תרגיל בית שלישי (scala). סמסטר 2018 ב (4 יוני 2018, עודכן 12 ליוני (2:39))

יש לענות כל השאלות

בכל השאלות אין להשתמש במשתנים המוגדרים עם var ואין להשתמש ב- while או ב- for

(אלא אם נאמר אחרת). במקום לולאות ניתן להשתמש ברקורסיה.

יש להגדיר את כל הפונקציות "העצמאיות" (כאלו שאינן methods של class ) בתוך

object Exercise3.

שאלה 1

כתבו פונקציה

def evenPlaces[A] (alist: List[A]): List[A]

הפונקציה מקבלת רשימה כארגומנט ומחזירה רשימה חדשה הכוללת רק את האיברים שנמצאים במקומות הזוגיים (האיבר הראשון הוא בעל אינדקס אפס ולכן מיקומו זוגי)

למשל

val mylist = List("a", "b", "c", "d")

evenPlaces(myList) // List("a", "c")

כתבו שתי גרסאות של הפונקציה (קראו לגרסה השניה evenPlaces2 כדי להבדילה מהגרסה הראשונה). אחת הגרסאות תעשה שימוש ב- isEmpty, head, tail (שהן methods של List) והשניה תעשה שימוש ב- pattern matching.

ניתן להגדיר פונקצית עזר שתהיה מקוננת בתוך evenPlaces.

שאלה 2

כתבו פונקציה

def zipMap[A, B] (alist: List[A], funList: List[A => B]): List[B]

הפונקציה מקבלת רשימה alist ורשימה של פונקציות funList. היא מפעילה כל אחת מהפונקציות על האיבר המתאים ב- alist (פונקציה ראשונה על האיבר הראשון, פונקציה שניה על האיבר השני וכן הלאה). הפונקציה מחזירה את רשימת התוצאות.

(במקרה שאחת משתי הרשימות קצרה מהשניה יש להחזיר רשימה באורך של הרשימה הקצרה

מבין השתיים).

למשל:

val mylist = List("a", "bb", "2018")

val myfunlist: List[String => Int] =

List(s => s.length,

s => s.length\*5,

s => s.toInt)

zipMap(myList, myfunList) // List(1, 10, 2018)

שאלה 3.

נייצג רשימות בצורה הבאה. (זה דומה ל- List של scala אבל פשוט יותר).

Empty הוא אוביקט המייצג רשימה ריקה (לא משנה מאיזה טיפוס --

זה בגלל שה- type parameter A הוגדר עם + לפניו ובגלל ש- Empty הוא

subtype של MyList[Nothing] אבל הבנה של זה לא נחוצה לצורך פתרון התרגיל).

Empty ממלא כאן את התפקיד של Nil ב- List של scala.

כל מופע של ה- Pair class מייצג רשימה לא ריקה: head הוא האיבר הראשון

ו- tail היא "הזנב" של הרשימה כלומר הרשימה ללא איברה הראשון. ה- tail

יכול להיות Empty.

**שימו לב שהשם הוא MyList ולא List -- הזהרו לא לכתוב List במקום MyList.**

sealed trait MyList[+A] {

def isEmpty: Boolean

def head: A

def tail: myList[A]

def foldLeft[B](initial: B, f: (B, A) => B): B

def takeWhile(p: A => Boolean): MyList[A]

def enumerate: MyList[(Int, A)]

def dmap[B](f: A => B, g: A => B): MyList[B]

}

final case class Pair[A](head: A, tail: MyList[A])

extends myList[A] {

def isEmpty = false

}

final case object Empty extends MyList[Nothing] {

def head: Nothing =

**throw new** NoSuchElementException("head of empty list")

def tail: MyList[Nothing] =

throw new NoSuchElementException("tail of empty list")

def isEmpty = true

}

יש לממש את כל ה- methods המוגדרים ב- MyList תוך שימוש ב- isEmpty, head ו-

tail שהמימוש שלהם כבר נתון.

הדרכה: ניתן לממש את ה- methods בתוך ה- trait myList (ואז Pair ו- Empty יקבלו אותם בירושה). לחילופין ניתן לממש אותם בתוך Pair ו- Empty.

הסבר מה ה- methods עושים:

foldLeft פועלת באופן דומה ל- foldLeft של List

takeWhile(p) מחזירה רשימה חדשה הכוללת את הרישא הארוכה ביותר של הרשימה המקורית הכוללת איברים עבורם הפונקציה p מחזירה true.

enumerate מחזירה רשימה חדשה של זוגות (tuples). כל זוג כולל איבר מהרשימה המקורית ביחד עם האינדקס שלו ברשימה המקורית (האינדקס מופיע קודם). האיבר הראשון הוא בעל אינדקס אפס.

dmap(f, g) מחזירה רשימה חדשה שבה על כל האיברים ברשימה המקורית שמיקומם זוגי הופעלה הפונקציה f ועל כל האיברים שמיקומם אי זוגי הופעלה g. (האיבר הראשון מיקומו אפס שזה מספר זוגי).

דוגמאות:

val alist = Pair(5, Pair(6, Pair(7, Pair(8, Empty))))

alist.sum // 26 (5+6+7+8 == 26)

alist.foldLeft("start", (s, i) => s + i.toString)

// "start5678"

alist.takeWhile(\_ < 7) // Pair(5, Pair(6, Empty))

alist.enumerate // Pair((0, 5), Pair((1, 6), Pair((2,

7), Pair((3, 8), Empty))))

alist.dmap((x => x.toString + x), (y => y.toString+y+y)) 100))

// Pair("55", Pair("666", Pair("77",

Pair("888",Empty))))

שאלה 4

ממשו שוב את שתי המתודות foldLeft, takeWhile, -- הפעם תוך שימוש ב- pattern matching. יש להשלים את הקוד במקומות המסומנים.

שימו לב שהשם הוא MyList2 (ולא MyList) כדי להבדיל מהשם בשאלה הקודמת.

**sealed** **trait** MyList2[+A] {

**def** isEmpty: Boolean

**def** head: A

**def** tail: MyList2[A]

**def** foldLeft[B](initial: B, f: (B, A) => B): B = this match

**complete this code**

**def** takeWhile(p: A => Boolean): MyList2[A] = this match

**complete this code**

**case** **class** **Pair2**[A](head: A, tail: MyList2[A]) **extends** MyList2[A] {

**def** isEmpty = false

}

**case** **object** **Empty2** **extends** MyList2[Nothing] {

**def** isEmpty = **true**

**def** head: Nothing = **throw** **new** *NoSuchElementException*("head of empty list")

**def** tail: MyList2[Nothing] = **throw** **new** *NoSuchElementException*("tail of empty list")

}

שאלה 5

נייצג ביטויים אריתמטיים בהם אופרטורים פלוס, מינוס, וכפל מופעלים על מספרים

מסןג Int באופן הבא. (נוותר על אופרטור החילוק).

sealed trait Expr {

def eval: Int

def complexity: Int

def fold[B] (plus: (B, B) => B,

minus: (B, B) => B,

mul: (B, B) => B,

number: Int => B)

}

final case class Plus(left: Expr, right: Expr) extends Expr

final case class Minus(left: Expr, right: Expr) extends Expr

final case class Mul(left: Expr, right: Expr) extends Expr

final case class Number(n: Int) extends Expr

השדות left ו- right מתיחסים לאופרנדים של האופרטור.

לדוגמא את הביטוי 3 + 4 \* 5 נייצג כך:

val n1 = Plus(Number(3), Mul(Number(4), Number(5)))

את הביטוי (3 + 4) \* 5 נייצג כך:

val n1 = Mul(Plus(Number(3), Number(4)), Number(5))

סעיף א

יש לכתוב את ה- methods eval and complexity. הגדירו אותם בתוך

Expr תוך שימוש ב- pattern matching או, לחילופין, השאירו אותם בתוך Expr כ-abstract methods (כלומר ללא הגדרה של

גוף הפונקציות) והגדירו אימפלמנטציות שלהם בתוך המחלקות היורשות מ- Expr

(Plus, Minus, Mul, Number) .

eval מחזירה את תוצאת חישוב הביטוי. למשל (בהמשך לדוגמא הנ"ל) n1.eval תחזיר

23.

complexity תחזיר את "הסיבוכיות של הביטוי" כאשר הסיבוכיות מוגדרת באופן הבא:

כל אופרטור פלוס או מינוס מגדיל את הסיבוכיות באחד. כל אופרטור כפל מגדיל אותה בשתיים. לדוגמא לביטוי 3+4\*5 יש סיבוכיות 3. לביטוי 3 + 4\*5\*6 יש סיבוכיות 5.

לביטוי 8 יש סיבוכיות אפס (כי אין אופרטורים).

סעיף ב

נוסיף method בשם fold ל- Expr. הנה ההגדרה שלה:

sealed trait Expr {

...

def fold[B] (plus: (B, B) => B,

minus: (B, B) => B,

mul: (B, B) => B,

number: Int => B): B = this match {

case Plus(op1, op2) => {

val op1Fold = op1.fold(plus, minus, mul, number)

val op2Fold = op2.fold(plus, minus, mul, number)

plus(op1Fold, op2Fold)

}

case Minus(op1, op2) => {

val op1Fold = op1.fold(plus, minus, mul, number)

val op2Fold = op2.fold(plus, minus, mul, number)

minus(op1Fold, op2Fold)

}

case Mul(op1, op2) => {

val op1Fold = op1.fold(plus, minus, mul, number)

val op2Fold = op2.fold(plus, minus, mul, number)

mul(op1Fold, op2Fold)

}

case Number(n) => number(n)

}

}

fold זו method גנרית המחזירה ערך מטיפוס B המחושב עבור הביטוי עליו היא מופעלת .

היא מזכירה את foldLeft ו- foldRight שמחשבות ערך כשהן מופעלות על List.

הארגומנטים של fold הם ארבע פונקציות. הארגומנט הראשון (plus -- שימו לב שהוא מתחיל באות קטנה כדי להבדילו מה- case class Plus) משמש לחישוב

התוצאה כאשר הביטוי הוא סכום של שני תתי ביטויים. הארגומנטים של הפונקציה plus

הם התוצאות המחושבות (באופן רקורסיבי) עבור שני תתי הביטויים (האופרנדים של אופרטור הפלוס). לארגומנטים (של fold) minus ו- mul יש תפקיד דומה: משתמשים בהם כאשר הביטוי הוא הפרש או מכפלה של שני תתי ביטויים. הארגומנט האחרון של fold (number)

משמש לחישוב התוצאה כאשר הביטוי הוא מספר פשוט.

הנה דוגמא לשימוש ב- fold שיחזיר רשימה של כל המספרים המופיעים בביטוי:

n1.fold[List[Int]]((list1, list2) => list1 ++ list2,

(list1, list2) => list1 ++ list2,

(list1, list2) => list1 ++ list2,

(list1, list2) => list1 ++ list2,

n => List(n)

)

התוצאה תהיה List(3, 4, 5).

כתבו שוב את eval ואת complexity תוך שימוש ב- fold.

(קראו להם eval2 ו- complexity2 כדי להבדיל בינן לבין ה- methods של סעיף א).

שאלה 6

לפעמים ניתן להסיק מה המשמעות של פונקציה מהחתימה שלה.

ממשו את הפונקציה הבאה:

def map2[A,B,C](o1: Option[A], o2: Option[B],

f:(A,B)=>C): Option[C]