- récursion sur des `IntList`, ne pas oublier `IntCons(head, tail) =>` et pas `head :: tail ->` comme dans une liste normale.
- on peut faire un `case IntCons(head, tail) if head > 0 =>` (ajouter un test supplémentaire dans le case).
- on utilise pas `Math.max` mais `scala.math.max`.
- ne pas oublier que les chaînes de caractères `String` disposent des propriétés `.head`, `.tail`, etc.
- ne pas oublier les `.toList` sur les `for comprehensions`.
- on peut créer des contextes :

```
enum Context:
  case Empty
  case Cons(name: String, value: Int, rem: Context)
```

Une clef associée à une valeur dans un contexte est appelée un “Binding”.

Folds

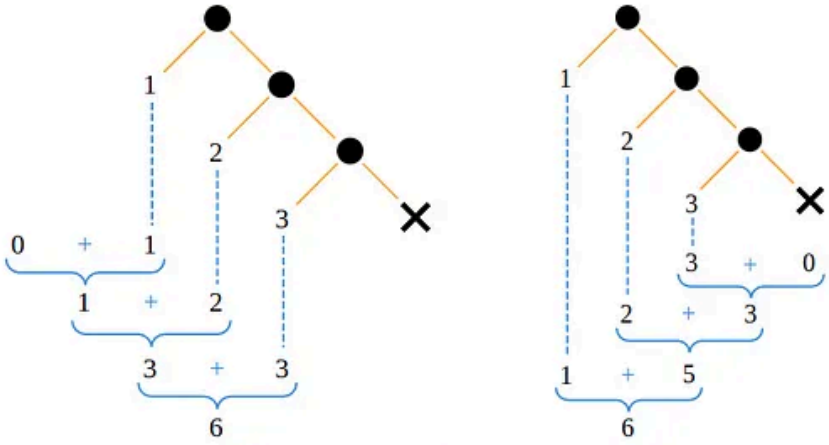
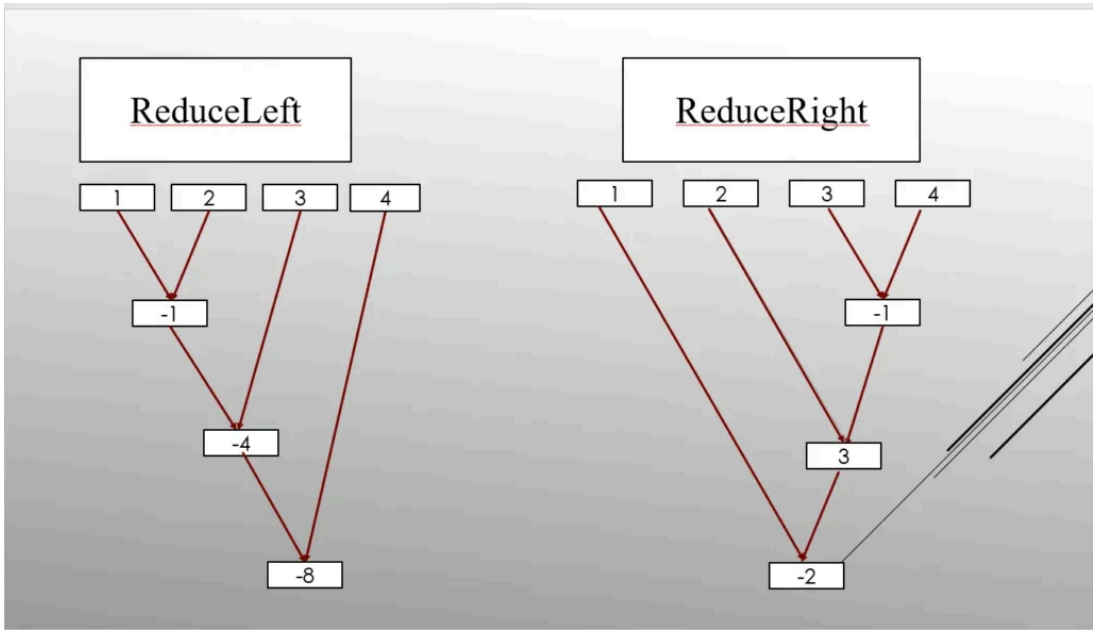


Figure 11: Illustration of foldLeft and foldRight

Le fold right demande une base (qui sera un accumulateur), ici c’est 0.

Reducers



Scala docs

Traits, Class, companion objects

Un trait est similaire à une interface Java :

```
trait EtreVivant {
  def respirer(): String
}
```

Une classe peut implémenter un ou plusieurs traits :

```
class Humain(val nom: String) extends EtreVivant {
  def respirer(): String = s"$nom est un humain qui respire."

  def marcher(): String = s"$nom est en train de marcher."
}
```

Un companion object permet de définir l'équivalent des méthodes statiques en Java :

```
object Humain {
  def apply(nom: String): Humain = new Humain(nom)

  // l'équivalent d'une méthode statique en Java
  def descriptionGenerale(): String = "Les humains sont des êtres vivants qui
respirent et peuvent marcher."
}
```

Case class

```
case class WordCountState(count: Int, lastWasWS: Boolean)
```

automatiquement l'égalité par valeur (pas référence)

Variance

Covariance

```
enum MyList[+A] {
  case Nil
```

```
  case Cons(head: A, tail: MyList[A])
}
```

ça nous permet d'écrire :

```
MyList[Chien] listeChiens = ... // blabla
MyList[Animal] listeAnimaux = listeChiens;
// compile ! ne compilerait pas en Java, ou sans le `+`.
```

Contravariance

```
class Veterinaire[-A] {
  def heal(animal: A): Unit = animal.heal()
}
```

ça nous permet d'écrire :

```
class Animal {
  def heal(): Unit = println("Animal heals")
}

class Dog extends Animal {
  override def heal(): Unit = println("Dog heals")
}

val animalVeterinaire: Veterinaire[Animal] = new Veterinaire[Animal]
val dogVeterinaire: Veterinaire[Dog] = animalVeterinaire

dogVeterinaire.heal(new Dog()) // accepte des dog!
```

Invariance

```
class Container[A](value: A) {
  def get: A = value
}
```

Par défaut, les types en scala sont invariants :

```
val dogContainer: Container[Dog] = new Container(new Dog)
val animalContainer: Container[Animal] = dogContainer  // Erreur de compilation
```

Bounds

```
S <: IntSet // S est un sous-type de IntSet ou un InSet lui-même
S >: IntSet // S est un super-type de IntSet ou un IntSet lui-même
S >: NonEmpty <: IntSet // S est un super-type de NonEmpty mais un sous-type de
IntSet
```