The List Type

Like arrays, lists are *homogeneous*. That is, the elements of a list must all have the same type.

The type of a list with elements of type T is written $List[T]$. (compare to $[\,]T$ for the type of arrays of type T in C or Java).

E.g.

$$
\begin{aligned}
&\textbf{val}\ fruit : List[String] &&= [\text{"apples"}, \text{"oranges"}, \text{"pears"}] \\
&\textbf{val}\ nums : List[Int] &&= [1, 2, 3, 4] \\
&\textbf{val}\ diag3 : List[List[Int]] &&= [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
\end{aligned}
$$

Lists

The list is a fundamental data structure in functional programming.

A list with elements $x_1, \ldots, x_n$ is written

$$[x_1, \ldots, x_n]$$

Examples:

$$
\begin{aligned}
&\textbf{val}\ fruit &&= [\text{"apples"}, \text{"oranges"}, \text{"pears"}] \\
&\textbf{val}\ nums &&= [1, 2, 3, 4] \\
&\textbf{val}\ diag3 &&= [[1, 0, 0], [0, 1, 0], [0, 0, 1]] \\
&\textbf{val}\ empty &&= [\,]
\end{aligned}
$$

Note the similarity to an array initializer in C or Java. However there are two important differences between lists and arrays:

1. Lists are immutable – elements of a list cannot be changed.
2. Lists are recursive where arrays are flat.

Operations on lists

All operations on lists can be expressed in terms of the following three:

head	returns the first element of a list
tail	returns the list consisting of all elements except the first
isEmpty	returns *true* iff the list is empty

These operations are defined as methods of list objects. E.g.

$$\textit{fruit.head} = \textit{"apples"}$$
$$\textit{fruit.tail.head} = \textit{"oranges"}$$
$$\textit{diag3.head} = [1,\ 0,\ 0]$$
$$\textit{empty.head} \quad \rightarrow \quad (\textit{Exception "head of empty list"})$$

List Constructors

All lists are built from:

- The empty list *Nil*.
- The construction operation $x :: xs$ which returns a new list with first element x, followed by the elements of xs.

I.e:

$$\begin{aligned} \textit{fruit} &= \textit{"apples"} :: (\textit{"oranges"} :: (\textit{"pears"} :: \textit{Nil})) \\ \textit{nums} &= 1 :: (2 :: (3 :: (4 :: \textit{Nil}))) \\ \textit{empty} &= \textit{Nil} \end{aligned}$$

Convention: The '::' operation associates to the right: $A :: B :: C$ is interpreted as $A :: (B :: C)$.

Therefore, we can drop the parentheses in the definition above. E.g.

$$\textit{nums} \quad = 1 :: 2 :: 3 :: 4 :: \textit{Nil}$$

Functions over lists

Using the two constructors *Nil* and :: and the three methods *head*, *tail*, and *isEmpty*, we can now formulate other common functions over lists.

The length function

length (*xs*) should return the number of elements of *xs*. It is defined as follows.

```
def length(xs: List[String]): Int =
  if (xs.isEmpty) 0
  else 1 + length(xs.tail)

> length(nums)
4
```

Problem: *length* can be applied to lists of strings only.

How can we formulate the function so that it can be applied to all lists?

Example

Suppose we want to sort a list of numbers into ascending order:

- One way to sort the list [7, 3, 9, 2] is to sort the tail [3, 9, 2] to give [2, 3, 9].
- It is then a matter of inserting the head 7 in the right place to give the result [2, 3, 7, 9].

This idea describes *Insertion Sort*:

```
def isort(xs: List[Int]): List[Int] =
  if (xs.isEmpty) Nil
  else ins(xs.head, isort(xs.tail))
```

What is an implementation of the missing function *ins*?

What is the complexity of insertion sort?

In our example, one can equally write:

```
length(nums)      /* [Int] inferred since nums: List[Int] */
```

However, one cannot omit formal type parameters:

```
> def length(x: a) = ...
colsole:1: not found: a
```

Functions that take type parameters are called *polymorphic*.

The word means “many-formed” in Greek: i.e. the function can be applied to arguments of many different types.

Polymorphism

Idea: Pass the type of elements as an additional *type*-parameter to *length*.

```
def length[a](xs: List[a]): Int =
  if (xs.isEmpty) 0
  else 1 + length(xs.tail)

> length[Int](nums)
4
```

Syntax:

- Formal and actual type parameters are enclosed in brackets, e.g. [a], [Int].
- Actual type parameters can be dropped if they can be inferred from the function's value parameters and expected result type (which is usually the case).

Functions *last* and *init*

The *head* method returns the first element of a list. A function to return the last element of a list can be written as follows.

```
def last[a](xs: List[a]) = {
  if (xs.isEmpty) error("last of empty list")
  else if (xs.tail.isEmpty) xs.head
  else last(xs.tail)
}
```

Exercise: Write a function *init* which returns all elements of a list except the last one (i.e. *init* is the complement to *tail*).

```
def init[a](xs: List[a]) = {
  if (xs.isEmpty) error("init of empty list")
  else if (xs.tail.isEmpty) ?
  else ?
}
```


Concatenating Lists

The :: operator is asymmetric – it is applied to a list element and a list.

There is also symmetric operator ::: which *concatenates* two lists.

```
> [1, 2] ::: [3, 4]
[1, 2, 3, 4]
```

::: can be defined in terms of the primitive operations. Let's write an equivalent function

```
def concat[a](xs: List[a], ys: List[a]): List[a] =
  if (xs.isEmpty)
    ?
  else
    ?
```

Q: What is the complexity of *concat*?

Function *reverse*

Here's a function to reverse the elements of a list.

```
def reverse[a](xs: List[a]) = {
  if (xs.isEmpty) []
  else reverse(xs.tail) ::: [xs.head]
}
```

Q: What is the complexity of *reverse*?

A: $n + (n-1) + ... + 1 = n(n+1)/2$ where n is the length of *xs*.

Can you do better? (to be solved later).

Aside: Exceptions

There is a predefined function *error*, which aborts the program with a given error message.

This is defined as follows.

```
def error[a](msg: String): a =
  java.lang.RuntimeException(msg).throw
```

Note that *error* is polymorphic – it is declared to return an argument of an arbitrary type.

Of course, in fact *error* does not return at all.

So the polymorphism of *error* is used here to make it usable in all contexts.

Classes *Nil* and *Cons*

These classes are defined as follows.

```
class Nil[a] extends List[a] with {
  def head: a = error("head of empty list")
  def tail: List[a] = error("tail of empty list")
  def isEmpty: Boolean  = True
}

class Cons[a](x: a, xs: List[a]) extends List[a] with {
  def head: a = x
  def tail: List[a] = xs
  def isEmpty: Boolean = False
}
```


The List Class

List is not a primitive type in Scala; it can be defined by a class. Here's a possible implementation.

```
abstract class List[a] with {
  abstract def head: a
  abstract def tail: List[a]
  abstract def isEmpty: Boolean
}
```

Note that *List* is a parametric class.

All methods in class *List* are abstract. Implementations of these methods are found in two concrete subclasses:

- *Nil* for empty lists
- *Cons* for nonempty lists.

The Cons and Concat operators

Operators starting with ':' are treated specially in Scala.

- All such operators are right-assocative. E.g.

$$x + y + z = (x + y) + z \qquad but \qquad x :: y :: z = x :: (y :: z)$$

- All such operators are treated as methods of their right operand. E.g.

$$x + y = x.+(y) \qquad but \qquad x :: y = y.::(x)$$

  (Note however, that operand expressions are still evaluated from left to right. So, if d and e are expressions, then the expansion is:

$$d :: e = (\textbf{val}\ x = d;\ e.::(x))$$

Further List Methods

The functions covered so far all exist as methods in class *List*. For instance:

```
abstract class List[a] with {
  abstract def head: a
  abstract def tail: List[a]
  abstract def isEmpty: Boolean

  def length = if (isEmpty()) 0 else 1 + tail.length

  def init: List[a] =
    if (isEmpty) error("init of empty list")
    else if (tail.isEmpty) Nil
    else head :: tail.init
  ...
}
```

More List Methods

Method *take*(n) returns the n leading elements of its list (or the list itself if it is shorter than n.

Method *drop*(n) returns its list without the n leading elements.

Method *at*(n) returns the n'th element of a list.

These are defined as follows:

```
abstract class List[a] with {
  ...
  def take(n: Int): List[Int] =
    if (n == 0) [] else head :: tail.take(n - 1)

  def drop(n: Int): List[Int] =
    if (n == 0) this else tail.drop(n - 1)

  def at(n: Int) = drop(n).head
}
```


The definition of :: and ::: is now straightforward:

```
abstract class List[a] with {
  ...
  def ::(x: a) = Cons(x, this)

  def :::(prefix: List[a]): List[a] =
    if (prefix.isEmpty) this
    else prefix.head :: (prefix.tail ::: this)

    /* or, equivalently:
          this.:::(prefix.tail).::(prefix.head)
    */
```

The most flexible design is to make the *sort* function polymorphic, and to pass the desired comparison operation as additional parameter. E.g.:

```
def msort[a](less: (a, a)Boolean)(xs: List[a]): List[a] = {
  val n = xs.length/2
  if (n == 0) xs
  else {
    def merge(xs1: List[a], xs2: List[a]): List[a] = ...
    val xs1 = xs.take(n)
    val xs2 = xs.drop(n)
    merge(msort(less)(xs1), msort(less)(xs2))
  }
}
```

Exercise: Define the *merge* function. Here are two test cases.

```
merge([1, 3], [2, 4]) = [1, 2, 3, 4]
merge([1, 2], []) = [1, 2]
```


Sorting Lists Faster

As a non-trivial example, let's design a function to sort the elements of a list which 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,
  1. Split the list in two sublists each containing about half the elements of the original list.
  2. Sort the two sublists.
  3. Merge the two sorted sublists into one sorted list.

To implement this, we still have to specify

- What are the type(s) of the elements to be sorted?
- What is the method to compare two elements?

Patterns of Computation

- The examples show that functions over lists often have similar structures
- We can identify several patterns of computation like
  - Transform every element of a list in some way.
  - Extract from a list all elements saisfying a criterion.
  - Combine the elements of a list using some operator.
- Functional programming languages enable programmers to write general functions which implement patterns like this
- These functions are *higher-order functions* which get a transformation or an operator as one argument

Here is an example how *msort* is used.

```
> def less(x: Int, y: Int) = x < y
> msort(less)([5, 7, 1, 3])
[1, 3, 5, 7]
```

The definition of *msort* is curried, to make it easy to specialize it with particular comparison functions.

```
> val intSort = msort(less)
> val reverseSort = msort(x: Int, y: Int => x > y)
> intSort([6, 3, 5, 5])
[3, 5, 5, 6]
> reverseSort([6, 3, 5, 5])
[6, 5, 5, 3]
```

Complexity:

The complexity of *msort* is $O(n \log n)$.

This complexity does not depend on the initial distribution of elements in the list.

Using *map*, *scaleList* can be more consisely written as follows.

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

Exercise: Consider a function which squares all elements of a list and returns a list with the results. Complete the following two equivalent definitions of *squareList*.

```
def squareList(xs: List[Int]): List[Int] =
  if (xs.isEmpty) ??
  else ??
```

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


Mapping over lists

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

For instance, to scale each element of a list by a given factor.

```
def scaleList(xs: List[Double], factor: Double): List[Double] =
  if (xs.isEmpty) []
  else xs.head * factor :: scaleList(xs.tail, factor)
```

This pattern can be generalized to the *map* method of *List*:

```
abstract class List[a] with {
  ...
  def map[b](f: (a)b): List[b] =
    if (isEmpty) []
    else f(head) :: tail.map(f)
}
```

Using *filter*, *posElems* can be more consisely written as follows.

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


Filtering

Another common operation selects from a list all elements fulfilling a given criterion. For instance:

```
def posElems(xs: List[Int]): List[Int] =
  if (xs.isEmpty) []
  else if (xs.head > 0) xs.head :: posElems(xs.tail)
  else posElems(xs.tail)
```

This pattern is generalized to the method *filter* in *List*:

```
abstract class List[a] with {
  ...
  def filter(p: (a)Boolean): List[a] =
    if (isEmpty) []
    else if (p(head)) head :: tail.filter(p)
    else tail.filter(p)
```

The generalization *reduce* inserts a given binary operator between adjacent elements.

E.g.

$$[x_1, ..., x_n].reduce(op) \quad = \quad (...(x_1 \; op \; x_2) \; op \; ... \;) \; op \; x_n$$

Then we can simply write:

$$\begin{array}{lcl} \textbf{def}\; sum(xs\colon List[Int]) & = & (0 :: xs).reduce(x, y \Rightarrow x + y) \\ \textbf{def}\; product(xs\colon List[Int]) & = & (1 :: xs).reduce(x, y \Rightarrow x * y) \end{array}$$

Reduce and Fold

Another common operation is to combine the elements of a list with some operator.

For instance:

$$\begin{array}{lcl} sum([x_1, ..., x_n]) & = & 0 + x_1 + ... + x_n \\ product([x_1, ..., x_n]) & = & 1 * x_1 * ... * x_n \end{array}$$

These can be implemented with the usual recursive scheme:

```
def sum(xs: List[Int]): Int =
  if (xs.isEmpty) 0 else xs.head + sum(xs.tail)

def product(xs: List[Int]): Int =
  if (xs.isEmpty) 1 else xs.head * product(xs.tail)
```

fold takes as additional parameter an *accumulator* *acc*, which is returned for empty lists.

That is,

$$[x_1, ..., x_n].fold(op)(acc) \quad = \quad (...(acc \; op \; x_1) \; op \; ... \;) \; op \; x_n$$

So *sum* and *product* could be defined alternatively as follows.

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


Implementation of Reduce

How can *reduce* be implemented?

```
abstract class List[a] with {
  ...
  def reduce(op: (a, a): a): a = {
    if (isEmpty) error("reduce of empty list")
    else tail.fold(op)(head)
  }
  def fold[b](op: (b, a): b)(acc: b): b = {
      if (isEmpty) acc
      else tail.fold(op)(op(acc, head))
  }
}
```

The *reduce* function is defined in terms of another generally useful function, *fold*.

FoldRight and ReduceRight

Applications of *fold* and *reduce* expand to left-leaning trees:

They have duals foldRight and reduceRight, which produce right-leaning trees. I.e.

$$
\begin{aligned}
[x_1, ..., x_n].\mathit{reduceRight}(\mathit{op}) &= x_1 \ \mathit{op} \ (\ ... \ (x_{n-1} \ \mathit{op} \ x_n)...) \\
[x_1, ..., x_n].\mathit{foldRight}(\mathit{op})(\mathit{acc}) &= x_1 \ \mathit{op} \ (\ ... \ (x_n \ \mathit{op} \ \mathit{acc})...)
\end{aligned}
$$

These are defined as follows.


```
def reduceRight(op: (a, a)a): a =
  if (isEmpty) error("reduce of empty list")
  else if (tail.isEmpty) head
  else op(head, tail.reduceRight(op))

def foldRight[b](op: (a, b)b)(z: b): b =
  if (isEmpty) z
  else op(head, tail.foldRight(op)(z))
```

For associative operators *op*, *fold* and *foldRight* are equivalent. (even there may be a difference in efficiency.)

But sometimes, only one of the two operators is appropriate or has the right type:

Example: Here is an alternative formulation of *concat*:

```
def concat[a](xs: List[a], ys: List[a]): List[a] = {
  def cons(x: a, xs: List[a]): List[a] = x :: xs
  xs.foldRight(cons)(ys)
}
```

Here it is not possible to replace the *foldRight* with *fold*. (Why not?)

Second,

$$\begin{aligned} & x :: [\,] \\ = \ & \mathit{reverse}\,(x :: [\,]) && \textit{// by specification} \\ = \ & (x :: [\,]).\mathit{fold}\,(\mathit{op})\,(z) && \textit{// by the template for reverse} \\ = \ & \mathit{op}\,([\,], x) && \textit{// by definition of fold} \end{aligned}$$

Hence, $\mathit{op}\,([\,], x) = x :: [\,]$. This suggests to take as *op* the :: operator with its operands exchanged.

Hence, we arrive at the following implementation fo *reverse*.

```
def reverse[a](xs: List[a]): List[a] = {
  def snoc(xs: List[a], x: a): List[a] = x :: xs
  xs.fold(snoc)([])
}
```

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

List Reversal Again

Here is list reversal function with linear cost.

The idea is to use a *fold* operation:

```
def reverse[a](xs: List[a]): List[a] = xs.fold(op?)(acc?)
```

We only need to fill in the *op?* and *acc?* parts.

Let's try to deduce them from examples.

First,

$$\begin{aligned} & [\,] \\ = \ & \mathit{reverse}\,([\,]) && \textit{// by specification} \\ = \ & [\,].\mathit{fold}\,(\mathit{op})\,(z) && \textit{// by the template for reverse} \\ = \ & z && \textit{// by definition of fold} \end{aligned}$$

Hence, $\mathit{acc} = [\,]$.

Appendix: Class List

Here is an implementation of class *List* in Scala (as far as we have discussed it in the course).

```
import Boolean

def error[a](x: String):a = (new java.lang.RuntimeException(x)).throw

abstract class List[a] with {
  abstract def isEmpty: Boolean
  abstract def head: a
  abstract def tail: List[a]

  def ::(x: a) =
    Cons(x)(this)

  def :::(prefix: List[a]): List[a] =
    if (prefix.isEmpty) this
    else prefix.head :: (prefix.tail ::: this)
```


Summary

- We have encountered the list as a fundamental data structure in functional programming.
- Lists are defined by parameterized classes, and operated upon by polymorphic methods.
- Lists are the analogue of arrays in imperative languages.
- But unlike arrays, lists elements are usually not accessed by their index.
- Instead, lists are traversed recurisvely or via higher-order combinators. such as *map*, *filter*, *fold*, *foldRight*.

```
def map[b](f: (a)b): List[b] =
  if (isEmpty) Nil
  else f(head) :: tail.map(f)

def filter(p: (a)Boolean): List[a] =
  if (isEmpty) this
  else if (p(head)) head :: tail.filter(p)
  else tail.filter(p)

def reduce(op: (a, a)a): a =
  if (isEmpty) error("reduce of empty list")
  else tail.fold(op)(head)

def reduceRight(op: (a, a)a): a =
  if (isEmpty) error("reduce of empty list")
  else if (tail.isEmpty) head
  else op(head, tail.reduceRight(op))

def fold[b](op: (b, a)b)(z: b): b =
  if (isEmpty) z
  else tail.fold(op)(op(z, head))
```



```
def length: Int =
  if (isEmpty) 0
  else 1 + tail.length

def init: List[a] =
  if (isEmpty) error("Nil.init")
  else if (tail.isEmpty) Nil
  else head :: tail.init

def last: a =
  if (isEmpty) error("Nil.last")
  else if (tail.isEmpty) head
  else tail.last

def take(n: Int): List[a] =
  if (n == 0) Nil
  else head :: tail.take(n-1)

def drop(n: Int): List[a] =
  if (n == 0) this
  else tail.drop(n-1)

def at(n: Int) = drop(n).head
```

```
final class Nil[a] extends List[a] with {
    def isEmpty = True
    def head: a = error("head of empty list")
    def tail: List[a] = error("tail of empty list")
}

final class Cons[a](x: a)(xs: List[a]) extends List[a] with {
    def isEmpty = False
    def head = x
    def tail = xs
}

def cons[a](x: a, xs: List[a]): List[a] = Cons(x)(xs)
def nil[a]: List[a] = Nil[a]
```



```
    def foldRight[b](op: (a, b)b)(z: b): b =
        if (isEmpty) z
        else op(head, tail.foldRight(op)(z))

    def reverse: List[a] = {
        if (isEmpty) Nil
        else tail ::: head :: Nil
    }

    override def toString(): String = "[" + mkString(",") + "]"
    def mkString(sep: String): String = {
        if (isEmpty) ""
        else if (tail.isEmpty) head.toString()
        else head.toString().concat(sep).concat(tail.mkString(sep))
    }
}
```