# PL Lecture No. 7

## 4.4 First-class Functions and Lists

### Map

- Example 4.4.1

```
def inc1 (l: List[Int]): List[Int] = l match {
  case Nil => Nil
  case h :: t => h + 1 :: inc1(t)
}

def square (l: List[Int]): List[Int] = l match {
  case Nil => Nil
  case h :: t => h * h :: square(t)
}
```

- Given list and function, the map function generates a new list by mapping the given function to each element

- Example 4.4.2

```
def map (l : List[Int], f: Int=>Int) : List [Int] = {
  l match {
    case Nil => Nil
    case h::t => f(h)::map(t, f)
  }
}

def inc1 (l: List[Int]): List[Int] = map(l, h => h + 1)
def square (l: List[Int]) : List[Int] = map(l, h => h * h)
```

## Filter

- Example 4.4.3

```
def odd (l: List[Int]) : List[Int] = {
  l match {
     case Nil => Nil
     case h::t => if (h % 2 != 0) h::odd(t) else odd(t)
  }
}

def positive (l: List[Int]) : List[Int] = {
  l match {
     case Nil => Nil
     case h::t => if (h > 0) h::positive(t) else positive(
  }
}
```

- Example 4.4.4

```
def filter (l : List[Int], f : Int=>Boolean) : List[Int] =
  l match {
    Nil => Nil
    h::t => if (f(h)) h::filter(t) else filter(t)
  }
}

// odd
// positive
```

## Fold (reduce)

- Example 4.4.5

```
def sum (l: List[Int]): Int = {
  l match {
    case Nil => 0
    case h :: t => h + sum(t)
  }
}

def product (l: List[Int]): Int = {
  l match {
    case Nil => 1
    case h :: t => h * product(t)
  }
}
```

- Example 4.4.6

```
def fold (l : List[Int], n: Int, f: (Int, Int) => Int) : I
  l match {
    case Nil => n
    case h::t => f(h, fold(t))
  }
}

//sum
//product
```

- Use `fold` to construct function `digitToDecimal : List[Int] ⇒ Int` which returns the integer having the integers in the given list as decimals in order.

▼ Example 4.4.7

```
def digitToDecimal (l: List[Int]) : Int = {
  def aux (l: List[Int], inter: Int) : Int = {
    l match {
```

```
      case Nil => inter
      case h::t => aux(t, 10 * inter + h)
    }
  }
  aux(l, 0)
}
```

▼ Example 4.4.8

```
def foldLeft (l: List[Int], n: Int, f: (Int, Int) => Int):
  def aux (l: List[Int], inter: Int): Int = l match {
    case Nil => inter
    case h :: t => aux(t, f(inter, h))
  }
  aux(l, n)
}
//digitToDecimal
```

▼ Example 4.4.9

```
def digitToDecimal (l: List[Int]) = {
  def aux (l: List[Int], inter: Int) : Int = {
    l match {
      case Nil => inter
      case h::t => aux(t, 10 * inter + h)
    }
  }
  aux(l, 0)
}

def sum (l: List[Int]): Int = {
  def aux (l: List[Int], inter: Int): Int = l match {
    case Nil => inter
    case h :: t => aux(t, inter + h)
  }
```

```
  aux(l, 0)
}

def product (l: List[Int]): Int = {
  def aux (l: List[Int], inter: Int): Int = l match {
    case Nil => inter
    case h :: t => aux(t, inter * h)
  }
  aux(l, 1)
}
```

- The Scala standard library provides `map`, `filter`, `foldRight`, `foldLeft` as the methods of `List`

```
def inc1 (l: List[Int]): List[Int] = l.map(h => h + 1)
def square (l: List[Int]): List[Int] = l.map(h => h * h)

def odd(l: List[Int]): List[Int] = l.filter(_ % 2 != 0)
def positive(l: List[Int]): List[Int] = l.filter(_ > 0)

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

## 4.5. For-loops

- a for-loop is an expression which evaluates to a collection containing the results of evaluating a given expression at each iteration

```
for ([name] <- [expression])
  yield [expression]
```

```
for ([name] <- [expression] if [expression])
  yield [expression]
```

- Example 4.5.1

```
val l1 = for (n <- List(0, 1, 2)) yield n * n
//l1 == List(0, 1, 4)

val l2 = for (n <- List(1, 2, 3, 4, 5, 6) if n % 2 == 0) y
//l2 == List(1, 2, 3)
```

### 5.1-5.2 Algebraic Data Types & Advantages : Pattern Matching

- Algebraic Data Type (ADT) expresses a type that includes values of different shapes:
  - a binary tree is an empty tree, a tree containing a root element and a left-child tree, a tree containing a root element and a right-child tree, or a tree containing a root element and two children trees.
  - an arithmetic expression is a a number, a variable, the sum of two arithmetic expressions, or the difference of two arithmetic expressions.
- An ADT is the sum type of product types
  - a product type has an element as an enumeration of values of types in the same specific order (e.g., tuple)
  - a sum type has values of multiple types (variants) as its values
    - a variant is given with a tagged name
- Ex. Arithmetic Expression (AE)
  - AE has three variants:
    - a number
    - the sum of two arithmetic expressions,
    - the difference of two arithmetic expressions
  - AE is the sum type of
    - `Int` (tagged with Num)
    - `AE + AE` (tagged with Add)
    - `AE * AE` (tagged with Sub)
- In Scala, a new type can be defined as trait

    - syntax: `trait [type-name]`
    - once a type is defined as a trait, the type can be used just like any other types.
- Ex. Arithmetic Expression AE

```
trait AE

case class Num (value: Int) extends AE
case class Add (left: AE, right: AE) extends AE
case class Sub (left: AE, right: AE) extends AE

val n = Num(10)
val m = Num(5)
val e1 = Add(n, m)
val e2 = Sub(e1, Num(3))

def identity (ae: AE): AE = ae
```

- Use pattern matching to access the value of a newly created type
    - pattern matching compares a value to patterns sequentially from top to bottom and selects the first matching pattern
        - exhaustivity checking
        - reachability checking
    - Example

```
def eval (e: AE) : Int = {
  e match {
    case Num(n) => n
    case Add(l, r) => eval(l) + eval(r)
    case Sub(l, r) => eval(l) - eval(r)
  }
}
#exhaustivity checking
#reachability checking
assert(eval(Sub(Add(Num(3), Num(7)), Num(5))) == 5)
```

- Without pattern matching, handling ADTs would be complicated since it would involve many dynamic type checking

  ```
  def eval(e: AE): Int = {
    if (e.isInstanceOf[Num])
      e.asInstanceOf[Num].value
    else if (e.isInstanceOf[Add]) {
      val e0 = e.asInstanceOf[Add]
      eval(e0.left) + eval(e0.right)
    } else {
      val e0 = e.asInstanceOf[Sub]
      eval(e0.left) - eval(e0.right)
    }
  }
  ```

## 5.3 Patterns in Scala

- pattern matching is a general form of switch-case
  - underscore(_) matches every value (i.e., the wildcard)
  - Example

    ```
    def grade(score: Int): String = {
      (score / 10) match {
        case 10 => "A"
        case 9 => "A"
        case 8 => "B"
        case 7 => "C"
        case 6 => "D"
        case _ => "F"
      }
    }
    ```

- Or-pattern

  ```
  def grade(score: Int): String = {
    (score / 10) match {
  ```

```
    case 10 | 9 => "A"
    case 9 => "A"
    case 8 => "B"
    case 7 => "C"
    case 6 => "D"
    case _ => "F"
  }
}
```

- Nested Patterns

```
def optimizeAdd (e: AE): AE = e match {
  case Num(_) => e
  case Add(Num(0), r) => optimizeAdd(r)
  case Add(l, Num(0)) => optimizeAdd(l)
  case Add(l, r) => Add(optimizeAdd(l), optimizeAdd(r))
  case Sub(l, r) => Sub(optimizeAdd(l), optimizeAdd(r))
}
```

- Type pattern

```
case class Abs(e: AE) extends AE

def optimizeAbs(e: AE): AE = e match {
  case _: Num => e
  case Add(l, r) => Add(optimizeAbs(l), optimizeAbs(r))
  case Sub(l, r) => Sub(optimizeAbs(l), optimizeAbs(r))
  case Abs(e0 @ Abs(_)) => optimizeAbs(e0)
  case Abs(e0) => Abs(optimizeAbs(e0))
}
```

- Tuple patterns

```
def equal (l0: List[Int], l1 : List[Int]) : Boolean = {
    (l0, l1) match {
      case (h0::t0, h1::t1) => h0 == h1 && equal(t0, t1)
      case (Nil, Nil) => true
      case _ => false
```

```
        }
}
```