Week 5 — Lists

More functions on Lists

methods

- xs.length
- xs.last
- xs.init - all elements but last

- xs take n - first n elements of xs
  (or xs if it has less than n els)
- xs drop n - the rest of list after taking first n els.

- xs(n) or xs apply n - the element at index n.

- xs ++ ys - append (all from xs followed by all from ys)

- xs.reverse
- xs updated (n, x) - changes n-th element on x - on a new list (old untouched)

- xs indexOf x
- ~~x~~ xs.contains x - same as indexOf ≥ 0.

Implementation of last

```
def last[T](xs: List[T]): T = xs match {

    case Nil => throw ...
    case List(x) => x
    case y::ys => last(ys)
}
```

$O(n)$!

Concat

```
def concat(xs: List[T], ys: List[T]) = xs match {
    case Nil => ys
    case z :: zs => z :: concat(zs, ys)
}
```

$O(|xs|)$

Reverse

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

Can we do better? than $O(n^2)$

Pairs and Tuples

Merge Sort

- Separate the list into 2 sub-lists, each containing approx a half of the elements
- Sort the sublists
- merge them into one sorted list

```
def  msort(xs: List[Int]): List[Int] = {

     val n = xs.length / 2

     if (n == 0)  xs
     else {
        val (fst, snd) = xs splitAt n
        merge (msort(fst), msort(snd))

     }

}
```

```
def merge (xs: List[Int], ys: List[Int]) = xs match {

   case Nil => ys
   case x :: xs1 =>

        ys match {
           case Nil => xs
           case y :: ys1 =>
               if (x < y) x :: merge(xs1, ys)
               else
                  y :: merge(xs, ys1)

        }
}
```

The splitAt function

returns two sublists - in a pair

Pairs

(x, y) in Scala

```
val pair = ("answer", 42)
    | type (String, Int) )
```

Pairs can be used as patterns:

```
val (label, value) = pair
      ↓        ↘
  "answer"      42
```

Works analogously for tuples

~~Type~~

A tuple type $(T_1 \ldots T_n)$ - is an abbr for

scala.Tuple$n$ $[T_1, \ldots, T_n]$

```
case class Tuple2[T1, T2](_1: +T1, _2: +T2) {
    override def toString ...
}
```

so we can use names _1, _2:

pair._1, pair._2,

but the pattern matching form is generally preferred

Exercise:

p.m. on pair

```
def merge(xs: List[Int], ys: List[Int]):
    List[Int] = (xs, ys) match {

  case (Nil, _) => ys
  case (_, Nil) => xs
  case (x::xs1, y::ys1) =>
        if (x<y)
            x:: merge(xs1, ys)
        else
            y:: merge(xs, ys1)
}
```

}

Implicit Parameters

how to parametrize msort?

idea: to have a comparison function

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

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

parameters can be inferred

```
merge (xs) ((x,y) => x < y )
merge (xs) ((x,y) => x.compareTo(y) < 0)
```

Parametrization with ordering

scala.math. Ordering [T]
provides ways to compare elements of type T.

=> ord: Ordering ...
if (ord. lt(x, y) ..

import math.Ordering

```
msort (nums) (Ordering.Int)
msort (fruits) (Ordering.String)
```

Problem: Passing around lt or ord values is cumbersome.

We can avoid this by making ord as an <u>implicit</u> parameter

```
def msort[T](xs: List)(implicit ord: Ordering) =
    ...
  def merge(xs: ..., ys: ...) = {
      ... if (ord.lt(x, y))...

  ...
   merge(msort(fst), msort(snd))
```

And then we can avoid the ordering param:

```
msort(nums)
msort(fruits)
```

the compiler will figure out the right implicit param pased on the needed type

<u>Rules</u>:

The compiler will search an implicit definition that

- is marked „implicit"
- has a type compatible with T
- is visible

Otherwise – error.

High-order List functions

Recurring patterns:

- transforming each element in a list
- filtering a list
- combining the elements

map: transforming each elem.

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

xs.map( x => x * factor)
xs.map( x => x * x )
```

filter

```
class List[T] {
   ...
    def filter(p: T => Boolean): List[T] =
        this match {

        case Nil => this
        case x::xs =>
            if (p(x))
                x:: xs.filter(p)
            else
                xs.filter(p)
    }
}
```

Variation of filter:

- xs filterNot p –
  same as xs fiter (x => !p(x))

- xs partition p –
  same as
  (xs filter p, xs filterNot p),
  but computed in a single traversal

- xs takeWhile p –
  the longest prefix of xs will elements
  that all satisfy p

- xs dropWhile p –
  the remainder of the list after
  removing leading elements that
  satisfy p

- xs span p –
  same as
  (xs takeWhile p, xs dropWhile p) –
  but in a single traversal.

<u>Exercise:</u>

<u>pack</u> function

"a", "a", "a", "b", "c", "c", "a" =>

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

```
def pack[T](xs: List[T]): List[List[T]] = xs match {
    case Nil => Nil
    case x :: xs1 => {
        (l, r) = xs.span (el => x==el)
        List(l) :: pack(r)
    }
}
```

heck.
?

encode func.

using pack, produce:

aaa bcc a

$\Downarrow$

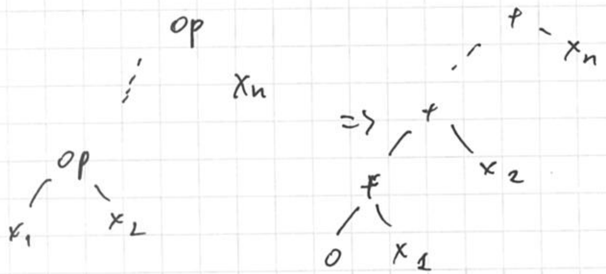
List(("a", 3), ("b", 1), ("c", 2), ("a", 1))

reduce: Reduction of a list

eg. sum – $0 + x1 + \ldots + xn$
product – $1 * x1 * \ldots * xn$

reduceLeft

inserts a given binary operator
between adjacent elements of a list:




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

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

A Shorter way to write functions

instead of ~~((x,y) => (x*y))~~
((x,y) => x * y) we can write

(_ * _)

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

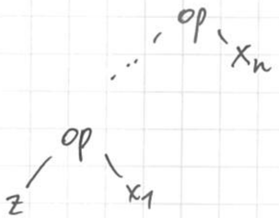
So:

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

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

foldLeft

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

it's like ~~to~~ reduceLeft, but takes an accumulator,
z, as an additional parameter -
which is returned when it's called on
an empty list.



So, sum and product
can be defined as

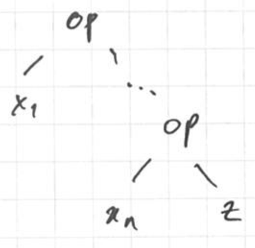
```
def sum(--) =
        (xs foldLeft 0) (_ + _)
```

Implementation

foldLeft and reduceLeft:

```
def reduceLeft (op: (T, T) => T): T = this match {

    case Nil => throw...
    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)

}
```

Fold Right and Reduce Right -
same, but folds / reduces from right to left



implementation

```
def reduceRight(op: (T, T) => T): T = this match {
      case Nil => throw..
      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))
```

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

Some Reasoning on Lists

How we can prove that ++ (concat) is associative ~~and~~ and Nil is a neutral el.?

$$(xs ++ ys) ++ zs = xs ++ (ys ++ zs)$$

$$xs ++ Nil = xs$$

$$Nil + xs = xs$$

We can prove that by structural induction on lists.

The natural induction

the principle of proof by natural induction:

to show a property $P(n)$ for all integers $n \geq b$

- show that we have $P(b)$ (base case)
- for all integers $n \geq b$ show the induction step
  - if one has $P(n)$, it also has $P(n+1)$

Eg.

Factorial

$n! = n \cdot (n-1)!$

Show that for all $n \geq 4$ $n! \geq 2^n$

base $n = 4$, $4! = 24 \geq 16 = 2^4$

induction step $n! \geq 2^n$

$$\cancel{n!}\ (n+1)! \geq (n+1)\, n! > 2 \cdot n! \geq$$
$$\geq 2 \cdot 2^n = 2^{n+1}$$

$n! \geq 2^n$

by our hypothesis

Referential Transparency

A proof can freely apply reduction steps as equalities to some parts of a term

That works because pure functional programs don't have side effects - so a term is equivalent to the term to which it reduces.

Structural Induction

analogous to natural induction

to have a property $P(xs)$ for all lists xs

- show that $P(nil)$ holds (base case)
- for a list xs and some element x, show the induction step

  if $P(xs)$ holds, then $P(x :: xs)$ also holds

Eg
- let's show that for lists xs, ys, zs:

$$(xs ++ ys) ++ zs = xs ++ (ys ++ zs)$$

```
def concat[T](xs: List[T], ys: List[T]) = xs match {
    case Nil => Nil
    case x :: xs1 => x :: concat(xs1, ys)
}
```

$Nil ++ ys = ys$ // first clause
$(x :: xs1) ++ ys = x :: (xs1 ++ ys)$ // second clause

base case

Nil

left: $(Nil ++ ys) ++ zs = ys ++ zs$ (1st clause)
right: $Nil ++ (ys ++ zs) = ys ++ zs$ (1st clause)

case established.