Week 4: Pattern Matching

Say, we want to write a little interpreter for arithmetic expressions.

For simplicity, we restrict ourselves to numbers and + operations.

Expressions can be represented as a class hierarchy, with a base class *Expr* and two subclasses *Number* and *Sum*.

To process an expression, we need to know what form it is, and what its components are.

This leads to the following implementation.


```
abstract class Expr {
    def isNumber: boolean;
    def isSum: boolean;
    def numValue: int;
    def leftOp: Expr;
    def rightOp: Expr;
}
class Number(n: int) extends Expr {
    def isNumber: boolean = true;
    def isSum: boolean = false;
    def numValue: int = n;
    def leftOp: Expr = error("Number.leftOp");
    def rightOp: Expr = error("Number.rightOp");
}
class Sum(e1: Expr, e2: Expr) extends Expr {
    def isNumber: boolean = false;
    def isSum: boolean = true;
    def numValue: int = error("Sum.numValue");
    def leftOp: Expr = e1;
    def rightOp: Expr = e2;}
```

Now, we can write an evaluation function as follows.

```
def eval(e: Expr): int = {
    if (e.isNumber) e.numValue
    else if (e.isSum) eval(e.leftOp) + eval(e.rightOp)
}
```

Problem: Writing all these classification and access functions is rather boring!

Also, what happens if we want to add new forms of expression, say

```
class Prod(e1: Expr, e2: Expr) extends Expr;     // e1 * e2
class Var(x: String) extends Expr;               // Variable 'x'
```

Then we need to add new classification and access methods to all previously defined classes.

How can we solve this problem?

Solution 1: Object-Oriented Decomposition

For instance, assume all we want is to evaluate expressions.

Then we could define:

```
abstract class Expr {
    def eval: int;
}
class Number(n: int) extends Expr {
    def eval: int = n;
}
class Sum(e1: Expr, e2: Expr) extends Expr {
    def eval: int = e1.eval + e2.eval;
}
```

But what happens if we want to pretty-print expressions instead?

- need to define new method in all subclasses.

What if we want to simplify them? Say by applying the rule

$$a * b + a * c \quad \to \quad a * (b + c)$$

Problem: This is a non-local simplification. It cannot be encapsulated in a method of a single object.

So, we are back to square 1: Need access methods for all different subclasses.

Solution 2: Functional Decomposition via Pattern Matching

Observation: The only purpose of access functions is to reverse the construction process

- what sub-class was used?
- what were the constructor arguments?

This situation is so common that we automate it in Scala.

Case Classes (Algebraic Data Types)

A case class definition is like a normal class definition, prefixed with the modifier ***case***. E.g.

```
abstract class Expr;
case class Number(n: int) extends Expr;
case class Sum(e1: Expr, e2: Expr) extends Expr;
```

As before, this defines an abstract superclass *Expr* with two concrete subclasses *Number* and *Sum*.

Furthermore, it also implicitly defines *factory methods*

```
def Number(n: int) = new Number(n);
def Sum(e1: int, e2: int) = new Sum(e1, e2)
```

so one can write shorter *Number(1)* instead of ***new*** *Number(1)*.

However, all three classes are now empty, so how can we access the members?

Pattern Matching

Pattern matching is a generalization of C/Java's *switch* statement to class hierarchies.

It is expressed by a standard method *match*, which is defined in the root class *Any* (so it is available everywhere).

Example:

```
def eval(e: Expr): int = e match {
    case Number(n) => n
    case Sum(e1, e2) => eval(e1) + eval(e2)
}
```

Rules:

- *match* is followed by a sequence of *cases*.
- Every case associates a *pattern* with an *expression*.
- A *MatchError* exception is thrown if no pattern matches the selector value.

Forms of Patterns

- Patterns are built from:
  - Constructors, e.g. *Number*, *Sum*,
  - variables, e.g. *n*, *e1*, *e2*,
  - the “wildcard” pattern _,
  - constants, e.g. *1*, ***true***.
- Variables always start with a lower-case letter.
- Each variable name may occur only once in a pattern. E.g. *Sum*(*x*, *x*) is illegal as a pattern.
- Constructors and constant names start with an upper-case letter, except for the reserved words ***null***, ***true***, ***false***.

Meaning of Pattern Matching

A pattern matching expression

e.match { ***case*** $p_1 \Rightarrow e_1$ ... ***case*** $p_n \Rightarrow e_n$ }

matches the patterns $p_1, \ldots, p_n$ in the order they are written against the selector value *e*.

- A constructor pattern $C(p_1, \ldots, p_n)$ matches all values that are of type C (or a subtype thereof) and that have been constructed with C-arguments matching patterns $p_1, \ldots, p_n$.
- A variable pattern x matches any value and *binds* the variable name to that value.
- A constant pattern C matches values that are equal (in terms of ==) to C.

The pattern matching expression rewrites to the right-hand-side of the first case whose pattern matches the selector value.

References to pattern variables are replaced by corresponding constructor arguments.

Example: :

```
     eval(Sum(Number(1), Number(2)))
→    Sum(Number(1), Number(2)) match {
       case Number(n) ⇒ n
       case Sum(e1, e2) ⇒ eval(e1) + eval(e2)
     }
→    eval(Number(1)) + eval(Number(2))
→    Number(1) match {
       case Number(n) ⇒ n
       case Sum(e1, e2) ⇒ eval(e1) + eval(e2)
     } + eval(Number(2))
→    1 + eval(Number(2))
→*   1 + 2 → 3
```


Mixing Pattern Matching with Methods

Of course, it is also possible to define a pattern matching function as a method of the superclass.

Example:

```
abstract class Expr {
  def eval: int = this match {
    case Number(n) ⇒ n
    case Sum(e1, e2) ⇒ e1.eval + e2.eval
  }
}
```

Exercise

Consider the following three classes representing trees of integers. These classes can be seen as an alternative representation of *IntSet*:

```
abstract class IntTree;
case class Empty extends IntTree;
case class Node(elem: int, left: IntTree, right: IntTree) extends IntTree;
```

Complete the following implementation of function *contains* for *IntTree*'s.

```
def contains(t: IntTree, v: int): boolean = t match {
  ...
}
```


Lists

The list is a fundamental data structure in functional programming.

A list with elements $x_1, ..., x_n$ is written

```
List(x1, ..., xn)
```

Examples:

```
val fruit  = List("apples", "oranges", "pears");
val nums   = List(1, 2, 3, 4);
val diag3  = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1));
val empty = List();
```

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.

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.

$$\begin{array}{ll}
\textbf{val}\ fruit\ : List\,[String] & = List\,(\text{"apples"},\ \text{"oranges"},\ \text{"pears"});\\
\textbf{val}\ nums : List\,[int] & = List\,(1,\ 2,\ 3,\ 4);\\
\textbf{val}\ diag3 : List\,[List\,[int]] & = List\,(List\,(1,\ 0,\ 0),\ List\,(0,\ 1,\ 0),\ List\,(0,\ 0,\ 1));
\end{array}$$

List Constructors

All lists are built from:

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

I.e:

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

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.

$$nums \quad = \ 1 :: 2 :: 3 :: 4 :: Nil$$

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.

fruit.head = *"apples"*
fruit.tail.head = *"oranges"*

diag3.head = *List* (*1, 0, 0*)

empty.head → (*Exception "head of empty list"*)

Example

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

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

This idea describes *insertion sort*:

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

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

What is the complexity of insertion sort?

List Patterns

Both :: and *Nil* exist as case classes, so it is also possible to decompose lists via pattern matching.

As syntactic sugar, the *List* (...) constructor can also be used as a pattern, with the translation $List(p_1, ..., p_n) = p_1 :: ... :: p_n :: Nil$.

Hence, *isort* can be written alternatively as follows.

```
def isort(xs: List[int]): List[int] = xs match {
  case List() => List()
  case x :: xs1 => insert(x, isort(xs1))
}
```

where

```
def insert(x: int, xs: List[int]): List[int] = xs match {
  case List() => List(x)
  case y :: ys => if (x ≤ y) x :: xs else y :: insert(x, ys)
}
```


Other Functions over lists

Using list constructors and patterns 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 = xs match {
  case List() => 0
  case x :: xs1 => 1 + length(xs1)
}
> 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?

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).

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) = ...
console: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.

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.

```
> List(1, 2) ::: List(3, 4)
List(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] = xs match {
    case List() =>
        ?
    case x :: xs1 =>
        ?
}
```

Q: What is the complexity of *concat*?

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]): List[a] = xs match {
    case List() => error("last of empty list")
    case List(x) => x
    case x :: xs1 => 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*).

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.

Function *reverse*

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

```
def reverse[a](xs: List[a]): List[a] = xs match {
  case List() => List()
  case x :: xs => reverse(xs) ::: List(x)
}
```

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).

The List Class

List is not a primitive type in Scala. It is instead defined by an abstract base class and two case classes for :: and *Nil*. Here's a partial implementation.

```
abstract class List[a] {
  def head: a;
  def tail: List[a];
  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
- :: for nonempty lists.

Classes *Nil* and ::

These classes are defined as follows.

```
case class Nil[a] extends List[a] {
  def isEmpty = true;
  def head: a = error("Nil.head");
  def tail: List[a] = error("Nil.tail");
}

case class ::[a](x: a, xs: List[a]) extends List[a] {
  def isEmpty = false;
  def head: a = x;
  def tail: List[a] = xs;
}
```

Further List Methods

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

```
abstract class List[a] {
  def head: a;
  def tail: List[a];
  def isEmpty: boolean;

  def length = match {
    case Nil => 0
    case x :: xs => 1 + xs.length
  }

  def init: List[a] = match {
    case Nil => error("Nil.init")
    case x :: Nil => List()
    case x :: xs => x :: init(xs)
  }
  ...
}
```


The Cons and Concat operators

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

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

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

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

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

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

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

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

```
abstract class List[a] {
  ...
  def ::(x: a): List[a] = new scala.::(x, this);

  def :::(prefix: List[a]): List[a] = prefix match {
    case Nil => this
    case p :: ps => p :: ps ::: this   /* or, equivalently: this.:::(ps).::(p) */
  }
```


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] {
  ...
  def take(n: int): List[int] =
    if (n == 0) List() 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;
}
```

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?

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] = ...
    merge(msort(less)(xs take n), msort(less)(xs drop n))
  }
}
```

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

```
merge(List(1, 3), List(2, 4)) = List(1, 2, 3, 4)
merge(List(1, 2), List()) = List(1, 2)
```

Here is an example how *msort* is used.

```
> def iless(x: int, y: int) = x < y
> msort(iless)(List(5, 7, 1, 3))
List(1, 3, 5, 7)
```

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

```
> val intSort = msort(iless)
> val reverseSort = msort((x: int, y: int) => x > y)
> intSort(List(6, 3, 5, 5))
List(3, 5, 5, 6)
> reverseSort(List(6, 3, 5, 5))
List(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.

Pairs

Pair is another common class in Scala. It can be defined as follows.

```
case class Pair[a, b](_1: a, _2: b);
```

As a usage example, here is a function which returns the quotient and rest of two given integer numbers...

```
def divmod(x: int, y: int) = Pair(x / y, x % y);
```

And here is how the function can be used:

```
divmod(x, y) match {
  case Pair(n, d) => System.out.println("quotient: " + n + ", rest: " + d);
}
```

It is also possible to use the constructor parameter names for accessing elements of a case class directly. E.g.

```
val p = divmod(x, y);
System.out.println("quotient: " + p._1);
```

The idea of pairs is generalized in scala to tuples of greater arity. There is a case class *Tuple_n* for every *n* from *2* to *9*.

So *Pair* is in effect an alias for *Tuple2*.

Patterns of Computation over Lists

- 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 satisfying 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

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] = xs match {
  case Nil ⇒ xs
  case x :: xs1 ⇒ x * factor :: scaleList(xs1, factor)
}
```

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

```
abstract class List[a] { ...
  def map[b](f: a ⇒ b): List[b] = this match {
    case Nil ⇒ this
    case x :: xs ⇒ f(x) :: xs.map(f)
  } ... }
```


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] = xs match {
  case List() ⇒ ??
  case y :: ys ⇒ ??
}
def squareList(xs: List[int]): List[int] =
  xs map ??
```

Filtering

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

```
def posElems(xs: List[int]): List[int] = xs match {
  case Nil => xs
  case x :: xs1 => if (x > 0) x :: posElems(xs1) else posElems(xs1)
}
```

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

```
abstract class List[a] {
  ...
  def filter(p: a => boolean): List[a] = this match {
    case Nil => this
    case x :: xs => if (p(x)) x :: xs.filter(p) else xs.filter(p)
  }
```


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

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

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] {
  def isEmpty: boolean
  def head: a
  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)
```



```
  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 || isEmpty) Nil
    else head :: tail.take(n-1)

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

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

```
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 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))
}
}
```

```
final class Nil[a] extends List[a] {
    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] {
    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]
```