香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Principles of Programming Languages (Lecture 6)

COMP 3031, Fall 2025

Lionel Parreaux

Tuples and Generic Methods

Sorting Lists Faster

As a non-trivial example, let's design a function to sort lists that is more efficient than insertion sort.

A good algorithm for this is *merge sort*. The idea is as follows:

If the list consists of zero or one elements, it is already sorted.

Otherwise,

- Separate the list into two sub-lists, each containing around half of the elements of the original list.
- Sort the two sub-lists.
- Merge the two sorted sub-lists into a single sorted list.

First MergeSort Implementation

Here is the implementation of that algorithm in Scala:

```scala
def msort(xs: List[Int]): List[Int] =
  val n = xs.length / 2
  if n == 0 then xs
  else
    def merge(xs: List[Int], ys: List[Int]) = ???
    val (fst, snd) = xs.splitAt(n)
    merge(msort(fst), msort(snd))
```

The SplitAt Function

The `splitAt` function on lists returns two sublists

- the elements up the the given index
- the elements from that index

The lists are returned in a *pair*.

Detour: Pair and Tuples

The pair consisting of `x` and `y` is written `(x, y)` in Scala.

Example

```
val pair = ("answer", 42)
```
> *pair : (String, Int)*

The type of `pair` above is `(String, Int)`.

Pairs can also be used as patterns:

```
val (label, value) = pair
```
> *label: String; value: Int*

This works analogously for tuples with more than two elements.

Example Use of Pairs

We can define `splitAt` as follows...

```
extension [A](xs: List[A])


  def splitAt(n: Int): (List[A], List[A]) =
```

Example Use of Pairs

We can define `splitAt` as follows...

```
extension [A](xs: List[A])

  def splitAt(n: Int): (List[A], List[A]) =
    (xs.take(n), xs.drop(n))
```

Translation of Tuples

For small(*) n, the tuple type $(T_1, ..., T_n)$ is an abbreviation of the parameterized type

$$\texttt{scala.Tuple}n[T_1, ..., T_n]$$

A tuple expression $(e_1, ..., e_n)$ is equivalent to the function application

$$\texttt{scala.Tuple}n(e_1, ..., e_n)$$

A tuple pattern $(p_1, ..., p_n)$ is equivalent to the constructor pattern

$$\texttt{scala.Tuple}n(p_1, ..., p_n)$$

(*) Currently, “small” = up to 22. There’s also a TupleXXL class that handles Tuples larger than that limit.

The Tuple class

Here, all `Tuple`*n* classes are modeled after the following pattern:

```
case class Tuple2[+T1, +T2](_1: T1, _2: T2):
  override def toString = "(" + _1 + "," + _2 +")"
```

The fields of a tuple can be accessed with names _1, _2, ...

So instead of the pattern binding

```
val (label, value) = pair
```

one could also have written:

```
val label = pair._1
val value = pair._2
```

But the pattern matching form is generally preferred.

Definition of Merge

Here is a definition of the `merge` function:

```
def merge(xs: List[Int], ys: List[Int]) = (xs, ys) match
  case (Nil, ys) => ys
  case (xs, Nil) => xs
  case (x :: xs1, y :: ys1) =>
    if x < y
    then x :: merge(xs1, ys)
    else y :: merge(xs, ys1)
```

Note how this programming style is hard to get wrong:
the compiler will warn us if we forget a case!

(Compare with traditional index-based imperative loops.)

Definition of Merge

Here is a definition of the `merge` function:

```
def merge(xs: List[Int], ys: List[Int]) = (xs, ys) match
  case (Nil, ys) => ys
  case (xs, Nil) => xs
  case (x :: xs1, y :: ys1) =>
    if x < y
    then x :: merge(xs1, ys)
    else y :: merge(xs, ys1)
```

Note how this programming style is hard to get wrong:
the compiler will warn us if we forget a case!

(Compare with traditional index-based imperative loops.)

Making Sort More General

Problem: How to parameterize `msort` so that it can also be used for lists with elements other than `Int`?

```
def msort[T](xs: List[T]): List[T] = ???
```

does not work, because the comparison < in `merge` is not defined for arbitrary types `T`.

Idea: Parameterize `merge` with the necessary comparison function.

Parameterization of Sort

The most flexible design is to make the function `sort` polymorphic and to pass the comparison operation as an additional parameter:

```
def msort[T](xs: List[T])(lt: (T, T) => Boolean) =
  ...
    merge(msort(fst)(lt), msort(snd)(lt))
```

Merge then needs to be adapted as follows:

```
def merge[T](xs: List[T], ys: List[T]) = (xs, ys) match
  ...
  case (x :: xs1, y :: ys1) =>
    if lt(x, y) then ...
    else ...
```

Important Note: Syntax and Semantics of Pair-Taking Functions

In Scala,

- `f: (T, T) => Boolean`
  is the type of a function that takes *two* parameters
- `g: ((T, T)) => Boolean`
  is the type of a function that takes *one* parameters of *pair type*
- i.e., the latter is syntax sugar for `g: (Tuple2[T, T]) => Boolean`

However, Scala often (but *not always*) glosses over the difference...

```
g((0, 1)) // the formally correct way of calling g
g(0, 1)   // also compiles
f(0, 1)   // the formally correct way of calling f
f((0, 1)) // does NOT compile (missing argument)
```

Important Note: Syntax and Semantics of Pair-Taking Functions

In Scala,

- `f: (T, T) => Boolean`
  is the type of a function that takes *two* parameters
- `g: ((T, T)) => Boolean`
  is the type of a function that takes *one* parameters of *pair type*
- i.e., the latter is syntax sugar for `g: (Tuple2[T, T]) => Boolean`

However, Scala often (but *not always*) glosses over the difference...

```
g((0, 1)) // the formally correct way of calling g
g(0, 1)   // also compiles
f(0, 1)   // the formally correct way of calling f
f((0, 1)) // does NOT compile (missing argument)
```

Calling Parameterized Sort

We can now call `msort` as follows:

```
val xs = List(-5, 6, 3, 2, 7)
val fruits = List("apple", "pear", "orange", "pineapple")

msort(xs)((x: Int, y: Int) => x < y)
msort(fruits)((x: String, y: String) => x.compareTo(y) < 0)
```

Or, since parameter types can be inferred from the call `msort(xs)`:

```
msort(xs)((x, y) => x < y)
```

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.

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      => Nil
    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`, we can rewrite `scaleList` 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      => xs
    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       => ???
  case x :: xs1 => ???
```

Exercise

Write a function `pack` that packs consecutive duplicates of list elements into sublists.

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

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)] =
```

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(ys => (ys.head, ys.length))
```

But you told us to avoid using head*!* True, and there is a type-safe way:

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

Type ::[T] represents *non-empty lists*, on which calling head is safe.

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(ys => (ys.head, ys.length))
```

But you told us to avoid using head*!* True, and there is a type-safe way:

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

Type ::[T] represents *non-empty lists*, on which calling head is safe.

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(ys => (ys.head, ys.length))
```

But you told us to avoid using `head`*!* True, and there is a type-safe way:

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

Type `::[T]` represents *non-empty lists*, on which calling `head` is safe.

Reduction of Lists

Reduction of Lists

Another common operation on lists is to combine the elements of a list using a given operator.

For example:

```
sum(List(x1, ..., xn))          =  0 + x1 + ... + xn
product(List(x1, ..., xn))      =  1 * x1 * ... * xn
```

We can implement this with the usual recursive schema:

```
def sum(xs: List[Int]): Int = xs match
  case Nil      => 0
  case y :: ys => y + sum(ys)
```

ReduceLeft

This pattern can be abstracted out using the generic method `reduceLeft`:

`reduceLeft` inserts a given binary operator between adjacent elements:

```
List(x1, ..., xn).reduceLeft((x, y) => x.op(y))  =  x1.op(x2). ... .op(xn)
```

Using `reduceLeft`, we can simplify:

```
def sum(xs: List[Int])     = (0 :: xs).reduceLeft((x, y) => x + y)
def product(xs: List[Int]) = (1 :: xs).reduceLeft((x, y) => x * y)
```

ReduceLeft

This pattern can be abstracted out using the generic method `reduceLeft`:

`reduceLeft` inserts a given binary operator between adjacent elements:

```
List(x1, ..., xn).reduceLeft((x, y) => x.op(y))  =  x1.op(x2). ... .op(xn)
```

Using `reduceLeft`, we can simplify:

```
def sum(xs: List[Int])     = (0 :: xs).reduceLeft((x, y) => x + y)
def product(xs: List[Int]) = (1 :: xs).reduceLeft((x, y) => x * y)
```

Caution: `reduceLeft` function only works on non-empty lists!

A Shorter Way to Write Simple Functions

Instead of `((x, y) => x * y))`, one can also write the shorter:

```
(_ * _)
```

Every `_` represents a new parameter, going from left to right.

The parameters are defined at the next outer pair of parentheses (or the whole expression if there are no enclosing parentheses).

So `sum` and `product` can also be expressed like this:

```
def sum(xs: List[Int])     = (0 :: xs).reduceLeft(_ + _)
def product(xs: List[Int]) = (1 :: xs).reduceLeft(_ * _)
```

FoldLeft

`reduceLeft` is defined in terms of a more general function, `foldLeft`.

`foldLeft` is like `reduceLeft` but takes an *accumulator*, `z`, as an additional parameter, which is returned when `foldLeft` is called on an empty list.

```
List(x1, ..., xn).foldLeft(z)(_.op(_))  =  z.op(x1).op ... .op(xn)
```

So `sum` and `product` can also be defined as follows:

```
def sum(xs: List[Int])      =  xs.foldLeft(0)(_ + _)
def product(xs: List[Int])  =  xs.foldLeft(1)(_ * _)
```

Implementations of ReduceLeft and FoldLeft

`foldLeft` and `reduceLeft` can be implemented in class `List` as follows.

```
sealed abstract class List[T]:

  def reduceLeft(op: (T, T) => T): T = this match
    case Nil     => throw IllegalOperationException("Nil.reduceLeft")
    case x :: xs => xs.foldLeft(x)(op)

  def foldLeft[U](z: U)(op: (U, T) => U): U = this match
    case Nil     => z
    case x :: xs => xs.foldLeft(op(z, x))(op)
```

FoldRight and ReduceRight

Uses of `foldLeft` and `reduceLeft` unfold on trees that lean to the left.

They have two dual functions, `foldRight` and `reduceRight`, which produce trees which lean to the right, i.e.,

```
List(x1, ..., x{n-1}, xn).reduceRight(op)
  ==
x1.op(x2.op( ... x{n-1}.op(xn) ... ))

List(x1, ..., xn).foldRight(z)(op )
  ==
x1.op(x2.op( ... xn.op(z) ... ))
```

Implementation of FoldRight and ReduceRight

They are defined as follows

```
def reduceRight(op: (T, T) => T): T = this match
  case Nil      => throw UnsupportedOperationException(”Nil.reduceRight”)
  case x :: Nil => x
  case x :: xs  => op(x, xs.reduceRight(op))

def foldRight[U](z: U)(op: (T, U) => U): U = this match
  case Nil     => z
  case x :: xs => op(x, xs.foldRight(z)(op))
```

Difference between FoldLeft and FoldRight

For operators that are associative and commutative, `foldLeft` and `foldRight` are equivalent (though there may be a difference in efficiency).

But sometimes, only one of the two operators is appropriate.

Exercise

Here is another formulation of `concat`:

```
def concat[T](xs: List[T], ys: List[T]): List[T] =
  xs.foldRight(ys)(_ :: _)
```

Here, it isn't possible to replace `foldRight` by `foldLeft`. Why?

```
O      The types would not work out
O      The resulting function would not terminate
O      The result would be reversed
```

Exercise

Here is another formulation of `concat`:

```
def concat[T](xs: List[T], ys: List[T]): List[T] =
  xs.foldRight(ys)(_ :: _)
```

Here, it isn't possible to replace `foldRight` by `foldLeft`. Why?

```
X      The types would not work out
O      The resulting function would not terminate
O      The result would be reversed
```

Back to Reversing Lists

We now develop a function for reversing lists which has a linear cost.

The idea is to use the operation `foldLeft`:

```
def reverse[T](xs: List[T]): List[T] = xs.foldLeft(z?)(op?)
```

All that remains is to replace the parts `z?` and `op?`.

Let's try to *calculate* them from examples.

Deduction of Reverse (1)

To start calculating `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can calculate as follows:

```
Nil
```

Deduction of Reverse (1)

To start calculating `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can calculate as follows:

```
  Nil

=   reverse(Nil)
```

Deduction of Reverse (1)

To start calculating `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can calculate as follows:

```
  Nil

=   reverse(Nil)

=   Nil.foldLeft(z?)(op)
```

Deduction of Reverse (1)

To start calculating `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can calculate as follows:

```
  Nil

=   reverse(Nil)

=   Nil.foldLeft(z?)(op)

=   z?
```

Consequently, `z? = Nil`

Deduction of Reverse (2)

We still need to calculate `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
x :: Nil
```

Deduction of Reverse (2)

We still need to calculate `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
x :: Nil

=   reverse(x :: Nil)
```

Deduction of Reverse (2)

We still need to calculate `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
x :: Nil

=   reverse(x :: Nil)

=   (x :: Nil).foldLeft(Nil)(op?)
```

Deduction of Reverse (2)

We still need to calculate `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
x :: Nil

=   reverse(x :: Nil)

=   (x :: Nil).foldLeft(Nil)(op?)

=   op?(Nil, x)
```

Consequently, `op?(Nil, x) = x :: Nil`.

This suggests to take for `op?` the operator `::` but with its operands swapped.

Deduction of Reverse(3)

We thus arrive at the following implementation of `reverse`.

```
def reverse[T](xs: List[T]): List[T] =
  xs.foldLeft[List[T]](Nil)((xs, x) => x :: xs)
```

Q: What is the complexity of this implementation of `reverse` ?

Deduction of Reverse(3)

We thus arrive at the following implementation of `reverse`.

```
def reverse[T](xs: List[T]): List[T] =
  xs.foldLeft[List[T]](Nil)((xs, x) => x :: xs)
```

Q: What is the complexity of this implementation of `reverse` ?

A: Linear in `xs`

Exercise

Complete the following definitions of the basic functions `map` and `length` on lists, such that their implementation uses `foldRight`:

```
def mapFun[T, U](xs: List[T], f: T => U): List[U] =
  xs.foldRight(Nil)( ??? )

def lengthFun[T](xs: List[T]): Int =
  xs.foldRight(0)( ??? )
```

Exercise

Complete the following definitions of the basic functions `map` and `length` on lists, such that their implementation uses `foldRight`:

```
def mapFun[T, U](xs: List[T], f: T => U): List[U] =
  xs.foldRight(Nil)((y, ys) => f(y) :: ys)

def lengthFun[T](xs: List[T]): Int =
  xs.foldRight(0)((y, n) => n + 1)
```