030: Lazy Streams

This set of exercises of this lecture is to be found in the "030-streams" folder of our reporepository. The following three files are the most relevant for the homework.

- Stream.scala, located in src/main/scala/adpro
- Stream_test.scala, located in src/test/scala/adpro
- Main.scala, located in src/main/scala/adpro

Instructions You are expected to only hand in Stream.scala to learnIt. The file contains predefined classes and functions that you can use for your exercises. Your solutions to the exercises should be put below the comment lines "//Exercise X." Function interfaces are provided for some, but not all, of the exercises. The answer to the first exercise has been provided in Stream.scala, below "//Exercise 1". It is to be used as a sanity check for your environments. First change directory to 030-streams. Then, launching sbt should get you into the interactive mode. Type compile and you should see all the scala files compiled successfully; type run and you will run Main.scala and see results printed on the terminal; at last, type test should pass the tests we have written for Exercise 1 but fail for unimplemented parts.

You are (strongly) encouraged to test your code. Use Stream_test.scala for unit testing. Tests for the first three exercises are provided as examples. You may consider test-driven development (TDD), writing tests before coding solutions. Alternatively, use Main.scala, which is akin to Java's main, for printing out your computation. Note: Do not hand in your tests; they are for improving your coding practice.

Exercise 1. Define functions from and to that generate streams of natural numbers above (and below) a given natural number.

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

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

Exercise 2. Write a function to convert a Stream to a List, which will force its evaluation and let you look at it in the REPL. You can convert to the regular List type in the standard library. You can place this and other functions that operate on a Stream inside the Stream trait.

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

Test this function using the factory of streams to build finite streams and converting the to lists (to see whether they yield expected lists). Then create a few finite streams of integers using to (n) from the previous exercise, and convert them to lists.¹

Exercise 3. Write the function take(n) for returning the first n elements of a Stream, and drop(n) for skipping the first n elements of a Stream.

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

Try the following test case (should terminate with no memory exceptions and very fast). Why? naturals.take(100000000).drop(41).take(10).toList (this way we also test the function from(n), from the first exercise)²

¹Exercise 5.1 [Chiusano, Bjarnason 2014]

²Exercise 5.2 [Chiusano, Bjarnason 2014]

Exercise 4. Write the function takeWhile (p) for returning all starting elements of a Stream that match the given predicate p.

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

Test your implementation on the following test case:

```
naturals.takeWhile.(_<1000000000).drop(100).take(50).toList</pre>
```

It should terminate very fast, with no exceptions thrown. Why?³

Exercise 5. Implement for All (p), which checks that all elements in this Stream satisfy a given predicate. Terminate the traversal as soon as it encounters a non-matching value.

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

Use the following test cases for forAll:

```
This should succeed: naturals.forAll (_ < 0)
This should crash: naturals.forAll (_ >=0). Explain why.
```

Recall that exists has already been implemented before (in the book). Both forAll and exists are a bit strange for infinite streams; 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 streams. Why?⁴

Exercise 6. Use foldRight to implement takeWhile. Reuse the test case from Exercise 4.5

Exercise 7. Implement headOption using foldRight. Devise a couple of suitable test cases using infinite streams. you can reuse naturals.⁶

Exercise 8. Implement the following functions. The task involves designing their types.

Implement map, filter, append, and flatMap using foldRight. The append method should be non-strict in its argument.⁷

- 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
```

³Exercise 5.3 [Chiusano, Bjarnason 2014]

⁴Exercise 5.4 [Chiusano, Bjarnason 2014]

⁵Exercise 5.5 [Chiusano, Bjarnason 2014]

⁶Exercise 5.6 [Chiusano, Bjarnason 2014]

⁷Exercise 5.7 [Chiusano, Bjarnason 2014]

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 streams and would not be optimal for lists.

Exercise 10. Compute a lazy stream 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 be translating to List a finite prefix of fibs, or a finite prefix of an infinite suffix.⁸

Exercise 11. Write a more general stream-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 stream.

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

Test this function by unfolding the stream of natural numbers and checking whether its finite prefix is equal to the corresponding prefix of naturals.⁹

Exercise 12. Write fib and from in terms of unfold. Use these test cases:

```
from(1).take(1000000000).drop (41).take(10).toList ==
from1(1).take(1000000000).drop (41).take(10).toList and
fibs1.take(100).toList ==fibs.take(100).toList,
```

where identifiers suffixed with 1 refer to the new versions of the functions.¹⁰

Exercise 13. Use unfold to implement map, take, takeWhile, and zipWith. 11

You can reuse test-cases from earlier exercises, or devise new ones. Remember to test with infinite streams. Infinite streams are a good way whether your implementations are indeed non-strict.

This is a good test case for zipWith:

```
naturals.zipWith[Int,Int] (_+_) (naturals).take(200000000).take(20).toList
```

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 &&). On the other hand, I experienced some trouble using strict functions in this context. You can choose yourself, what you implement.

What should be the result of this?

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

Don't get tricked into just running this and seeing the result. There might be a bug in your implementation, so convince yourself, what the results of these two test cases should be.

⁸Exercise 5.10 [Chiusano, Bjarnason 2014]

⁹Exercise 5.11 [Chiusano, Bjarnason 2014]

¹⁰Exercise 5.12 [Chiusano, Bjarnason 2014]

¹¹Exercise 5.13 [Chiusano, Bjarnason 2014]