Higher-Order List Functions

Principles of Functional Programming

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.

Applying a Function to Elements of a List

A common operation is to transform each element of a list and then return the list of results.

For example, to multiply each element of a list by the same factor, you could write:

```
def scaleList(xs: List[Double], factor: Double): List[Double] = xs match
  case Nil     => xs
  case y :: ys => y * factor :: scaleList(ys, factor)
```

Mapping

This scheme 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)
```

(in fact, the actual definition of `map` is a bit more complicated, because it is tail-recursive, and also because it works for arbitrary collections, not just lists).

Using `map`, `scaleList` can be written more concisely.

```
def scaleList(xs: List[Double], factor: Double) =
  xs.map(x => x * factor)
```

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     => ???
  case y :: ys => ???

def squareList(xs: List[Int]): List[Int] =
  xs.map(???)
```

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. For example:

```
def posElems(xs: List[Int]): List[Int] = xs match
  case Nil     => xs
  case y :: ys => if y > 0 then y :: posElems(ys) else posElems(ys)
```

Filter

This pattern is generalized by the method `filter` of the `List` class:

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

Using `filter`, `posElems` can be written more concisely.

```
def posElems(xs: List[Int]): List[Int] =
  xs.filter(x => x > 0)
```

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))`; 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 pack that packs consecutive duplicates of list elements into sublists. For instance,

```
pack(List("a", "a", "a", "b", "c", "c", "a"))
```

should give

```
List(List("a", "a", "a"), List("b"), List("c", "c"), List("a")).
```

You can use the following template:

```
def pack[T](xs: List[T]): List[List[T]] = xs match
  case Nil      => Nil
  case x :: xs1 => ???
```

Exercise

Write a function pack that packs consecutive duplicates of list elements into sublists. For instance,

```
pack(List("a", "a", "a", "b", "c", "c", "a"))
```

should give

```
List(List("a", "a", "a"), List("b"), List("c", "c"), List("a")).
```

You can use the following template:

```
def pack[T](xs: List[T]): List[List[T]] = xs match
  case Nil      => ???
  case x :: xs1 => ???
```

Exercise

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

The idea is to encode `n` consecutive duplicates of an element `x` as a pair `(x, n)`. For instance,

```
encode(List("a", "a", "a", "b", "c", "c", "a"))
```

should give

```
List(("a", 3), ("b", 1), ("c", 2), ("a", 1)).
```

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)] = ???
```