

Profs. Martin Odersky and Viktor Kuncak

CS-210 Functional programming

Date: 08.12.2021

Duration: 25 minutes (dry run).
The real exam will last 90 minutes.

1

Ada Lovelace

 $\mathrm{SCIPER}\colon 1000001$

Wait for the start of the exam before turning to the next page. This document is printed double sided, 8 pages. Do not unstaple.

- This is a closed book exam. No electronic devices allowed.
- Place on your desk: your student ID, writing utensils place all other personal items below your desk or on the side.
- You each have a different exam. For technical reasons, do use black or blue pens for the MCQ part, no pencils! Use white corrector if necessary.
- Your 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.

This dry run contains 4 multiple choice questions worth 4 points each, 1 true/false questions worth 2 points and 1 open questions worth 12 points, for a total of **30 points**.

The real exam will last 90 minutes and will have a total of 100 points:

- 16 multiple choice questions with a single correct answer: +4 for the correct answer, 0 otherwise.
- 6 true/false questions: +2 for the correct answer, -1 for a wrong answer, 0 if left unanswered.
- 2 open questions worth 12 points each.
- Your Attention: The exam problems are precisely and carefully formulated, some details can be subtle. Pay attention, because if you do not understand a problem, you cannot obtain full points.
- Stay Functional: You are strictly forbidden to use return statements, mutable state (vars) and mutable collections in your solutions.
- The last page of this exam contains an appendix. Do not detach this page.

Respectez les consignes suivantes Observe this 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 <u>PAS</u> faire what should <u>NOT</u> be done was man <u>NICHT</u> tun sollte								

First part: single choice questions

Each question has **exactly one** correct answer. Marking the box corresponding to the correct answer (and only that one) will get you 4 points. Otherwise, you will get 0 points. There are no negative points.

Given the following lemmas, holding for all types A, x: A, b1: Bool, b2: Bool, p: A \Rightarrow Bool, xs: List[A] and ys: List[A]:

```
(FORALLNIL) nil.forall(p) === True

(FORALLCONS) (x :: xs).forall(p) === p(x) && xs.forall(p)

(EXISTSNIL) nil.exists(p) === False

(EXISTSCONS) (x :: xs).exists(p) === p(x) || xs.exists(p)

(NEGFALSE) !False === True

(NEGOR) ! (b1 || b2) === !b1 && !b2

(NEGAND) ! (b1 && b2) === !b1 || !b2
(NEGINVOLUTIVE) !!b1 === b1
```

Let us prove the following lemma for all 1: List[A] and all p: A => Bool

```
(LISTNEGEXISTS) !1.exists(x \Rightarrow !p(x)) === 1.forall(p)
```

We prove it by induction on 1.

Base case: 1 is Nil.

Therefore, we need to prove:

```
!Nil.exists(x \Rightarrow !p(x)) === Nil.forall(p)
```

Question 1 Starting from the left hand-side (!Nil.exists(x => !p(x))), what exact sequence of lemmas should we apply to get the right hand-side (Nil.forall(p))?

NEGINVOLUTIVE, FORALLNIL, EXISTSNIL
FORALLNIL, NEGFALSE, EXISTSNIL
NegFalse, ExistsNil, ForallNil,
NEGFALSE, FORALLNIL, EXISTSNIL
ExistsNil, NegInvolutive, ForallNil
EXISTSNIL, NEGFALSE, FORALLNIL
FORALLNIL, NEGINVOLUTIVE, EXISTSNIL
NEGINVOLUTIVE, EXISTSNIL, FORALLNIL

For your examination, preferably print documents compiled from auto-multiple-choice.

Induction step: let 1 = x :: xs. Therefore, we need to prove: !(x :: xs).exists(x => !p(x)) === (x :: xs).forall(p)Our inductions hypothesis is that for xs: (IH) !xs.exists(x \Rightarrow !p(x)) === xs.forall(p) Question 2 Starting from the left hand-side (!(x :: xs).exists(x \Rightarrow !p(x))), what exact sequence of lemmas should we apply to get the right hand-side ((x :: xs).forall(p))? EXISTSCONS, NEGFALSE, IH, NEGAND, FORALLCONS EXISTSCONS, NEGOR, NEGINVOLUTIVE, IH, FORALLCONS EXISTS CONS, IH, NEGINVOLUTIVE, NEGAND, FORALL CONS EXISTSCONS, NEGINVOLUTIVE, IH, NEGOR, FORALLCONS EXISTSCONS, NEGAND, NEGINVOLUTIVE, IH, FORALLCONS EXISTSCONS, NEGFALSE, IH, NEGOR, FORALLCONS EXISTSCONS, IH, NEGAND, NEGFALSE, FORALLCONS EXISTS CONS, NEGINVOLUTIVE, NEGOR, IH, FORALL CONS EXISTS CONS, NEGINVOLUTIVE, IH, NEGAND, FORALL CONS EXISTSCONS, NEGFALSE, NEGOR, IH, FORALLCONS EXISTSCONS, NEGFALSE, NEGAND, IH, FORALLCONS ExistsCons, IH, NegOr, NegFalse, ForallCons EXISTSCONS, NEGOR, NEGFALSE, IH, FORALLCONS EXISTSCONS, NEGAND, NEGFALSE, IH, FORALLCONS EXISTSCONS, NEGINVOLUTIVE, NEGAND, IH, FORALLCONS EXISTSCONS, IH, NEGFALSE, NEGOR FORALLCONS

and false are functions of two parameters: Church encoding of tru: t => f => t Church encoding of fls: t => f => f Question 3 What does the following function implement? b => c => b c fls not(b and c) b or c not b not(b xor c) b and c not c Question 4 What should replace ??? so that the following function computes not (b and c)? b => c => b ??? (not b) (not b) (not c) tru fls

Church booleans are a representation of booleans in the lambda calculus. The Church encoding of true

+1	15/	15	6+

Second part: yes/no questions

The answer of each question is either "Yes", either "No". Marking the box corresponding to the correct answer (and only that one) will get you 2 points. Not marking anything will get you 0 points. Otherwise, you will get -1 points

Question 5 Is "type-directed programming" a language mechanism that infers types from values?

Yes No



Third part, open questions

Question 5: This question is worth 12 points.



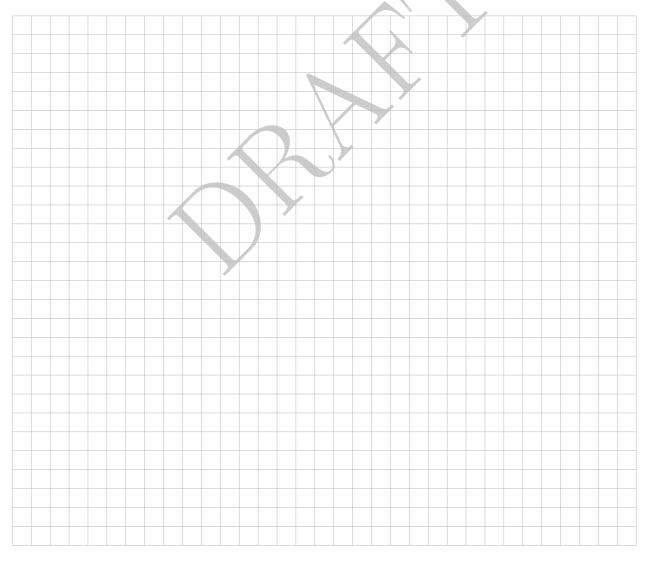
Monoids can be represented by the following type class:

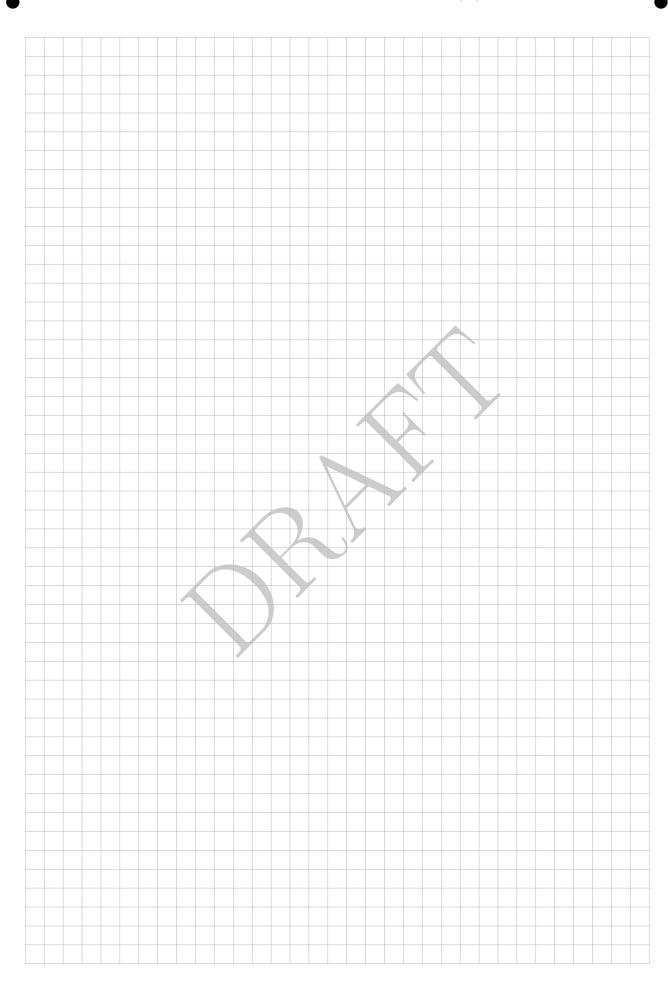
```
trait Monoid[T]:
    extension (x: T) def combine (y: T): T
def unit: T
```

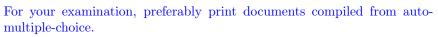
Additionally the three following laws should hold for all Monoid[M] and all m1, m2, m3: M:

```
(Associativity) a.combine(b).combine(c) === a.combine(b.combine(c))
  (Left unit) unit.combine(a) === a
  (Right unit) a.combine(unit) === a
```

Write a Monoid implementation for pairs of arbitrary types (A, B) as a **given**, using Monoid[A] and Monoid[B].







Appendix: Scala Standard Library Methods

Here are some methods from the Scala standard library that you may find useful. If xs is a List [A] then:

- xs.head: A returns the first element of the list. Throws an exception if the list is empty.
- xs.tail: List[A] returns the list xs without its first element. Throws an exception if the list is empty.
- x :: (xs: List[A]): List[A] prepends the element x to the left of xs, returning a List[A].
- xs ++ (ys: List[A]): List[A] appends the list ys to the right of xs, returning a List[A].
- xs.apply(n: Int): A, or xs(n: Int): A returns the n-th element of xs. Throws an exception if there is no element at that index.
- xs.drop(n: Int): List[A] returns a List[A] that contains all elements of xs except the first n ones. If there are less than n elements in xs, returns the empty list.
- xs.filter(p: A => Boolean): List[A] returns all elements from xs that satisfy the predicate p as a List[A].
- xs.flatMap[B] (f: A => List[B]): List[B] applies f to every element of the list xs, and flattens the result into a List[B].
- xs.foldLeft[B](z: B)(op: (B, A) => B): B applies the binary operator op to a start value and all elements of the list, going left to right.
- xs.foldRight[B](z: B)(op: (A, B) => B): B applies the binary operator op to a start value and all elements of the list, going right to left.
- xs.foreach[U](f: (A) => U): Unit applies f to each element for its side effects.
- xs.map[B] (f: A => B): List[B] applies f to every element of the list xs and returns a new list of type List[B].
- xs.max[A] (using ord: Ordering[A]): A finds the largest element of the list xs.
- xs.min[A] (using ord: Ordering[A]): A finds the smallest element of the list xs.
- xs.isEmpty: Boolean returns true if the list has zero element, false otherwise.
- xs.nonEmpty: Boolean returns **true** if the list has at least one element, **false** otherwise.
- xs.reduce[A] (op: (A, A) => A): A reduces the elements of xs using the specified associative binary operator.
- xs.reduceLeft[A] (op: (A, A) => A): A applies a binary operator to all elements of xs, going left to right.
- xs.reduceRight[A] (op: (A, A) => A): A applies a binary operator to all elements of xs, going right to left.
- xs.reverse: List[A] reverses the elements of xs.
- xs.size: Int returns the number of elements xs.
- xs.sorted[A] (using ord: Ordering[A]): List[A] sorts xs according to an Ordering.
- xs.take(n: Int): List[A] returns a List[A] containing the first n elements of xs. If there are less than n elements in xs, returns these elements.
- xs.zip(ys: List[B]): List[(A, B)] zips elements of xs and ys in a pairwise fashion. If one list is longer than the other one, remaining elements are discarded. Returns a List[(A, B)].

The trait Ordering contains a single abstract method. If ord is an Ordering, then:

• ord.compare(x: T, y: T): Int returns an integer whose sign communicates how x compares to y.