הקודם?
(כלומר, תחת אילו תנאים ניתן להחליף כל אחד מהקודמים באופרטור זה)

A. אופרטור =:= דורש ששני האגפים יהיו instainted , במקרה שזה מתקיים ניתן להשתמש ב =:= ו =# בצורה דומה, דוגמה:

'X is 2, Y is 2, X #= Y.' ≡ 'X is 2, Y is 2, X =:= Y.'

'X is 2, X #= Y. ' returns X = 2, Y = 2 ... whereas 'X is 2, X =:= Y.' returns an error because Y is not instainted.

B. אופרטור is משערך רק את צד ימין לעומת אופרטור =# שמשערך את שני הצדדים, לכן אם אגף שמאל של ה"שאלה" לא ניתן לשערך אותו עוד יותר (אטום) ניתן להשתמש ב-is ו-=# בצורה דומה, דוגמה:

'X = 2, Y = 1+1, Y is X.' return false because Y which is +(1,1) isn't evaluated and +(1,1) isn't equal to 2, whereas 'X = 2, Y = 1+1, Y is X.' is true and returns X=2, Y= 1+1, because #= evaluates both sides.

'X =2, Y = 1+1, X is Y.' ≡ 'X =2, Y = 1+1, X #= Y.' return X=2, Y=1+1, which is true in both cases because in this case Y is evaluated and X cannot be further evaluated.

C. אופרטור = לא משערך אף אחד משני האגפים לעומת =#, אז אם שני האגפים לא ניתנים לשערוך עוד יותר ניתן להשתמש ב-= ו-=# בצורה דומה, דוגמה:

'X = 1 + 1, Y = 2 + 0, X = Y.' returns false whereas 'X = 1 + 1, Y = 2 + 0, X #= Y.' is true and returns X = 1+1, Y = 2+0, because #= evaluates both sides whereas = doesn't.

'X =2, Y =2, X = Y.' ≡ 'X =2, Y =2, X #= Y' both are true and return X=2, Y =2, because '2' cannot be evaluated further unlike '2+0' which is basically translated to '+(2,0)'.

שאלה 3

בתרגול למדנו על שפת התכנות TypeScript כדוגמה לשפה עם טיפוסיות הדרגתית (Gradual Typing).

1. ראינו שב-TypeScript ניתן להגדיר טיפוס Record באופן הבא:

```
type Record = { x: number, y: number };
```

האם תאימות בין טיפוסים כאלו הוא Nominal או Structural? מה לגבי תאימות בין טיפוסים המוגדרים על ידי מחלקות? הסבירו והביאו דוגמאות קוד התומכות בטענה שלכם.

פתרון:

ב- TypeScript התאמת טיפוסים (type matching) היא מבנה (structural) ולא שמו (nominal). זאת אומרת שהתאמה נעשית לפי מבנה הטיפוס ולא לפי שם הטיפוס עצמו.

כאשר מגדירים טיפוס כמו ' **Record = { x: number, y: number }** ' , מדובר בהגדרת טיפוס מבני (literal type), או במילים אחרות, טיפוס שהתאמתו תלויה במבנה הטיפוס עצמו ולא בשמו.

בנוסף, התאמת טיפוסים בין מחלקות ב TypeScript-גם היא מבנה (structural) ולא שמו (nominal) זאת אומרת לפי מה שהסברנו לעיל ההתאמה גם כן נעשית לפי המבנה של המחלקה ולא לפי שמה.

<u>דוגמאות:</u>

1) תאימות בין טיפוסים מבניים:

```
type Point2D= {x:number , y:number};
type Point3D={x:number , y:number , z:number};
let pont2D: Point2d = {x:1,y:2};
let pont3D: Point3D = {x:1,y:2,z:3};

point2D=point3D;    #ניתן לעשות התאמה זו מכיוון ששני הטיפוסים ממבנה טיפוסי דומה
point3D=point2D;    #ניתן לעשות התאמה זו , ואז הטיפוס השלישי יימחק
```

2) תאימות בין מחלקות:

```
class Animal {
name: string;

constructor(name: string) {

    this.name=name;

  }
```

```
}

class Dog {
   name:string;
   constructor(name: string) {
        This.name=name;
   }
bark() {
       Console.log('Woof!');
}
}
Let animal: Animal=new Animal('Animal');
Let dog: Dog=new Dog('Dog');

animal=dog;  #התאמה תקינה כיוון שמבנה המחלקה דומה
dog=animal;   #התאמה תקינה, הפעולות הספציפיות למחלקת דוג לא יהיו זמינות.
```

2. בתרגול ראינו שמערכת הטיפוסים של השפה אינה בטוחה (safe), מכיוון שניתן להשתמש ב-any כדי לבטל בדיקות טיפוסים והטיפוסים אינם נבדקים בזמן ריצה. בנוסף לכך, מערכת הטיפוסים של השפה אינה בטוחה גם ללא שימוש ב-any. הביאו דוגמת קוד בה **לכל** הערכים יש טיפוס **סטטי קונקרטי** מוגדר, אך בזמן ריצה עדיין תתרחש שגיאת טיפוס.

הערות:

- תוכלו להשתמש ב-TypeScript Playground כדי לנסות את הדוגמאות שלכם.
- אין להשתמש ב- type assertions (המקביל ב-TypeScript להמרות טיפוסים).

רמזים לסעיף 2:

a. בהינתן טיפוס B היורש מ-A, הקומפיילר מתייחס למערך של הראשון כתת-טיפוס של מערך של השני, כלומר קטע הקוד הבא חוקי:

```
let bArr: B[] = [ new B(), new B() ];
```

```
let aArr: A[] = bArr;
```

b. כאשר B תת-טיפוס של A, ניתן גם לבצע השמות של B למשתנים מסוג A וגם להעביר ערך מסוג B לפונקציה המצפה לקבל ארגומנט מסוג A.

c. עבור טיפוס נוסף C היורש מ-A (ולא מ-B), נסו לגרום לכך שיכנס איבר מסוג C למערך של איברים מסוג B.

פתרון:

ב TypeScript, טיפוסים סטטיים נבדקים בזמן קומפילציה, אך בערך ניתן להיתקל בשגיאות טיפוס בזמן ריצה כאשר מתבצעות השמות והשמות לטיפוסים יורשים. ניתן לראות דוגמה זאת באיור הבא:

```
class A {
public x: number;
constructor(x: number){
this.x = x;
    }
}

class B extends A  {
public y: number;
constructor(x: number, y: number){
super(x);
this.y = y;
      }
}

class C extends A{
public z: number;
constructor(x: number, z: number){
super(x);
```

```
this.z = z;
    }
}

let bArr: B[ ] = [new B(1, 2), new B(3, 4)];
```

הצבת מערך מסוג B[ ] במשתנה מסוג A[ ]

```
let aArr: A[ ] = bArr ;
```

הכנסת איבר מסוג C למערך של B .

```
aArr.push(new C(5, 6));

console.log(bArr); // שגיאת טיפוס בזמן ריצה!
```

בדוגמה זו אנו משתמשים במערך מסוג B ומציבים אותו במשתנה מסוג A .

לאחר מכן, מכניסים איבר מסוג C למערך של B .

בזמן ריצה, נתקל בשגיאת טיפוס מכיוון שהאיבר החדש שהוסף (מסוג C) איננו תת טיפוס של B .

תוצאות הרצה של הקוד תהיה:

```
TypeError: Cannot push an element of type C to an array of type B[]
```

מסקנה: מספר שגיאות טיפוסים יכול להימצא בזמן ריצה, גם עם שימוש בטיפוסים סטטיים ב-TypeScript .