A Closer Look at Lists

List Methods

Sublists and element access:

- ***xs.lenght*** – the number of elements of xs
- ***xs.last*** – the list's last element, exception if xs is empty
- ***xs.init*** – a list consisting of all elements of xs except the last one, exception if xs is empty
- ***xs.take(n)*** – a list consisting of the first n elements of xs or xs itself if it is shorter than n
- ***xs.drop(n)*** – the rest of the collection after taking n elements
- ***xs(n)*** – the element of xs at index n, and if the index is out of the range of the list, you get an exception

Creating new lists:

- ***xs ++ ys*** – the list consisting of all elements of xs followed by all elements of ys
- ***xs.reverse*** – the list containing the elements of xs in reversed order
- ***xs.updated(n, x)*** – the list containing the same elements as xs, except at index n, where it contains x

Finding elements:

- ***xs.indexOf(x)*** – the index of the first element in xs equal to x, or -1 if x does not appear in xs
- ***xs.contains(x)*** – same as xs.indexOf(x) >= 0

Implementation of Last

- The complexity of head is small and constant in terms of time
- last, takes steps proportional to the length of the list xs, as it results from the following possible implementation of last:

```
def last[T](xs: List[T]): T = xs match
    case List() => throw Error("last of empty list")
    case List(x) => x
    case y :: ys => last(ys)
```

Exercise: *Implement* ***init*** *as an external function, analogous to last:*

```
def init[T](xs: List[T]): List[T] = xs match
  case List() => throw Error("init of empty list")
  case List(x) => List()
  case y :: ys => y :: init(ys)
```

Implementation of Concatenation

- A possible implementation for concatenation would be:

```
extension [T](xs: List[T])
  def ++(ys: List[T]): List[T] = xs match
      case Nil => ys
      case x :: xs1 => x :: (xs1 ++ ys)
```

- The complexity of this function is O(xs.length)

Implementation of Reverse

- A possible implementation for reverse would be:

```
extension [T](xs: List[T])
    def reverse: List[T] = xs match
        case Nil => Nil
        case y :: ys => ys.reverse ++ List(y)
```

- The complexity of this function is O(xs.length * xs.length)

Exercise: *Remove the n^{th} element of a list xs. If n is out of bounds, return xs itself.*

```
def removeAt[T](n: Int, xs: List[T]) = xs match
  case Nil => Nil
  case y :: ys => {
      if (n == 0)
        ys
      else
        y :: removeAt(n – 1, ys)
  }
```

Exercise: *Flatten a list structure.*

```
def flatten(xs: Any): List[Any] = xs match
  case Nil => Nil
  case y :: ys => flatten(y) ++ flatten(ys)
  case _ => xs :: Nil
```