Week 4 : Pattern Matching (Filtrage de motifs)

Suppose we want to write a small interpreter for arithmetic expressions.

To keep it simple, we will restrict ourselves to numbers and additions.

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

To treat an expression, it's necessary to know the expression's shape and its components.

This brings us 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
}
```

We can now 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)
  else error("Unknown expression " + e)
}
```

Problem: Writing all these classification and accessor functions quickly becomes tedious!

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

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

We should add methods for classification and access to all classes defined above.

How can we fix this problem?

Solution 1: Object-Oriented Decomposition

For example, suppose that we want to only evaluate expressions.

We could then 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'd like to display expressions now? We have to define new methods in all the subclasses.

And if you want to simplify the expressions, e.g. by means of the rule:

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

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

We are back to square one; we need access methods for different subclasses.

Solution 2: Functional Decomposition via Matching

Finding: the sole purpose of test and accessor functions is to reverse the construction process:

- Which subclass was used?
- What were the arguments of the constructor?

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

Case Classes (Type Algebras)

A *case class* is similar to a normal class definition, except that it is preceded by the modifier ***case***. For example:

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

Like before, this defines an abstract base class *Expr*, and two concrete subclasses *Number* and *Sum*.

It also implicitly defines construction functions, *factory functions*.

```
def Number(n: Int) = new Number(n)
def Sum(e1: Expr, e2: Expr) = new Sum(e1, e2)
```

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

However, these classes are now empty. So how can we access the members?

Pattern Matching

Pattern matching is a generalization of *switch* from C/Java to class hierarchies.

It's expressed in Scala using the keyword ***match***.

Exemple :

```
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*.
- Each case associates an *expression* to a *pattern*.
- An exception *MatchError* is thrown if no pattern matches the value of the selector.

Pattern forms

- Patterns are constructed from:
  - constructors, e.g. *Number*, *Sum*,
  - variables, e.g. *n*, *e1*, *e2*,
  - "*wildcard*" patterns _,
  - constants, e.g. *1*, ***true***.
- Variables always begin with a lowercase letter.
- The same variable name can only appear once in a pattern. So, *Sum*(*x*, *x*) is not a legal pattern.
- Constructors and the names of constants begin with a capital letter, with the exception of the reserved words ***null***, ***true***, ***false***.

Significance of Pattern Matching

An expression of the type

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

matches the value of the selector e with the patterns $p_1, \ldots, p_n$ in the order in which they are written.

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

The matching expression is rewritten to the right-hand side of the first case where the pattern matches the selector.

References to the pattern variables are replaced by the corresponding constructor arguments.

Exemple :

$$
\begin{array}{ll}
 & \mathit{eval}(\mathit{Sum}(\mathit{Number}(1),\ \mathit{Number}(2))) \\[1em]
\rightarrow & \mathit{Sum}(\mathit{Number}(1),\ \mathit{Number}(2))\ \textbf{\textit{match}}\ \{ \\
 & \quad \textbf{case}\ \mathit{Number}(n) \Rightarrow n \\
 & \quad \textbf{case}\ \mathit{Sum}(e1,\ e2) \Rightarrow \mathit{eval}(e1) + \mathit{eval}(e2) \\
 & \} \\[1em]
\rightarrow & \mathit{eval}(\mathit{Number}(1)) + \mathit{eval}(\mathit{Number}(2)) \\[1em]
\rightarrow & \mathit{Number}(1)\ \textbf{\textit{match}}\ \{ \\
 & \quad \textbf{case}\ \mathit{Number}(n) \Rightarrow n \\
 & \quad \textbf{case}\ \mathit{Sum}(e1,\ e2) \Rightarrow \mathit{eval}(e1) + \mathit{eval}(e2) \\
 & \} + \mathit{eval}(\mathit{Number}(2)) \\[1em]
\rightarrow & 1 + \mathit{eval}(\mathit{Number}(2)) \\
\rightarrow^* & 1 + 2 \rightarrow 3
\end{array}
$$

Pattern Matching and Methods

Of course, it's also possible to define the evaluation function as a method of the superclass.

Exemple :

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


Exercise

We 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 the function *contains* for the *IntTrees*.

```
def contains(t: IntTree, v: Int): Boolean = t match {
  ...
}
```

Lists

The list is a fundamental data structure in functional programming.

A list having x_1, ..., x_n as elements is written *List*(x_1, ..., x_n).

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 with the initialization of an array in C or Java.
However, there are two important differences between lists and arrays.

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

Type List

Like arrays, lists are *homogeneous*: 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] (compared to []T for the type of arrays of elements of type T in C or Java.)

For example:

```
val fruit : List[String]     = List("apples", "oranges", "pears")
val nums  : List[Int]        = List(1, 2, 3, 4)
val diag3 : List[List[Int]]  = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
```

Constructors of Lists

All lists are constructed from:

- the empty list Nil, and
- the construction operation :: (pronounced *cons*); so $x :: xs$ returns a new list with the first element x, followed by the elements of xs.

For example:

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

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

We can thus omit the parentheses in the definition above.

For example:

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

Operations on Lists

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

head	return the first element of the list
tail	return the list composed of all the elements except the first.
isEmpty	return ***true*** iff the list is empty

These operations are defined as methods of objects of type list. For example:

```
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 in ascending order:

- One way to sort the list *List*(*7*, *3*, *9*, *2*) is to sort the tail *List*(*3*, *9*, *2*) to obtain *List*(*2*, *3*, *9*).
- The next step is to insert the head *7* in the right place to obtain 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 a possible implementation of the missing function *insert*?

What is the complexity of insertion sort?

List Patterns

Because :: and *Nil* are both case classes, it is also possible to decompose lists via pattern matching.

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

An alternative is then to rewrite *isort* as follows.

```
def isort(xs: List[Int]): List[Int] = xs match {
  case List() ⇒ List()
  case y :: ys ⇒ insert(y, isort(ys))
}
```

with


```
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 on Lists

By using the list constructors and patterns, we can now formulate other common functions on lists.

The length function

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

```
def length(xs: List[String]): Int = xs match {
  case List() ⇒ 0
  case y :: ys ⇒ 1 + length(ys)
}
scala> length(nums)
4
```

Problem: We cannot apply *length* on lists of strings.

How can we formulate the function so that it is applicable to all lists?

Polymorphism

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

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

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

Syntax:

- We write the type parameters, formal or actual, between brackets. For example: [a], [*Int*].
- We can omit the actual type parameters when they can be inferred from the parameters of the function and the expected result type (which is usually the case).

In our example, we could have also written:

$$length(nums) \qquad /* \ [Int] \ inferred \ given \ that \ nums: List[Int] \ */$$

However, we cannot omit the formal type parameters:

```
scala> def length(x: a) = ...
<console>:4: error: not found: type a
```

Functions which take type parameters are called *polymorphic*.

This word means "which has several forms" in Greek; in fact, the function can be applied to different argument types.

Concatenating Lists

The :: is asymmetric: it is applied to an element of a list and a list.

There also exists the operator ::: (pronounced *concat*) which *concatenates* two lists.

```
scala> List(1, 2) ::: List(3, 4)
List(1, 2, 3, 4)
```

::: can be defined in terms of primitive operations. We 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*?

The *last* and *init* Functions

The method *head* returns the first element of a list. We can write a function that returns the last element of a list in the following way.

```
def last[a](xs: List[a]): a = xs match {
  case List() => error("last of empty list")
  case List(x) => x
  case y :: ys => last(ys)
}
```

Exercice : Write an *init* function which returns all the elements of a list without the last (in other words, *init* and *last* are complementary).

An Aside: Exceptions

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

It is defined as

```
def error(msg: String): Nothing =
  throw new RuntimeException(msg)
```

Note that the function *error* is declared as returning a value of type *Nothing*.

Nothing is a subtype of all other types. There exists no value of this type.

In fact, it indicates that *error* does not return at all.

The *reverse* Function

Here is a function that reverses the elements of a list.

```
def reverse[a](xs: List[a]): List[a] = xs match {
  case List() ⇒ List()
  case y :: ys ⇒ reverse(ys) ::: List(y)
}
```

Q : What is the complexity of *reverse* ?

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

Can we do better? (to solve later).

The List Class

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

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

Note that *List* is a parameterized class.

All the methods in the *List* class are abstract. The implementations of these methods can be found in the two concrete subclasses:

- *Nil* for empty lists.
- :: for non-empty lists.

The *Nil* and :: Classes

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
}
```

More Methods of Lists

The functions presented so far are all methods of the class *List*. For example:


```
abstract class List[a] {
  def head: a
  def tail: List[a]
  def isEmpty: Boolean
  def length = this match {
    case Nil => 0
    case x :: xs => 1 + xs.length
  }
  def init: List[a] = this match {
    case Nil => error("Nil.init")
    case x :: Nil => List()
    case x :: xs => x :: init(xs)
  }
  ...
}
```

The Cons and Concat Operators

Operators whose names end with ':' are treated specially in Scala.

- All operators of this type are right-associative. For example:

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

- All operators of this type are treated as a method of their right operand. For example:

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

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

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

The definition of :: and ::: is now trivial:

```
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   /* ou encore : this.:::(ps).::(p) */
  }
```

Even More Methods of Lists

The $take(n)$ method returns the first n elements of its list (or the list itself if it is shorter than n.)

The $drop(n)$ method returns its list without the first n elements.

The $apply(n)$ returns the n-th element of a list.

They are defined as:


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

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

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

Sorting Lists Faster

As a non-trivial example, design a function to sort itemiz in a list 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,
  1. Separate the list into two sub-lists, each containing around half of the elements of the original list.
  2. Sort the two sub-lists.
  3. Merge the two sorted sub-lists into a single sorted list.

To implement this, we must still specify

- the type of elements to sort
- how to compare two elements

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

```
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))
  }
}
```

Exercice : 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 of the usage of *msort*.

$$\begin{array}{l} \mathit{scala}{>}\ \textbf{def}\ \mathit{iless}(x\colon \mathit{Int},\ y\colon \mathit{Int}) = x < y \\ \mathit{scala}{>}\ \mathit{msort}(\mathit{iless})(\mathit{List}(5,\ 7,\ 1,\ 3)) \\ \mathit{List}(1,\ 3,\ 5,\ 7) \end{array}$$

The definition of *msort* is curried to facilitate its specialization by particular comparison functions.

$$\begin{array}{l} \mathit{scala}{>}\ \textbf{val}\ \mathit{intSort} = \mathit{msort}(\mathit{iless}) \\ \mathit{scala}{>}\ \textbf{val}\ \mathit{reverseSort} = \mathit{msort}((x\colon \mathit{Int},\ y\colon \mathit{Int}) \Rightarrow x > y) \\ \mathit{scala}{>}\ \mathit{intSort}(\mathit{List}(6,\ 3,\ 5,\ 5)) \\ \mathit{List}(3,\ 5,\ 5,\ 6) \\ \mathit{scala}{>}\ \mathit{reverseSort}(\mathit{List}(6,\ 3,\ 5,\ 5)) \\ \mathit{List}(6,\ 5,\ 5,\ 3) \end{array}$$

Complexity:

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

This complexity doesn't depend on the initial distribution of elements in

the list.

Tuples

Tuple2 is the class of Tuples. It can be defined as

```
case class Tuple2[a, b](_1: a, _2: b)
```

As a usage example, here is a function that returns the quotient and remainder of two given whole numbers...

```
def divmod(x: Int, y: Int) = Tuple2(x / y, x % y)
```

And this is how the function can be used:

```
divmod(x, y) match {
  case Tuple2(n, d) => println("quotient: " + n + ", remainder: " + d)
}
```

It is also possible to use the name of the constructor parameters to directly access the elements of a case class. For example:


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

The idea of pairs is generalized in Scala to tuples of larger arities. There exists a case class for each $Tuple_n$ for each n between 2 and 22.

In fact, tuples are so common that there is a special syntax:

The expression or pattern

$(x_1, ..., x_n)$ is an alias for $Tuplen(x_1, ..., x_n)$

The type

$(T_1, ..., T_n)$ is an alias for $Tuplen[T_1, ..., T_n]$

With these abbreviations, the previous example is written as follows:

```
def divmod(x: Int, y: Int): (Int, Int) = (x / y, x % y)
divmod(x, y) match {
  case (n, d) => println("quotient: " + n + ", reste: " + d)
}
```


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.
- These functions are *higher-order functions* that take a transformation or an operator as an argument.

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, we write:

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

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


```
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)
  }
  ...
}
```

In using *map*, *scaleList* can be written more concisely.

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

Exercice : 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 List() => ??
  case y :: ys => ??
}

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


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) y :: posElems(ys) else posElems(ys)
}
```

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

```
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 written more concisely.

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