+1/1/60+

EPFL

1

**Profs. Viktor Kunčak, Martin Odersky, and Clément Pit-Claudel**
**CS-214 Software Construction**
**30.10.2023 from 12:15 to 13:00**
**Duration: 45 minutes**

SCIPER : **1000001**

ROOM : **SG 1**

# Annie Easley

**Wait for the start of the exam before turning to the next page. This document is printed double sided, 12 pages. Do not unstaple.**

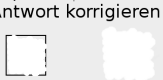
| | |
|---|---|
| **Material** | This is a closed book exam. Paper documents and electronic devices are not allowed. Place on your desk your student ID and writing utensils. Place all other personal items at the front of the room. If you need additional draft paper, raise your hand and we will provide some. |
| **Time** | All points are not equal: we do not think that all exercises have the same difficulty, even if they have the same number of points. Manage your time accordingly. You may want to look at the whole exam before starting on a particular exercise. |
| **Appendix** | The last page of this exam contains an appendix which is useful for formulating your solutions. Do not detach this sheet. |
| **Use a pen** | For technical reasons, **only use black or blue pens for the MCQ part, no pencils!** Use white corrector if necessary. |
| **Grading Scheme** | The exam contains a total of 50 points. For multiple choice questions, a good answer is worth 4 points and a bad answer 0 points. Note that there is always exactly one good answer to each question. For true-false questions, a good answer is worth 2 points and a bad answer 0 points. For open questions, the number of points is variable and indicated at the top of each question. |
| **Stay Functional** | Do not use **`var`**s, **`while`** loops, **`for...do`** loops, etc. This will result in 0 points for that question. |

Respectez les consignes suivantes | Read these guidelines | Beachten Sie bitte die unten stehenden Richtlinien

choisir une réponse | select an answer
Antwort auswählen

ne PAS choisir une réponse | NOT select an answer
NICHT Antwort auswählen

Corriger une réponse | Correct an answer
Antwort korrigieren



ce qu'il ne faut **PAS** faire | what should **NOT** be done | was man **NICHT** tun sollte

# Span (12 pts)

**Question 1** This question is worth 12 points.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6 ☐7 ☐8 ☐9 ☐10 ☐11 ☐12 *Do not write here.*

Your task is to complete the functions `span` and `spanTailRec`. Both functions implement the same functionality.

Both functions take a list `l` and a predicate `p` as parameters and return a pair of lists `(l1, l2)`, where

- `l1` is the longest prefix of `l` where all elements satisfy `p`, and
- `l2` is the remainder of the list.

In addition, `spanTailRec` takes a third `acc` parameter. Its initial value is `Nil`.

Furthermore, the `spanTailRec` function must be tail-recursive.

Here are example tests that your implementations must pass successfully:

```
class SpanTest extends munit.FunSuite:
  test("span(List(1, 2, 3, 4), _ < 3)"):
    val l = List(1, 2, 3, 4)
    val (l1, l2) = span(l, _ < 3)
    assertEquals(l1, List(1, 2))
    assertEquals(l2, List(3, 4))

  test("spanTailRec(List(1, 2, 3, 4), _ < 3)"):
    val l = List(1, 2, 3, 4)
    val (l1, l2) = spanTailRec(l, _ < 3)
    assertEquals(l1, List(1, 2))
    assertEquals(l2, List(3, 4))

  test("span(List(1, 2, 3), _ % 2 == 1)"):
    val l = List(1, 2, 3)
    val (l1, l2) = span(l, _ % 2 == 1)
    assertEquals(l1, List(1))
    assertEquals(l2, List(2, 3))

  test("spanTailRec(List(1, 2, 3), _ % 2 == 1)"):
    val l = List(1, 2, 3)
    val (l1, l2) = spanTailRec(l, _ % 2 == 1)
    assertEquals(l1, List(1))
    assertEquals(l2, List(2, 3))
```

You should only write in the boxes below, and you cannot declare new `val`s or `def`s.

Futhermore, `span` and `spanTailRec` must not call each other.

```
def span[T](l: List[T], p: T => Boolean): (List[T], List[T]) =
  l match

    case Nil =>

    case x :: xs =>

      if p(x) then

        val (ys, zs) =

      else
```

```
@scala.annotation.tailrec
def spanTailRec[T](
    l: List[T],
    p: T => Boolean,
    acc: List[T] = Nil
): (List[T], List[T]) =
  l match

    case Nil =>

    case x :: xs =>

      if p(x) then

      else
```

# Concat (10 pts)

Consider the following definition of lists of characters:

```
enum CharList:
    case CharNil
    case CharCons(char: Char, tail: CharList)

import CharList.*
```

A correct implementation must return and pass, among others, the tests defined in `testConcat`:

```
  import CharList.*
  val l1 = CharCons('a', CharNil)
  val l2 = CharCons('b',  CharCons('c', CharNil))
  val l12 = CharCons('a', CharCons('b', CharCons('c', CharNil)))
  val l21 = CharCons('b', CharCons('c', CharCons('a', CharNil)))

  def testConcat(version: Int, concat: (CharList, CharList) => CharList) =
    test("concat" + version + "(l1, l2) returns the right result"):
      assertEquals(concat(l1, l2), l12)
    test("concat" + version + "(l2, l1) returns the right result"):
      assertEquals(concat(l2, l1), l21)
```

Given below are 5 different implementations of the `concat` function. For each implementation, tick "Yes" if it is correct (for all possible inputs), or "No" if it is incorrect.

```
def concat1(l1: CharList, l2: CharList): CharList =
    l2 match
        case CharNil => l2
        case CharCons(char, tail) => CharCons(char, concat1(tail, l1))
```

**Question 2** Is `concat1` correct?

☐ Yes ☐ No

```
def concat2(l1: CharList, l2: CharList): CharList =
    l1 match
        case CharNil => l2
        case CharCons(char, tail) => CharCons(char, concat2(tail, l2))
```

**Question 3** Is `concat2` correct?

☐ Yes ☐ No

```
def concat3(l1: CharList, l2: CharList): CharList =
    l2 match
        case CharNil => l1
        case CharCons(char, tail) => concat3(CharCons(char, l1), tail)
```

**Question 4** Is `concat3` correct?

☐ Yes ☐ No

```
def concat4(l1: CharList, l2: CharList): CharList =
    l1 match
        case CharNil => l2
        case CharCons(char, tail) => concat4(CharCons(char, l2), tail)
```

**Question 5** Is `concat4` correct?

☐ Yes ☐ No

```
def concat5(l1: CharList, l2: CharList): CharList =
    l1 match
        case CharNil => l2
        case CharCons(char, tail) => concat5(tail, CharCons(char, l2))
```

**Question 6** Is `concat5` correct?

☐ Yes ☐ No

# Proof of SumFlatMap (12 pts)

**Question 7** This question, consisting of both cases of the proof, is worth 12 points.

0 1 2 3 4 5 6 7 8 9 10 11 12 *Do not write here.*

All lemmas on this page hold for all types `T` and `S`, and all `l: List[T]`, `x: T`, `xs: List[T]`, `y: Int`, `ys: List[Int]`, `zs: List[Int]`, `f: T => S`, `g: T => List[S]`, and `h: T => List[Int]`.

Given the following lemmas:

(MapNil) `Nil.map(f) === Nil`

(MapCons) `(x::xs).map(f) === f(x) :: xs.map(f)`

(FlatMapNil) `Nil.flatMap(g) === Nil`

(FlatMapCons) `(x::xs).flatMap(g) === g(x) ++ xs.flatMap(g)`

(SumNil) `sum(Nil) === 0`

(SumCons) `sum(y::ys) === y + sum(ys)`

(SumConcat) `sum(ys ++ zs) === sum(ys) + sum(zs)`

You need to prove:

(SumFlatMap) `sum(l.flatMap(h)) === sum(l.map(h).map(sum))`

Complete the proof below. For each step, you must write the name of the lemma you are using. You may only use the lemmas above.

The proof is done by induction on `l`.

*Base case:* `l` is `Nil`. Therefore, you need to prove:

`sum(Nil.flatMap(h)) === sum(Nil.map(h).map(sum))`

*Induction step:* `l` is `x :: xs`. Therefore, you need to prove:

```
sum((x::xs).flatMap(h)) === sum((x::xs).map(h).map(sum))
```

given that the induction hypothesis, named IH, holds:

(IH) `sum(xs.flatMap(h)) === sum(xs.map(h).map(sum))`

# Mystery function (6 pts)

**Question 8** This question is worth 6 points.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6 *Do not write here.*

In this exercise, your task is to use the substitution method to write the step-by-step evaluation of an expression, under the call-by-value evaluation strategy.

You must apply the definition of a single function call at a time and write the result of each step. You can directly reduce if-then-else expressions to their branches.

As an example, consider the function `factorial`:

```
def factorial(n: Int): Int =
  if n == 0 then 1
  else n * factorial(n-1)
```

The expression `factorial(1)` evaluates step-by-step as follows:

```
    factorial(1)
=== 1 * factorial(0)
=== 1 * 1
=== 1
```

Now, consider the function `f`:

```
def f(a: Int, b: Int): Int =
  if b > a then 0
  else f(a - b, b) + 1
```

Write the step-by-step evaluation of the expression `f(5, 2)`:

What does `f` do, in one word, assuming the arguments are positive integers?

# Collect (10 pts)

**Question 9** This question is worth 10 points.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6 ☐7 ☐8 ☐9 ☐10 *Do not write here.*

Your task is to implement the function `collect`. This function takes a list `l: List[T]` and a function `f: T => Option[U]` as parameters, and returns a list of all the elements `y` for which there is an element `x` in `l` such that `f(x) == Some(y)`.

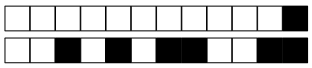
Here is one example test that your implementation must pass successfully:

```
test("""collect(List("abc", "451", "true", "1984"), _.toIntOption)"""):
  val l = List("abc", "451", "true", "1984")
  assertEquals(collect(l, _.toIntOption), List(451, 1984))
```

where `toIntOption` is a method of `String` that returns `Some(x)` if the string represents an integer `x`, or `None` otherwise.

Assuming the runtime complexity of `f` is constant ($\mathcal{O}(1)$), the runtime complexity of your implementation should not be more than linear ($\mathcal{O}(n)$).

```
def collect[T, S](l: List[T], f: T => Option[S]): List[S] =
```





# Appendix: Scala Standard Library Methods

Here are the prototypes of some Scala classes that you might find useful:

```
abstract class List[+A]:
  // Adds an element at the beginning of this list.
  def ::[B >: A](elem: B): List[B]
  // Get the element at the specified index.
  def apply(n: Int): A
  // Tests whether this list contains a given value as an element.
  def contains[A1 >: A](elem: A1): Boolean
  // Selects all elements except first n ones.
  def drop(n: Int): List[A]
  // Drops longest prefix of elements that satisfy a predicate.
  def dropWhile(p: A => Boolean): List[A]
  // Selects all elements of this list which satisfy a predicate.
  def filter(pred: A => Boolean): List[A]
  // Selects all elements of this list which do not satisfy a predicate.
  def filterNot(pred: A => Boolean): List[A]
  // Builds a new list by applying a function to all elements of this list and
  // using the elements of the resulting collections
  def flatMap[B](f: A => List[B]): List[B]
  // Applies a binary operator to a start value and all elements of this
  // sequence, going left to right.
  def foldLeft[B](z: B)(op: (B, A) => B): B
  // Applies a binary operator to a start value and all elements of this
  // sequence, going right to left.
  def foldRight[B](z: B)(op: (A, B) => B): B
  // Tests whether a predicate holds for every element of this collection
  def forall(p: A => Boolean): Boolean
  // Selects the first element of this list.
  def head: A
  // Computes the multiset intersection between this sequence and another sequence.
  def intersect[B >: A](that: Seq[B]): List[A]
  // Selects the last element.
  def last: A
  // Applies the function f to each element in the list.
  def map[B](f: A => B): List[B]
  // Returns a new list with elements in reversed order.
  def reverse: List[A]
  // The size of this collection.
  def size: Int
  // Sorts this sequence according to an Ordering.
  def sorted[B >: A](implicit ord: Ordering[B]): List[A]
  // Selects all elements except the first.
  def tail: List[A]
  // Takes longest prefix of elements that satisfy a predicate.
  def takeWhile(p: A => Boolean): List[A]

object List:
  // Produces a collection containing the results of some element computation a
  // number of times.
  def fill[A](n: Int)(elem: => A): List[A] = ???

abstract class ParList[+A] extends List[A]:
  // Aggregates the results of applying an operator to subsequent elements.
  def aggregate[B](z: => B)(seqop: (B, A) => B, combop: (B, B) => B): B
```

```
abstract class Option[+A]:
  // Returns this option's value.
  def get: A
  // Returns true if this option is an instance of Some, false otherwise.
  def isDefined: Boolean
  // Returns true if this option is None, false otherwise.
  def isEmpty: Boolean

object math:
  // Returns the value rounded down to an integer.
  def floor(x: Double): Double = ???
  // Returns the value of the first argument raised to the power of the second
    argument.
  def pow(x: Double, y: Double): Double = ???
  // Returns the square root of a Double value.
  def sqrt(x: Double): Double = ???

abstract class Double:
  // Converts this value to an integer
  def toInt: Int
```