Higher-Order List Functions

Recurring Patterns for Computations on Lists

- The examples have shown that functions on lists often have similar structures
- We can identify several recurring patterns, like:
  - Transforming each element in a list in a certain way
  - Retrieving a list of all elements satisfying a criterion
  - Combining the elements of a list using an operator
- Functional languages allow programmers to write generic functions that implement patterns such as these using higher-order functions

Mapping

- A common operation is to transform each element of a list and then return the list of results
- This operation can be generalized to the method ***map*** of the List class
- A simple way to define map is as follows:

```
extension [T](xs: List[T])
    def map[U](f: T => U): List[U] = xs match
        case Nil => xs
        case x :: xs => f(x) :: xs.map(f)
```

- The actual definition is more complicated; it uses tail-recursion and works for arbitrary collections, not just lists

Exercise: *Consider a function to square each element of a list and return the result. Complete the two following equivalent definitions of* ***squareList****.*

```
def squareList(xs: List[Int]): List[Int] = xs match
  case Nil => Nil
  case y :: ys => (y * y) :: squareList(ys)

def squareList(xs: List[Int]): List[Int] =
   xs.map(x => (x * x))
```

Filtering

- Another common operation on lists is the selection of all elements satisfying a given condition
- This pattern is generalized by the method filter of the List class:

```
extension [T](xs: List[T])
    def filter(p: T => Boolean): List[T] = xs match
        case Nil => Nil
        case x :: xs => if (p(x)) x :: xs.filter(p) else xs.filter(p)
```

Variations of Filter

- Besides filter, there are also the following methods that extract sublists based on a predicate:
    - ***xs.filterNot(p)*** – same as ***xs.filter(x => !p(x))***; returns the list consisting of those elements of xs that do not satisfy the predicate p
    - ***xs.partition(p)*** – same as ***(xs.filter(p), xs.filterNot(p))***, but computed in a single traversal of the list xs
    - ***xs.takeWhile(p)*** – the longest prefix of list xs consisting of elements that all satisfy the predicate p
    - ***xs.dropWhile(p)*** – the remainder of the list xs after any leading elements satisfying p have been removed
    - ***xs.span(p)*** – same as ***(xs.takeWhile(p), xs.dropWhile(p))*** but computed in a single traversal of the list xs

Exercise: *Write a function that packs consecutive duplicates of lists elements into sublists.*

```
def pack[T](xs: List[T]): List[List[T]] = xs match
    case Nil => Nil
    case x :: xs1 => {
        val (duplicates, rest) = xs1.span(y => y == x)
        (x :: duplicates) :: pack(rest)
     }
```

Exercise: *Using pack, write a function encode that produces the run-length encoding of a list.*

```
def encode[T](xs: List[T]): List[(T, Int)] =
   pack(xs).map(x => (x.head, x.length))
```