שפות תכנות – תרגיל מספר 4 חלק יבש

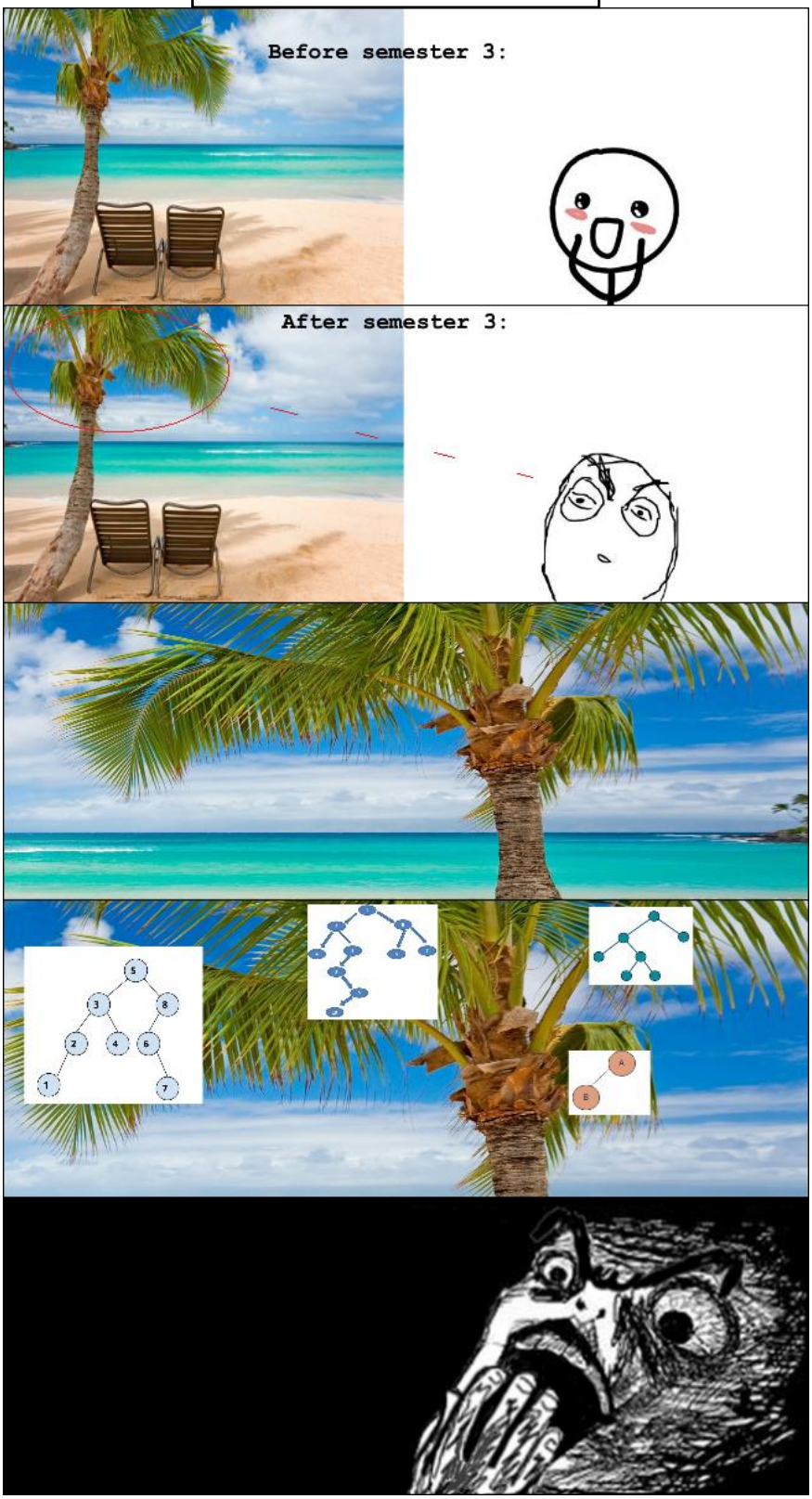
תאריך הגשה: *31.12.17*

מגישים:

סהר כהן – 206824088

יובל נהון – 206866832

חופשת חנוכה: לפני ואחרי



1) בשפת kotlin הביטוי if-else מהווה גם statement וגם command. כלומר, יש לו ערך החזרה. לעומת זאת בשפת c הביטוי if-else מהווה command בלבד ולכן אין לו ערך החזרה. בשפת ml הביטוי if-then-else לא מהווה command מכיוון ש-ml היא שפה פונקציונלית טהורה ו-commands משוויכות לשפות שתומכות בפרדיגמה האימפרטיבית.

2) פונקציה בשפת kotlin מהווה טיפוס בשפה מכיוון שניתן להשתמש בפונקציות כפרמטרים לפונקציות אחרות. ההבדל בין kotlin ל-ml בנושא זה הוא שבקוטלין על המתכנת להגדיר function-type על מנת להשתמש בפונקציות כטיפוסים. לעומת זאת ב-ml אין צורך לעשות זאת- טיפוס הפונקציה נקבע ישירות לפי ערכי הפרמטר וההחזרה של הפונקציה.
בשפת kotlin הגדרה של infix function חייבת להתבצע בהגדרת הפונקציה עצמה על ידי שימוש ב-infix keyword ואילו בשפת ml ההגדרה של infix function יכולה להתבצע לפני או אחרי הגדרת הפונקציה עצמה.

3) Void-safety זוהי תכונה של שפות תכנות שמגדירה את האופן בו שפת תכנות מוודאת שאף פעם לא נעשה dereference לערך שהוא null/void. שפת kotlin מקיימת תכונה זו על ידי פיצול כל ה-references של הטיפוסים לכאלה שיכולים להכיל null (nullable reference) וכאלה שלא יכולים (non-null reference). כאשר נעשה ניסיון ל-dereference ל-reference מסוג non-null reference הפעולה תמיד תהייה חוקית מכיוון שה-reference לא יכול להכיל ערך null ולכן הקומפיילר תמיד יאשר את הפעולה. לעומת זאת כאשר מתכנת ינסה לעשות פעולה כזו בצורה רגילה ב- nullable reference תזרק שגיאה.
על מנת לבצע dereferencing ל-nullable reference יש צורך להשתמש באחת מהשיטות הבאות:
א) בדיקה האם הreference הוא null על ידי שימוש ב-if. ברגע שהקומפיילר יזהה את הבדיקה שנעשתה הוא יאפשר את הפעולה.
ב) להשתמש ב-safe call. למשל עבור הפקודה הבאה a?.length הקומפיילר יחזיר a.length אם a הוא לא null ויחזיר null אם a הוא null.
ג) שימוש ב-Elvis operator.
למשל: val l = b?.length ?: -1 . ביטוי זה הוא זהה לביטוי `val l: Int = if (b != null) b.length else -1`.
אופרטור זה (?:) יחזיר במקרה זה את b.length (הביטוי משמאלו) אם b הוא לא null. אחרת הוא יחזיר 1-(הביטוי מימינו).
ד) שימוש ב-!!. אופרטור זה ינסה לעשות את ההמרה מ-nullable reference ל- non-null reference. במקרה של כשלון(ה-reference הוא null) תזרק שגיאה בזמן ריצה.
ה) שימוש ב-safe cast.
ניתן לשים לב ששפת קוטלין היא שפה חזקה מאוד ב-void safety ומצליחה להעביר את רוב הבעיות שנוצרות בנושא זה מזמן הריצה(כמו ב-java) לזמן קומפילציה.

4) בשפה kotlin קיים פולימורפיזם אוניברסלי. למשל:

```
abstract class Comparable<in T> {
 abstract fun compareTo(other: T): Int
}
fun demo(x: Comparable<Number>) {
 x.compareTo(1.0) // this function can receive any Comparable<T> object
Comparable<Double> = x // OK!
 }
```

הדוגמא למעלה מייצגת פולימורפיזם פרמטרי מכיוון שהמחלקה האבסטרקטית Comparable מייצגת מספר לא חסום של טיפוסים (בגלל ה-template).
בנוסף לכך הדוגמא מייצגת פולימורפיזם מסוג inclusion/sub-typing מכיוון שקיום של מחלקות אבסטרקטיות מכריח קיום תכונה זו. בפרמטר של הפונקציה demo ניתן להכניס כל אובייקט שמשוייך למחלקה שיורשת מ-comparable.
בשפה קוטלין אין תמיכה בפולימורפיזם מסוג ad-hoc.

5) Type punning זוהי היכולת לפרש את דרך התצוגה של v בצורה שלא עקבית עם הטיפוס של v. למשל בשפת c ניסיון לפרש union על ידי שימוש ב-member operator (.) עבור איבר ב-union שלא מחזיק את הערך העדכני יגרום לקריאה של הערך השמור ב-union בצורה שלא נועדה להעשות. למשל (דוגמא מההרצאה):

```
union {
 double foo;
 long bar;
 } baz;
 baz.foo = 3.7;
 printf("%ld\n", baz.bar);
```

בדוגמא זו רק קריאה של foo לא תגרום ל-type punning מכיוון שנעשתה השמה לתוכו. בקריאה של baz.bar התרחש type punning משום שב-union קיים שדה יחיד שמחזיק ערך יחיד שיכול להיות משוייך לכל אחד מהטיפוסים שקיימים ב-union(גודלו הוא גודל המקסימלי ביניהם). ברגע הקריאה ל baz.bar הערך ששמור ב-union הוא בכלל מטיפוס double לכן תתרחש קריאה לא אחידה עם הטיפוס.

דוגמא ל-type punning ב-rust:

```rust
let bitpattern = unsafe {
    std::mem::transmute::<f32, u32>(1.0)
};
assert_eq!(bitpattern, 0x3F800000);
```

בדוגמא זו נעשתה קריאה של משתנה מטיפוס double בצורה שבה ניתן לקרוא את ערכי הביטים שמייצגים את הערך. כמובן שזוהי דרך לא עקבית לקריאה של הערך השמור במשתנה.

6) תוכנית לדוגמא:

```c
#include <stdio.h>
int main() {
   unsigned int x = 0x76543210;
   char *c = (char*) &x;

   printf ("*c is: 0x%x\n", *c);
   if (*c == 0x10)
   {
      printf ("LE\n");
   }
   else
   {
      printf ("BE\n");
   }

   return 0;
}
```

תכנית זו מגדירה משתנה מטיפוס unsigned int ושומרת ערך מסויים בתוכו. לאחר מכן היא מגדירה מצביע מטיפוס char לאותו המשתנה (type punning). בגלל שגודל של char הוא בייט אחד וגודל של unsigned int הוא 4 בייטים ברגע שהשמנו את הכתובת של x לתוך המצביע c אנחנו מצביעים לבייט הראשון בסדר השמירה של הבתים בזיכרון. לכן לפי השוואה של הערך ש-c מצביע עליו לערך 0x10 (10 הקסדצימלי) ניתן לגלות מה הסדר שבו הבתים נשמרים בזיכרון (מהסוף להתחלה או מההתחלה לסוף).

7) Union בשפת c הוא ניסיון למימוש disjoint union שכן בכל רגע נתון הערך השמור בו שייך לאחד הטיפוסים שמהם הוא נבנה, אך לעומת disjoint union אין תיוג שמצביע לאיזה טיפוס שייך הערך הנוכחי בו.

8) Gradual typing- שפת התכנות דוגלת בטיפוסיות דינמית, כלומר הטיפוסים נאכפים בזמן ריצה.
Mixed typing- שפת התכנות דוגלת הן בטיפוסיות סטטית והן בטיפוסיות דינמית. כלומרת חלק מבדיקות הטיפוסים מבוצעות בזמן הידור וחלק מבוצעות בזמן ריצה ויתכן שלפעמים גם וגם.

9) בשפה nimrod יש שימוש במערכים סטטיים וגמישים.
השימוש במערכים סטטיים נעשה על ידי הגדרה של טיפוס מערך(טיפוס הערכים בתוכו וטווח האינדקסים של המערך שמיוצג על ידי כל טיפוס אורדינלי) ושימוש ב-[].דוגמא למערך סטטי:

```
type
  IntArray = array[0..5, int] # an array that is indexed with 0..5
var
  x: IntArray
```

```
x = [1, 2, 3, 4, 5, 6]
for i in low(x)..high(x):
echo x[i]
```

השימוש במערכים גמישים נעשה על ידי sequences. על מנת ליצור sequence יש להשתמש ב-@ ובבנאי המערך הרגיל ([]) כמו בדוגמא הבאה:

```
var
  x: seq[int] # a reference to a sequence of integers
x = @[1, 2, 3, 4, 5, 6]
```

מערך זה יוקצה דינמית על ה-heap וגודלו יכול להשתנות בזמן בריצה של התכנית.