Lazy Lists

The test suite and the exercise template assume that you are using Scala standard library immutable lists (not the lists of the text book).

Hand in: `Exercises.scala`

Exercise 1. Define functions `from` and `to` that generate lazy lists of integer numbers above (and below) a given number `n`. The function `from` should create a lazy list producing all numbers larger than `n`, starting from `n`, increasing. The function `to` should create a lazy list producing all numbers smaller than n, starting from n, decreasing. In both cases, the head of the list is `n` and both lists are infinite. In the source file, this exercise is in the very bottom, in the companion object of `LazyList`.

```
def from(n: Int): LazyList[Int]
def to(n: Int): LazyList[Int]
```

Use `from` to create a value `naturals: LazyList[Int]` representing all natural numbers in order.[1]

Exercise 2. Write a function `toList` that converts a `LazyList` to a `List`. It forces the list and allows to display it in the REPL using the regular `List.toString`. Convert to the `List` type in the standard library. Use pattern matching.

```
def toList: List[A]
```

Test this function using the factory of lazy lists to build finite ones and them convert to lists (to see whether they yield expected lists). Then create a few finite lazy lists of integers in the REPL and convert them to lists using `toList`.[2]

Exercise 3. Write the function `take(n)` for returning the first `n` elements of a LazyList, and `drop(n)` for skipping the first `n` elements of a `LazyList`. Use pattern matching.

```
def take(n: Int): LazyList[A]
def drop(n: Int): LazyList[A]
```

For fluency, try the following test case in the REPL (should terminate with no memory exceptions and very fast). Why does it terminate without exception? Answer this question as a comment in the Scala file under the same exercise number.

```
naturals.take(1000000000).drop(41).take(10).toList
```

Exercise 4. Write the function `takeWhile(p)` that returns the longest prefix of a `LazyList` in which all elements match the predicate `p`. Use pattern matching.

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

Test your implementation on the following test case in the REPL:

```
naturals.takeWhile { _ < 1000000000 }.drop(100).take(50).toList
```

The above should terminate very fast, with no exceptions thrown. Why?[3]

[1] Exercise 5.9 [Pilquist, Chiusano, Bjarnason, 2022]
[2] Exercise 5.1 [Pilquist, Chiusano, Bjarnason, 2022]
[3] Exercise 5.3 [Pilquist, Chiusano, Bjarnason, 2022]

Exercise 5. Implement `forAll(p)` that checks that all elements in `this LazyList` satisfy a given predicate. Terminate the traversal as soon as it encounters a non-matching value. Use recursion and pattern matching.

```
def forAll(p: A => Boolean): Boolean
```

We use the following test case for `forAll`: `naturals.forAll { _ < 0 }`

If we used this one, it would be crashing: `naturals.forAll { _ >=0 }`. Why?

Recall that `exists` has already been implemented before (in the book). Both `forAll` and `exists` are a bit strange for infinite lazy lists; you should not use them unless you know the result; but once you know the result there is no need to use them. They are fine to use on finite lazy lists. Why?[4]

Exercise 6. Use `foldRight` to implement `takeWhile`.[5]

Exercise 7. Implement `headOption` using `foldRight`.

Exercise 8. Implement the following functions. The task involves designing their types. Implement map, filter, append, and flatMap using foldRight (no recursion). The append method should be non-strict in its argument. We list several interesting test cases for the REPL.[6]

1. `map(f)`, using an analogous signature to the one from lists
   Test case: `naturals.map(_*2).drop(30).take(50).toList`
2. `filter(p)`
   Test case: `naturals.drop(42).filter(_%2 ==0).take(30).toList`
3. `append(that)`
   This one requires sorting out the variance of type parameters carefully. You may find it easier to implement it as a function in the companion object first.
   Test case: `naturals.append(naturals)` (useless, but should not crash)
   Test case: `naturals.take(10).append(naturals).take(20).toList`
4. `flatMap`
   Test case: `naturals.flatMap(to).take(100).toList`
   Test case: `naturals.flatMap(x =>from(x)).take(100).toList`

Exercise 9. The book presents the following implementation for find:

```
def find(p: A => Boolean): Option[A]= this.filter(p).headOption
```

Explain why this implementation is suitable (efficient) for lazy lists and would not be optimal for lists. (No automatic test here—if you want feedback, talk to Andrzej or TAs. The exam will contain open questions to be answered in English.)

Exercise 10. Compute a lazy list of Fibonacci numbers `fibs`: 0, 1, 1, 2, 3, 5, 8, and so on. It can be done with functions available so far. Test it in REPL by translating a finite prefix of `fibs` to `List`, and a finite prefix of some infinite suffix.[7] Again, no ready-made tests, because the types are not prescribed (you need to write the type yourself).

[4]Exercise 5.4 [Pilquist, Chiusano, Bjarnason, 2022]
[5]Exercise 5.5 [Pilquist, Chiusano, Bjarnason, 2022]
[6]Exercise 5.7 [Pilquist, Chiusano, Bjarnason, 2022]
[7]Exercise 5.10 [Pilquist, Chiusano, Bjarnason, 2022]

Exercise 11. Write a more general lazy-list building function called `unfold`. It takes an initial state, and a function for producing both the next state and the next value in the generated lazy list.

```
def unfold[A, S] (z: S) (f: S => Option[(A, S)]): LazyList[A]
```

If you solve it *without* using pattern matching, then you obtain a particularly concise solution, that combines aspects of this and last week's material.

You can test this function in REPL by unfolding the lazy list of natural numbers and checking whether its finite prefix is equal to the corresponding prefix of `naturals`.[8]

The exercise is placed in the companion object of `LazyList`, in the bottom of `LazyList.scala`.

Exercise 12. Write `fib` in terms of `unfold`.[9] Use this test case in the REPL:

`fibsUnfold.take (100).toList ==fibs.take (100).toList`.

Exercise 13. Use unfold to implement `map`, `take`, `takeWhile`, and `zipWith`.[10]

Note that there is a choice whether the operation used by `zipWith` is strict or not. The lazy (by-name) is more general as it allows using efficiently functions that ignore the first (or the second) operand if the other one is a special case (so if you zip with || or &&).

Some of the test cases for REPL listed above can be used here again. This is a good test case for `zipWith` in REPL:

```
naturals
  .zipWith[Int, Int] (_+_) (naturals)
  .take (2000000000)
  .take (20)
  .toList
```

What should be the result of this?

```
naturals
  .map { _%2==0 }
  .zipWith[Boolean,Boolean] (_||_) (naturals.map { _ % 2 == 1 })
  .take(10)
  .toList
```

Convince yourself what the results of these test cases should be before you run the code.

[8]Exercise 5.11 [Pilquist, Chiusano, Bjarnason, 2022]
[9]Exercise 5.12 [Pilquist, Chiusano, Bjarnason, 2022]
[10]Exercise 5.13 [Pilquist, Chiusano, Bjarnason, 2022]