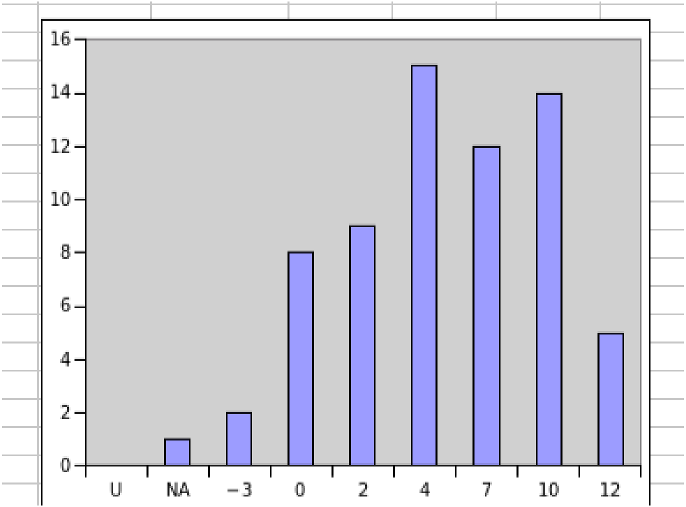
Grades 2018 Spring



Info about the exam

- **Time:** January 9th, 0900-1300
- **Format:** Written exam, 4 hours, on PCs
- **Aids:** All written and online materials (code, notes, books, book's GitHub website, etc.)
- **No time to Google a lot,** but recalling details of a type, which you know where to find, will be entirely possible.
- **Independent:** Interaction with other students and other individuals during the exam is not allowed, and will have severe consequences (you will be monitored for suspicious activity)
- **Questions:** You will get all questions in a scala file that you need to complete. No automated tests.
- **Hand in** an ASCII scala file (probably one, but this will be specified)
- A trial bundle released before the exam.

Scala in the exam

- Answers **graded manually**.
- **External** examiner: Mads Rosendahl
- **Don't solve** questions in an IDE to the point when they **compile and work** (at least not before you have drafted all the answers).
- Grading **permissive on small issues** (semicolons, punctuation, small deviations in function names, switching between curried and uncurried parameters, unless the question is about currying, etc).
- We will not check whether the **type inference** succeeded (as long as a human reader could infer types).
- Do not reorder, or re-factor the file.
- You can use **any functions and types from the course** (textbook, exercises) in your solutions, unless stated otherwise
- Answers to questions that require natural language should be put in comments in the scala file.

Exam preparation

- (Re-)**reading** the book is not sufficient
- (Re-)doing **exercises**, quizzes, the fake exam is more effective
- **Reflecting on different parts of material** and inventing questions that **combine** them is better
- **Discussing** such questions in a study group is most effective.
- We monitor learnIT for questions until the exam date (best effort)

Questions?

Referential Transparency

Disclaimer: what follows are **not** examples of exam questions

- Let `ones` be an infinite streams of integer constants, all equal to `1`
- Let `s` be a stream of integers with unknown values.

```
1 ones.flatMap (x => ones).filter (x => scala.util.random.nextInt != 0)
2 ones.flatMap (x => ones).map (x => scala.util.random.nextInt)
3 ones.flatMap (x => ones).filter (_ => true)
4 while (true) print ("1")
5 s.map ( x => if (x==0) throw DivisionByZero else 1/x )
```

Which line contains a referentially transparent expression? (go vote!)

Be able to **translate** imperative to functional and back

Monoids, Functors and Monads

```scala
trait Monoid[A] {
  def op(a1: A, a2: A): A
  def zero: A
}

trait Functor[F[_]] {
  def map[A,B] (fa: F[A]) (f: A => B) :F[B]
}

trait Monad[F[_]] {
  def unit[A]  (a: => A): F[A]
  def flatMap[A,B] (ma: F[A]) (f: A => F[B]) :F[B]
}
```

1. Which of these is a higher kind: Monoid? Monad? Functor?
2. Is monoid a special case of a monad ?
3. Is monad a special case of a functor ?
4. Is functor a special case of a monoid ?

Associativity in monoids

Assume that `l` contains integers $x_1, \ldots, x_n$

```
l.foldLeft (0) (_ + _)
l.foldRight (0) (_ + _)
l.reverse.foldRight (0) (_ + _)
l.foldRight (1) (_ + _)
```

Which of the above expressions computes $\sum_{i=1}^{n} x_i$?

Be prepared to use properties of types to construct solutions.

Algebraic Data Types, Recursion

```scala
sealed trait Tree[A]
case class Branch[A] (l: Tree[A], a: A, r: Tree[A]) extends Tree[A]
case class Leaf[A] (a: A) extends Tree[A]
```

```scala
def height[A] (t: Tree[A]) :Int = t match {
  case Leaf => 0
  case Branch(l,a,r) => ??? // <-- what should we insert here?
}
```

1 `height (l) + height (r)` ?

2 `height (l) - height (r)` ?

3 `throw Exception ("NonEmpty tree")` ?

4 `max (height (l), height(r)) + 1` ?

5 `min (height (l), height(r)) + 1` ?

tip: ask similar Qs about other structures (finger trees, lists, streams)

Polymorphic Recursion

```
1 def addL[A] (a: A, t: FingerTree[A]) :FingerTree[A] = t match {
2   case Empty () => Single (a)
3   case Single (b) => Deep (Digit(a), Empty(), Digit(b))
4   case Deep (Digit (b,c,d,e), m, sf) =>
5     Deep (Digit(a,b), addL[Node[A]] (Node3 (c,d,e), m), sf)
6   case Deep (pr, m, sf) => Deep (a::pr, m, sf)
7 }
```

- Which line contains the polymorphicly recursive call?
- How do I recognize such a call?
- What property is ensured by types in this implementation?

Questions ?