# Exercise sheet 04— Solutions

## 1 Old friends

The functions `map`, `filter`, `flatMap` and `append` for `LazyList` implemented with `foldRight` do not differ much from their implementations for `List`:

```
def map[B](f: A => B): LazyList[B] =
  foldRight(empty[B])((h, t) => cons(f(h), t))
```

```
def filter(p: A => Boolean): LazyList[A] =
  foldRight(empty[A])((h, t) => if p(h) then cons(h, t) else t)
```

```
def flatMap[B](f: A => LazyList[B]): LazyList[B] =
  foldRight(empty[B])((h, t) => f(h).append(t))
```

```
def append[B >: A](b: => LazyList[B]): LazyList[B] =
  foldRight(b)(cons(_, _))
```

The different behaviour we get from laziness comes from the use of the smart constructors like `cons` and our non-strict implementation of `foldRight` from the lecture.

## 2 takeWhile

### 2.1 via Pattern Matching

```
def takeWhile(p: A => Boolean): LazyList[A] = this match
  case Cons(h, t) if p(h()) => cons(h(), t().takeWhile(p))
  case _                    => Empty
```

Basically `takeWhile` works like `take`, but instead of counting down a number we use the given condition to check if we should take further elements.

### 2.2 via `foldRight`

```
def takeWhileViaFoldRight(p: A => Boolean): LazyList[A] =
  foldRight(empty[A])((h, t) => {
    if p(h) then cons(h, t)
    else empty
  })
```

Our z parameter for `foldRight` is the end of the list. In the function we pass in, we get the `LazyList`'s head as well as the list of results, to which we prepend the element if it fulfills the

condition. If the condition is not fulfilled, we return `empty` directly. As `t` is not used in this branch, this ends the recursion.

### 2.3 via unfold

As a reminder, this is our implementation of `unfold` from the lecture:

```
def unfold[A, S](z: S)(f: S => Option[(A, S)]): LazyList[A] =
  f(z) match
    case Some((a, s)) => cons(a, unfold(s)(f))
    case None => empty
```

With `unfold` we can generate a `LazyList` from a starting value. As we want to generate a `LazyList` with values from the one we already have, we use the following idea: we take the current list as starting value, check the given condition for it, and if it is fulfilled, we return a `Some`. This generates an element for the result. We set the new startinv value to the tail of the list, so we iterate through it. If the condition does not hold for an element or we reach the end of the list, we return `None` to end the recursion.

```
def takeWhileViaUnfold(p: A => Boolean): LazyList[A] = unfold(this){
  case Cons(h, t) if p(h()) => Some((h(), t()))
  case _                    => None
}
```

## 3 tails

Like in the previous exercise, we want to use `unfold` to generate a list from parts of our previous list. But this time, we want all suffixes, i.e. all bisherigen Liste besteht. Jedoch wollen wir diesmal alle Suffixe, d.h. alle Teillisten, die man durch Entfernen von 0 oder mehr Elementen vom Anfang der Liste erhalten kann. Hier gibt es diverse Möglichkeiten, dies umzusetzen. Zwei davon stellen wir vor.

```
def tails: LazyList[LazyList[A]] = unfold(this) {
    case Empty => None
    case s => Some((s, s.drop(1)))
  }.append(LazyList(empty))
```

In the first variant, we begin our `unfold` with the List as starting value. As the first element of our result list is the list itself (0 elements removed), we return it in the function passed to unfold and set the tail as next starting value. As we did not implement `tail` in the lecture, we use the equivalent `drop(1)`. When we reach the end of the list, we return `None` to end the recursion. Because of how we defined `unfold`, we cannot return an element in this last step, so we append the last element, the empty list, with `append`. Important `append` expects a list of type `LazyList[LazyList[A]]`. If we simply wrote `empty` instead of `LazyList(empty)`, it would still be valid but append nothing.

```
def tailsAlt: LazyList[LazyList[A]] = LazyList(this).append(unfold(this){
    case Empty => None
    case Cons(_, t) => Some((t(), t()))
  })
```

In the second variant, instead of returning the current partial list, we return its tail by matching on the `Cons`. This way we don't miss the empty list in the end, but we do miss the full list in the beginning. So we append the result of our `unfold` call to a list, to which we added our starting list.