Exercise sheet 02— Solutions

1 dropWhile

We want to remove elements from the front of the list, until `f` returns `false`. To do that we recurse through the list, similar to `drop`. But in the `match` we have to bind the head value to a variable too, not only the tail:

```
def dropWhile(f: A => Boolean): List[A] =
  this match
    case Cons(x, xs) =>
      if f(x) then xs.dropWhile(f)
      else this
    case Nil => this
```

If our list still has an element, we take its value and pass it to the function `f`. If the function returns true, we call `dropWhile` recursively on the tail of the list, otherwise we are done and return the current list unmodified (same with an empty list).

Scala also allows to add a condition to a `case` by adding `if <boolean expression>` between the pattern and the arrow:

```
def dropWhile(f: A => Boolean): List[A] =
  this match
    case Cons(a, as) if f(a) => as.dropWhile(f)
    case _                   => this
```

This is called a *pattern guard*, it causes the case to only match, if the given condition is also fulfilled. But this has the downside, that the compiler can no longer check if our cases are exhaustive, i.e. every possible case is matched.

2 Folds — step by step

Reminder:

```
@annotation.tailrec
final def foldLeft[B](z: B)(f: (B, A) => B): B = this match
  case Nil         => z
  case Cons(a, as) => as.foldLeft(f(z, a))(f)
```

a) `foldLeft` and addition

```
Cons(1, Cons(2, Cons(3, Nil))).foldLeft(0)((x,y) => x + y)
Cons(2, Cons(3, Nil)).foldLeft(0 + 1)((x,y) => x + y)
Cons(3, Nil).foldLeft(1 + 2)((x,y) => x + y)
Nil.foldLeft(3 + 3)((x,y) => x + y)
6
```

b) Incrementing every element of a list by 1 using `foldRight`

```
Cons(1, Cons(2, Nil)).foldRight(Nil: List[Int])((h, t) => Cons(h + 1, t))
Cons(2, Cons(2, Nil).foldRight(Nil: List[Int])((h, t) => Cons(h + 1, t)))
Cons(2, Cons(3, Nil.foldRight(Nil: List[Int])((h, t) => Cons(h + 1, t))))
Cons(2, Cons(3, Nil: List[Int]))
```

3 reverse

Implementation of `reverse` via `foldLeft`

```
def reverse: List[A] =
  foldLeft(Nil: List[A])((b, a) => Cons(a, b))
```

We start with an empty list. As `foldLeft` combines the first element of the list with the start value first (in contrast to `foldRight`), this element is also prepended in front of the end of the list first $\implies$ the order is reversed.

4 append

Implementation of `append` via foldRight:

```
def append[AA >: A](r: List[AA]): List[AA] =
  foldRight(r)((a, as) => Cons(a, as))
```

In the lecture we saw that `foldRight` with a start value of `Nil` and the Cons constructor as operator results in the input list, as we append all values in front of the start value. To make this into list concatenation, we only have to replace the start value with the list in the back: we now prepend all elements from the front list to that one instead of a `Nil`.

5 map, filter and flatMap

map: Nearly identical to the foldRight call, that keeps the list unmodified, only the function `f` is called on each value.

```
def map[B](f: A => B): List[B] =
  foldRight(Nil: List[B])((a, as) => Cons(f(a), as))
```

filter: In the function passed to the fold, we test if `p` is true for the current element. If yes, we prepend it to the accumulator as in the other exercises. If not, we return the previous accumulator.

```
def filter(p: A => Boolean): List[A] =
  foldRight(Nil: List[A])((a, as) => if p(a) then Cons(a, as) else as)
```

flatMap: The passed function returns a new list for every element of the input list. Instead of prepending the result with a Cons (which would give us a `List[List[B]]`), we use `append` to prepend the list element by element.

```
def flatMap[B](f: A => List[B]): List[B] =
  foldRight(Nil: List[B])((a, as) => f(a).append(as))
```

Excourse: stacksafe map, filter and flatMap

Problem: our implementation of `foldRight` is not tail recursive, i.e. it is not *stack-safe* (a long list results in large recursion depth, we can get a StackOverflowError).

But we implemented `map`, `filter` and `flatMap` via `foldRight`. `foldLeft` is stack safe, but is semantically different regarding the order, in which elements are combined. A solution to this problem is to implement `foldRight` using `foldLeft` by reversing the list with `reverse` first:

```
def foldRight[B](z: B)(f: (A, B) => B): B = reverse.foldLeft(z)((b, a) => f(a, b))
```

The implementation of `foldRight` on `List` in the standard library uses this trick.

6 zip

a) zipAdd

We want to pairwise add elements of two lists, i.e. the elements with same index. If the lists aren't the same length, we stop when one list ends (so the resulting list always is the same length as the shorter input list).

```
def zipAdd(a: List[Int], b: List[Int]): List[Int] =
  (a, b) match
    case (Nil, _) | (_, Nil)          => Nil
    case (Cons(ah, at), Cons(bh, bt)) => Cons(ah + bh, zipAdd(at, bt))
```

As we want to recurse through both lists together, we combine them into a tuple and match on both at the same time (a nested pattern match would also be possible, but more confusing).

In this solution we also see, that multiple patterns can be combined with `|`. This is only possible if the patterns don't use unbound variables (underscores are allowed). The result is the same as writing the patterns in separate `case` lines with the same expression on the right-hand-side. `(Nil, _) | (_, Nil)` matches if and only if we reached the end of at least one of the two lists. Therefore we can match a `Cons` in both lists in the second `case` and get the heads and tails of the lists. We add the heads together and call `zipAdd` recursively on the tails.

As our method only works with lists with an element type of `Int`, we implement it on the companion object. Having it directly on the enum is possible with Givens, which we will only handle later in the lecture. Here is such an implementation anyways (not relevant for the test):

```
def zipAdd[AA >: A](l: List[AA])(using num: Numeric[AA]): List[AA] =
  import num.*
  (this, l) match
    case (Nil, _) | (_, Nil)          => Nil
    case (Cons(ah, at), Cons(bh, bt)) => Cons(ah + bh, at.zipAdd(bt))
```

The method's body barely changes, but the signature is a good deal more complicated. Without going too much into Givens and using clauses, you can think of `using num: Numeric[AA]` as a request to the compiler, to check for an implementation of numeric operations for the type `AA` and provide them to us (so we can use the plus operator). The parameter `AA` has to be a supertype of the element types in our list and the passed second list (similar to e.g. `append`). `Numeric` is a so called typeclass, we will see those in more detail in a later lecture.

b) zipWith

We generalize `zipAdd` for arbitrary element types and operations. This time we can provide the implementation on the `List` enum without typeclasses, as we don't need to know anything about the element type.

Two type parameters `B` and `R` represent the element type of the second, passed list and the resulting list. The type `A` is already given by the list on which we call `zipWith`. We also add a parameter for the function, that takes one of each element type and combines them into the result type.

Now we only need to replace the plus with calling `f`.

```scala
def zipWith[AA >: A,B](l: List[AA])(f: (A, AA) => B): List[B] =
  (this, l) match
    case (Nil, _) | (_, Nil)          => Nil
    case (Cons(a, as), Cons(b, bs))   => Cons(f(a, b), as.zipWith(bs)(f))
```

The following solution goes a step further and makes the implementation tail recursive. For that we need a helper function. We use the same trick we used in the lecture already, building our result backwards and then reversing it:

```scala
def zipWithTailrec[B,R](l: List[B])(f: (A, B) => R): List[R] =
  def go(agg: List[R])(a: List[A], b: List[B]): List[R] =
    (a, b) match
      case (Nil, _) | (_, Nil)          => agg
      case (Cons(ah, at), Cons(bh, bt)) => go(Cons(f(ah, bh), agg))(at, bt)
  go(Nil)(this, l).reverse
```

7 Parametricity

`curry` converts a function with two parameters into a function with one parameter returning another function taking the only the second parameter:

```
def curry[A,B,C](f: (A,B) => C): A => B => C =
  a => b => f(a, b)
```

To get to the implementation, follow the types, starting with the return type:

- `A => B => C` is equivalent to `A => (B => C)`.
- So our return type is a function, which takes a parameter of type `A`. We start with that one:
  `(a: A) =>`
- The return type of that function is also a function, taking a parameter of type `B`:
  `(a: A) => (b: B) =>`
- This function should return a value of type `C`. Our only way to get such a value is the function `f`: `(a: A) => (b: B) => f(a,b)`
- As the types can be inferred from the return type of the function unambiguously, we can leave them off, the compiler will fill them in.

`uncurry` as the inverse function to `curry` works pretty similarly:

```
def uncurry[A,B,C](f: A => B => C): (A, B) => C =
  (a, b) => f(a)(b)
```

As our return type is a function with two parameters, we start with the corresponding parameter list for the anonymous function. Then we look at how we can get a value matching the expected return type from the passed function `f`.