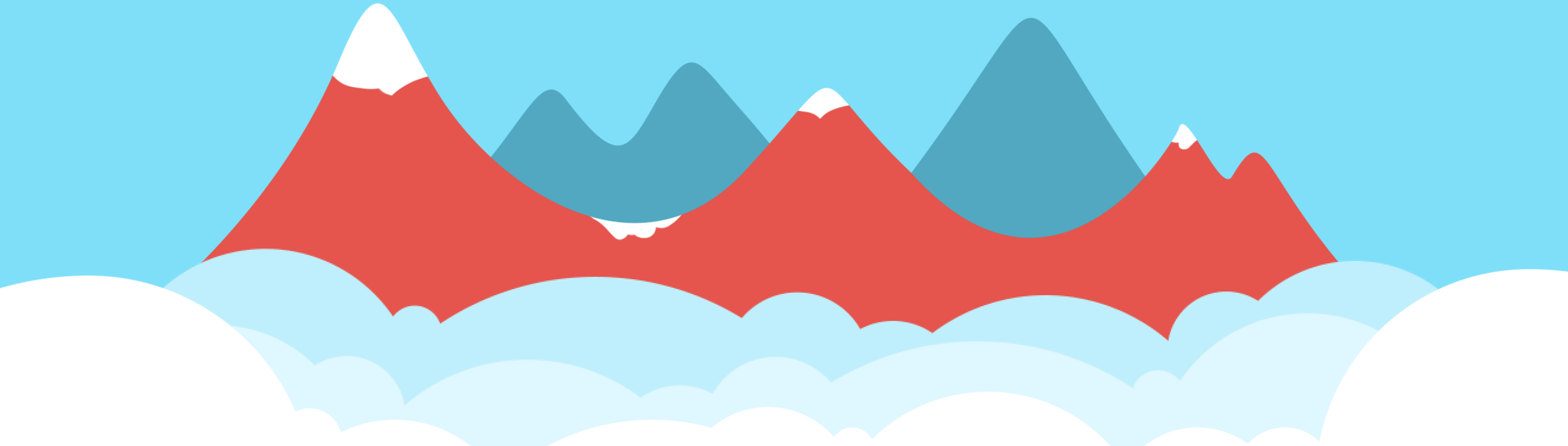
Scala **Enthusiasts** BS

Arne Brüsch – Philipp Wille

Pattern Matching Syntax



Scala

- **Modular programming** language
  - Some used to call it objectfunctional

Scala

- **Modular programming** language
  - Some used to call it objectfunctional
- Main designer and architect is Prof. **Martin Odersky**
  - Scala is an academic language
- Works on the **JVM** like Java

Scala

- **Modular programming** language
  - Some used to call it objectfunctional
- Main designer and architect is Prof. **Martin Odersky**
  - Scala is an academic language
- Works on the **JVM** like Java
- Influenced by (among others):
  - Object-oriented: Java, Smalltalk

Scala

- **Modular programming** language
  - Some used to call it objectfunctional
- Main designer and architect is Prof. **Martin Odersky**
  - Scala is an academic language
- Works on the **JVM** like Java
- Influenced by (among others):
  - Object-oriented: Java, Smalltalk
  - Functional: Haskell, Lisp, Scheme

Scala

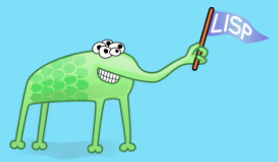
- **Modular programming** language
  - Some used to call it objectfunctional
- Main designer and architect is Prof. **Martin Odersky**
  - Scala is an academic language
- Works on the **JVM** like Java
- Influenced by (among others):
  - Object-oriented: Java, Smalltalk
  - Functional: Haskell, Lisp, Scheme
  - Concurrent: Erlang

Typesafe

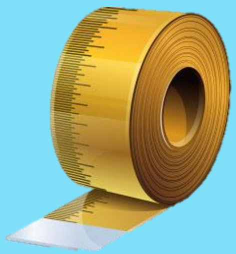
Scala

- **Modular programming** language
  - Some used to call it objectfunctional
- Main designer and architect is Prof. **Martin Odersky**
  - Scala is an academic language
- Works on the **JVM** like Java
- Influenced by (among others):
  - Object-oriented: Java, Smalltalk
  - Functional: Haskell, Lisp, Scheme
  - Concurrent: Erlang
- Commercially marketed by Odersky‘s **Typesafe Inc.**

What do we talk about?



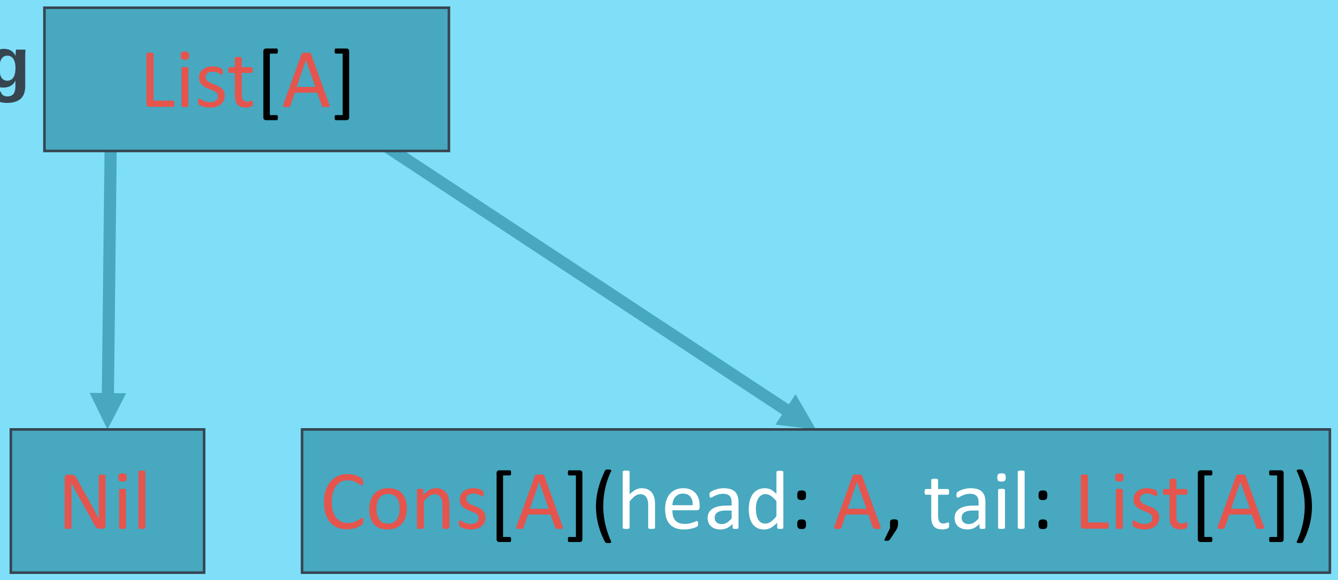
1. Algebraic Data Types & Pattern Matching
2. Implementing Algebraic Data Types



3. Pattern Matching Syntax
4. More Scala Features

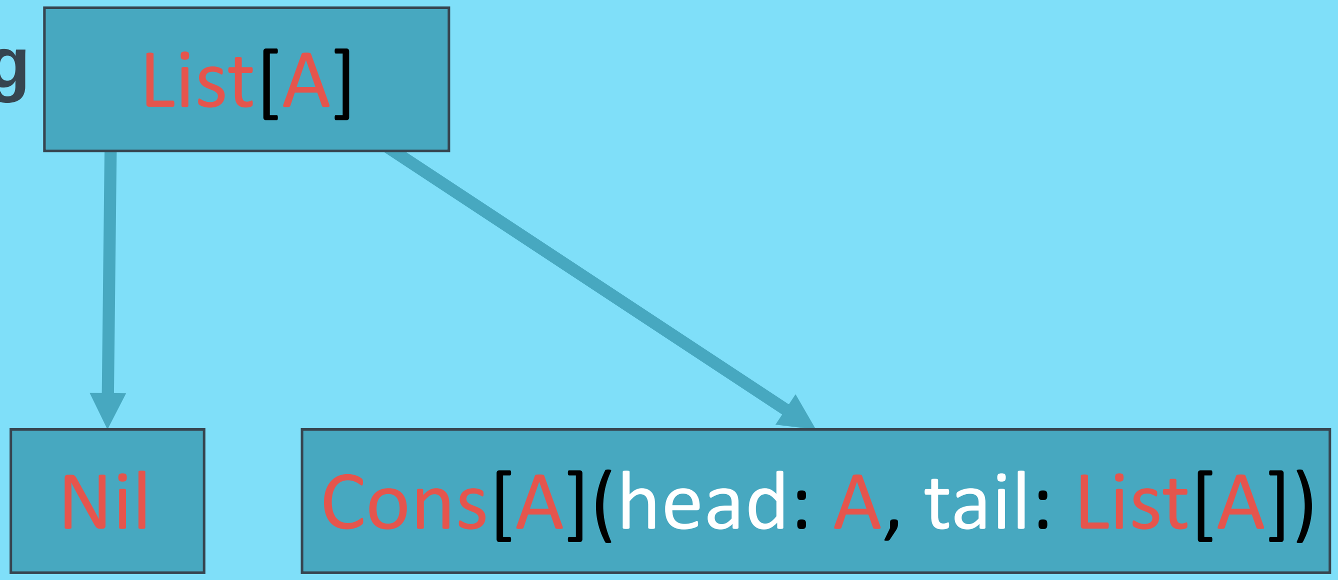
(Generalized) Algebraic Data Types (ADTs)

- Most **functional programming** languages work with ADTs
- ADTs are types formed by a **combination of other types**
- The most common example is the **singly linked list**:



(Generalized) Algebraic Data Types (ADTs)

- Most **functional programming** languages work with ADTs
- ADTs are types formed by a **combination of other types**
- The most common example is the **singly linked list**:




```
Cons(1, Cons(2, Cons(3, Nil)))
```

Pattern Matching

- **Check** a given ADT for an **exact pattern**

```
Cons(1, Cons(2, Cons(3, Nil)))
```

Pattern Matching

- **Check** a given ADT for an **exact pattern**
- Similar to Java‘s Switch/Case statement

```
Cons(1, Cons(2, Cons(3, Nil))) match {
  case Cons(1, tail) => println(“one”)
  case Cons(2, tail) => println(“two”)
}
```

Pattern Matching

- Most functional languages use a special notation for lists:

Pattern Matching

- Most functional languages use a special notation for lists:
  - Expresses the right-associativity of singly linked lists

```
1 :: 2 :: 3 :: Nil match {
  case 1 :: tail => println(“one”)
  case 2 :: tail => println(“two”)
}
```

List Notation

ML

List Notation

```
(cons 1 (cons 2 (cons 3 (cons 4 (cons 5 nil)))))
```

ML

List Notation

```
(cons 1 (cons 2 (cons 3 (cons 4 (cons 5 nil)))))
```

ML

```
cons (1, cons (2, cons (3, cons (4, cons (5, nil)))))
```

List Notation

```
(cons 1 (cons 2 (cons 3 (cons 4 (cons 5 nil)))))
```

ML

```
cons (1, cons (2, cons (3, cons (4, cons (5, nil)))))
```

```
Cons 1 (Cons 2 (Cons 3 (Cons 4 (Cons 5 Nil))))
```

List Notation

```
Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Nil)))))
```

```
(cons 1 (cons 2 (cons 3 (cons 4 (cons 5 nil)))))
```

ML

```
cons (1, cons (2, cons (3, cons (4, cons (5, nil)))))
```

```
Cons 1 (Cons 2 (Cons 3 (Cons 4 (Cons 5 Nil))))
```

List Notation

ML

List Notation

`(1 2 3 4 5)`

ML

List Notation

(1 2 3 4 5)

ML `1 :: 2 :: 3 :: 4 :: 5 :: nil`

List Notation

```
(1 2 3 4 5)
```

ML
```
1 :: 2 :: 3 :: 4 :: 5 :: nil
```

```
1 : 2 : 3 : 4 : 5 : Nil
```

List Notation

```
1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

```
(1 2 3 4 5)
```

ML
```
1 :: 2 :: 3 :: 4 :: 5 :: nil
```

```
1 : 2 : 3 : 4 : 5 : Nil
```

List Notation

ML

List Notation

```
(list 1 2 3 4 5)
```

ML

List Notation

```
(list 1 2 3 4 5)
```

ML `[1; 2; 3; 4; 5]`

List Notation

(list 1 2 3 4 5)

ML [1; 2; 3; 4; 5]

[1, 2, 3, 4, 5]

List Notation

```
List(1, 2, 3, 4, 5)
```

```
(list 1 2 3 4 5)
```

ML

```
[1; 2; 3; 4; 5]
```

```
[1, 2, 3, 4, 5]
```

I love LISTS.

Reimplementing `List`

- `Cons` and `Nil` are *Data Constructors* for List

Reimplementing `List`

- `Cons` and `Nil` are *Data Constructors* for List
- Basic class structure:

```scala
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

Reimplementing List

- `Cons` and `Nil` are *Data Constructors* for List
- Basic class structure:

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

- This gives us the following notation:

```
val list = Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Nil)))))
```

Reimplementing List

```scala
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

- Pattern matching the list:

```scala
list match {
  case Cons(h, t) => ...
}
```

Alternate `List` Notation

- We still need two additional notations:

Alternate `List` Notation

- We still need two additional notations:

```
val list = List(1, 2, 3, 4, 5)
val list = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

Alternate `List` Notation

- We still need two additional notations:

```
val list = List(1, 2, 3, 4, 5)
val list = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

- The first notation hides the implementation details from the user
  - The user does not need to know about `Cons` or `Nil`

Alternate List Notation

```scala
object List {
  def apply[A](as: A*): List[A] =
    if(as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))
}
```

Alternate `List` Notation

```
object List {
  def apply[A](as: A*): List[A] =
    if(as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))
}
```

- The `List` companion objects enables the following notations:

```
val list = List.apply(1, 2, 3, 4, 5)
```

Alternate `List` Notation

```scala
object List {
  def apply[A](as: A*): List[A] =
    if(as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))
}
```

- The `List` companion objects enables the following notations:

```scala
val list = List.apply(1, 2, 3, 4, 5)
val list = List(1, 2, 3, 4, 5)
```

Better DSL for `Lists`

- We have two ways to define `Lists` now:

```
val list = Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Nil)))))
val list = List(1, 2, 3, 4, 5)
```

RIGHT!

Better DSL for `Lists`

- We have two ways to define `Lists` now:

```
val list = Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Nil)))))
val list = List(1, 2, 3, 4, 5)
```

- But how can we implement the last one?

```
val list = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

Better DSL for `Lists`

- We have two ways to define `Lists` now:

```
val list = Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Nil)))))
val list = List(1, 2, 3, 4, 5)
```

- But how can we implement the last one?

```
val list = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

- Overloaded **operator**

RIGHT!

Better DSL for `Lists`

- We have two ways to define `Lists` now:

```
val list = Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Nil)))))
val list = List(1, 2, 3, 4, 5)
```

- But how can we implement the last one?

```
val list = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

- Overloaded **operator**
- **Right-associative**

RIGHT!

Nicer DSL for `Lists`

- Let us first look at a similar DSL:

```
val list = Nil.after(5).after(4).after(3).after(2).after(1)
```

Nicer DSL for `Lists`

- Let us first look at a similar DSL:

```
val list = Nil.after(5).after(4).after(3).after(2).after(1)
```

- In Scala, methods with just one parameter need **no brackets**

```
val list = Nil after 5 after 4 after 3 after 2 after 1
```

Nicer DSL for `Lists`

- Let us first look at a similar DSL:

```scala
val list = Nil.after(5).after(4).after(3).after(2).after(1)
```

- In Scala, methods with just one parameter need **no brackets**

```scala
val list = Nil after 5 after 4 after 3 after 2 after 1
```

- But `after` is still **left-associative**…

Nicer DSL for `Lists`

- In Scala, methods with just one parameter need **no brackets**

```
val list = Nil after 5 after 4 after 3 after 2 after 1
```

Nicer DSL for `Lists`

- In Scala, methods with just one parameter need **no brackets**

```
val list = Nil after 5 after 4 after 3 after 2 after 1
```

- It can be implemented as follows:

```
sealed trait List[+A] {
  def after[B >: A](head: B): List[B] = Cons(head, this)
}
```

Nicer DSL for `Lists`

- In Scala, methods with just one parameter need **no brackets**

```
val list = Nil after 5 after 4 after 3 after 2 after 1
```

- It can be implemented as follows:

```
sealed trait List[+A] {
  def after[B >: A](head: B): List[B] = Cons(head, this)
}
```

- But how to define right-associative functions?

Nicer DSL for `Lists`

- Scala methods ending with a `:` are always **right-associative**

Nicer DSL for `Lists`

- Scala methods ending with a `:` are always **right-associative**

```
sealed trait List[+A] {
  def after[B >: A](head: B): List[B] = Cons(head, this)
  def ::[B >: A](head: B): List[B] = this.after(head)
}
```

Nicer DSL for Lists

- Scala methods ending with a : are always **right-associative**

```scala
sealed trait List[+A] {
  def after[B >: A](head: B): List[B] = Cons(head, this)
  def ::[B >: A](head: B): List[B] = this.after(head)
}
```

- Which gives us our last notation

```scala
val list = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

Pattern Matching for `Lists`

- For now, we can only pattern match lists like this:

```
list match { case Cons(head, tail) => ... }
```

Pattern Matching for `Lists`

- For now, we can only pattern match lists like this:

```
list match { case Cons(head, tail) => ... }
```

- But how can we enable this notation?

```
val list = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
list match { case head :: tail => ... }
```

Pattern Matching for `Lists`

- For now, we can only pattern match lists like this:

```
list match { case Cons(head, tail) => ... }
```

- But how can we enable this notation?

```
val list = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
list match { case head :: tail => ... }
```

- Let‘s have a closer look at how pattern matching in Scala works…

Example 1: Sum of a List

- For a start we implement the sum of a list using Pattern Matching:

```scala
def sum(list: List[Int]): Int = list match {
  case Cons(head, tail) => head + sum(tail)
  case Nil => 0
}

sum(1 :: 2 :: 3 :: 4 :: 5 :: Nil)
```

Σ

Example 1: Sum of a List

- For a start we implement the sum of a list using Pattern Matching:

```
def sum(list: List[Int]): Int = list match {
  case Cons(head, tail) => head + sum(tail)
  case Nil => 0
}

1 + sum(2 :: 3 :: 4 :: 5 :: Nil)
```

Σ

Example 1: Sum of a List

- For a start we implement the sum of a list using Pattern Matching:

```
def sum(list: List[Int]): Int = list match {
  case Cons(head, tail) => head + sum(tail)
  case Nil => 0
}

1 + 2 + sum(3 :: 4 :: 5 :: Nil)
```

Σ

Example 1: Sum of a List

- For a start we implement the sum of a list using Pattern Matching:

```
def sum(list: List[Int]): Int = list match {
  case Cons(head, tail) => head + sum(tail)
  case Nil => 0
}
```

```
1 + 2 + 3 + sum(4 :: 5 :: Nil)
```

Example 1: Sum of a List

- For a start we implement the sum of a list using Pattern Matching:

```scala
def sum(list: List[Int]): Int = list match {
  case Cons(head, tail) => head + sum(tail)
  case Nil => 0
}

1 + 2 + 3 + 4 + sum(5 :: Nil)
```

Σ

Example 1: Sum of a List

- For a start we implement the sum of a list using Pattern Matching:

```scala
def sum(list: List[Int]): Int = list match {
  case Cons(head, tail) => head + sum(tail)
  case Nil => 0
}

1 + 2 + 3 + 4 + 5 + sum(Nil)
```

Example 1: Sum of a List

- For a start we implement the sum of a list using Pattern Matching:

```
def sum(list: List[Int]): Int = list match {
  case Cons(head, tail) => head + sum(tail)
  case Nil => 0
}

1 + 2 + 3 + 4 + 5 + 0
```

Σ

Example 1: Sum of a List

- For a start we implement the sum of a list using Pattern Matching:

```scala
def sum(list: List[Int]): Int = list match {
  case Cons(head, tail) => head + sum(tail)
  case Nil => 0
}

15
```

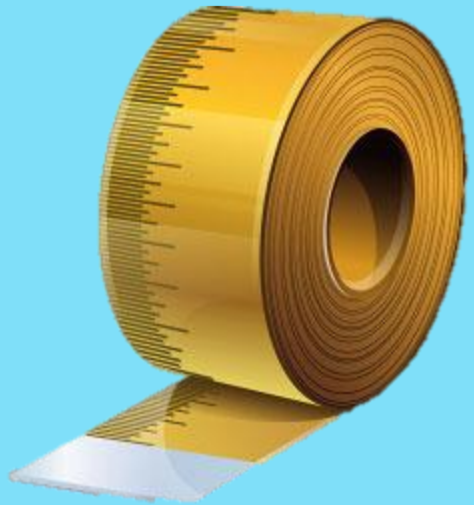
Σ

Example 2: Length of a List

- Another useful operation is the length of a given list:

```
def length[A](l: List[A]): Int = l match {
  case Nil => 0
  case cons: Cons => 1 + length(cons.tail)
}

length(1 :: 2 :: 3 :: 4 :: 5 :: Nil)
```

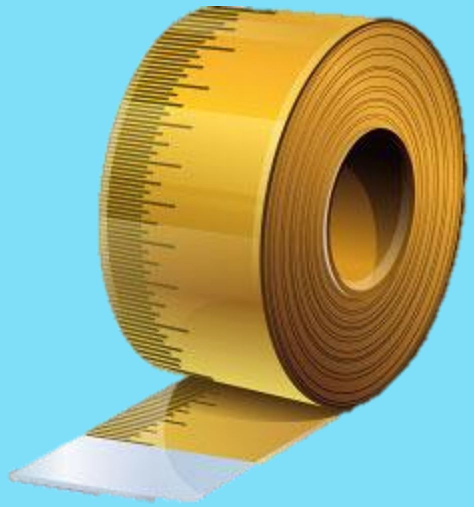


Example 2: Length of a List

- Another useful operation is the length of a given list:

```
def length[A](l: List[A]): Int = l match {
  case Nil => 0
  case cons: Cons => 1 + length(cons.tail)
}

1 + length(2 :: 3 :: 4 :: 5 :: Nil)
```

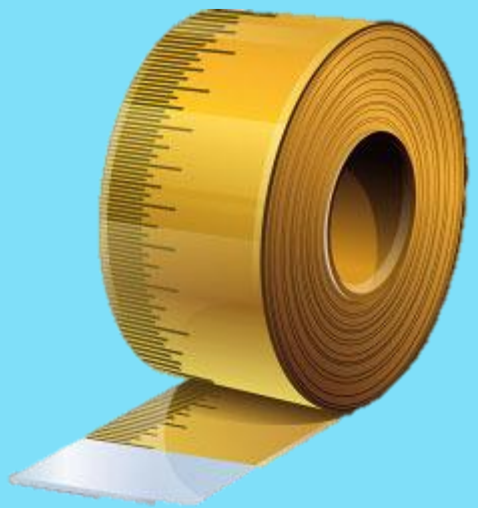


Example 2: Length of a List

- Another useful operation is the length of a given list:

```
def length[A](l: List[A]): Int = l match {
  case Nil => 0
  case cons: Cons => 1 + length(cons.tail)
}

1 + 1 + length(3 :: 4 :: 5 :: Nil)
```

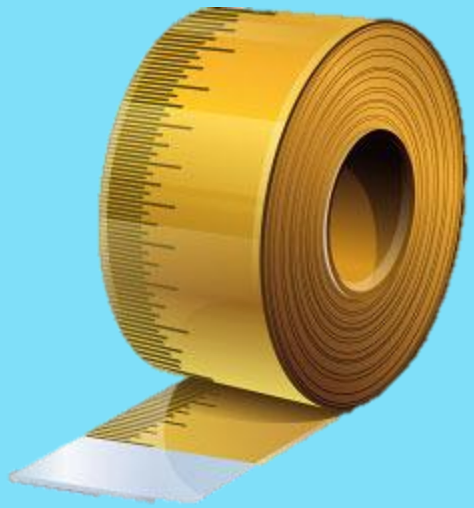


Example 2: Length of a List

- Another useful operation is the length of a given list:

```
def length[A](l: List[A]): Int = l match {
  case Nil => 0
  case cons: Cons => 1 + length(cons.tail)
}

1 + 1 + 1 + length(4 :: 5 :: Nil)
```

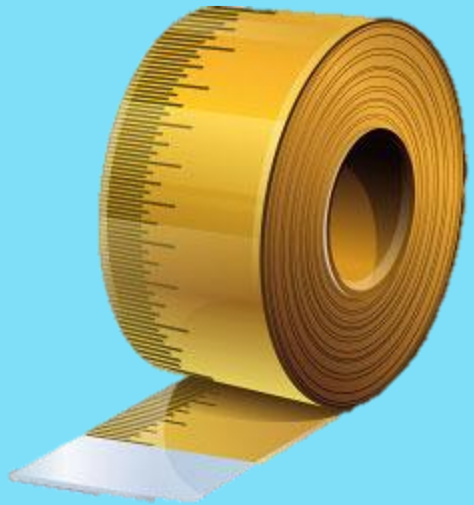


Example 2: Length of a List

- Another useful operation is the length of a given list:

```
def length[A](l: List[A]): Int = l match {
  case Nil => 0
  case cons: Cons => 1 + length(cons.tail)
}

1 + 1 + 1 + 1 + length(5 :: Nil)
```

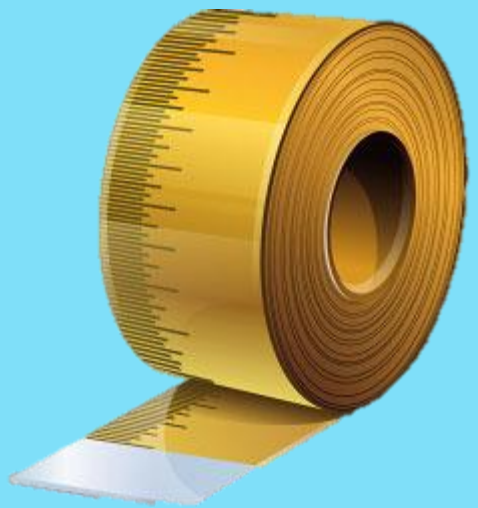


Example 2: Length of a List

- Another useful operation is the length of a given list:

```
def length[A](l: List[A]): Int = l match {
  case Nil => 0
  case cons: Cons => 1 + length(cons.tail)
}

1 + 1 + 1 + 1 + 1 + length(Nil)
```

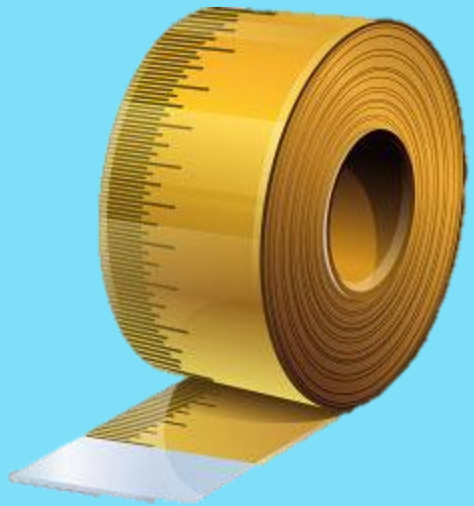


Example 2: Length of a List

- Another useful operation is the length of a given list:

```
def length[A](l: List[A]): Int = l match {
  case Nil => 0
  case cons: Cons => 1 + length(cons.tail)
}

1 + 1 + 1 + 1 + 1 + 0
```

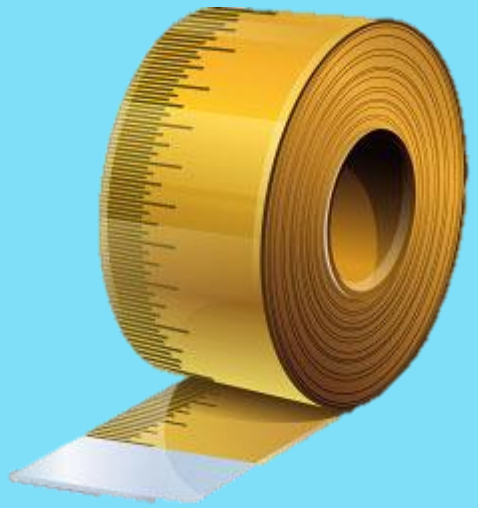


Example 2: Length of a List

- Another useful operation is the length of a given list:

```
def length[A](l: List[A]): Int = l match {
  case Nil => 0
  case cons: Cons => 1 + length(cons.tail)
}
```

5



Example 3: Take n first Elements

- Pattern Matching also allows us to select Elements from a list

```
def take[A](n: Int, ls: List[A]): List[A] = ls match {
  case Cons(hd, tl) if n > 0 => hd :: take(n-1, tl)
  case _ => Nil
}

take(2, 1 :: 2 :: 3 :: 4 :: 5 :: Nil)
```

Example 3: Take n first Elements

- Pattern Matching also allows us to select Elements from a list

```
def take[A](n: Int, ls: List[A]): List[A] = ls match {
  case Cons(hd, tl) if n > 0 => hd :: take(n-1, tl)
  case _ => Nil
}

1 :: take(1, 2 :: 3 :: 4 :: 5 :: Nil)
```

Example 3: Take n first Elements

- Pattern Matching also allows us to select Elements from a list

```
def take[A](n: Int, ls: List[A]): List[A] = ls match {
  case Cons(hd, tl) if n > 0 => hd :: take(n-1, tl)
  case _ => Nil
}

1 :: 2 :: take(0, 3 :: 4 :: 5 :: Nil)
```

Example 3: Take n first Elements

- Pattern Matching also allows us to select Elements from a list

```
def take[A](n: Int, ls: List[A]): List[A] = ls match {
  case Cons(hd, tl) if n > 0 => hd :: take(n-1, tl)
  case _ => Nil
}
```

```
1 :: 2 :: Nil
```

Pattern Matching Syntactically

- Pattern Matching is a very self-contained part of the Scala programming language. This is where Pattern Matching lives:

```
Expr ::= PostfixExpr `match' `{' CaseClauses `}'
CaseClauses ::= CaseClause {CaseClause}
CaseClause ::= `case' Pattern [Guard] `=>' Block
```

- A more abstract syntax

```
e match { case p1 => b1 … case pn => bn }
```

Guard Syntax

- Remember take?

```
def take[A](n: Int, ls: List[A]): List[A] = ls match {
  case Cons(hd, tl) if n > 0 => hd :: take(n - 1, tl)
  case _ => Nil
}
```

```
Guard ::=  `if' BExpr
```

```
Guarded Patterns are applied iff
The Pattern matches and
The Guard yields true
```

```
Expr ::= PostfixExpr `match' `{' CaseClauses `}'
CaseClauses ::= CaseClause {CaseClause}
CaseClause ::= `case' Pattern [Guard] `=>' Block
```

Pattern Syntax

```
Pattern ::= Pattern1 { ‘|’ Pattern1 }
Pattern1 ::= varid ‘:’ TypePat | ‘_’ ‘:’ TypePat | Pattern2
Pattern2 ::= varid [‘@’ Pattern3] | Pattern3
Pattern3 ::= SimplePattern | SimplePattern {id [nl] SimplePattern}
SimplePattern ::= ‘_’ |
                  varid |
                  Literal |
                  StableId |
                  StableId ‘(’ [Patterns] ‘)’ |
                  StableId ‘(’ [Patterns ‘,’] [varid ‘@’] ‘_’ ‘*’ ‘)’ |
                  ‘(’ [Patterns] ‘)’ |
                  XmlPattern
```

Typed Pattern

```
Pattern1 ::= varid ‘:’ TypePat |
             ‘_’ ‘:’ TypePat | Pattern2
```

```scala
def length[A](l: List[A]): Int = l match {
  case Nil => 0
  case cons: Cons => 1 + length(cons.tail)
}
```

```
Pattern ::= Pattern1 { ‘|’ Pattern1 }
Pattern1 ::= varid ‘:’ TypePat | ‘_’ ‘:’ TypePat | Pattern2
Pattern2 ::= varid [‘@’ Pattern3] | Pattern3
Pattern3 ::= SimplePattern | SimplePattern {id [nl] SimplePattern}
SimplePattern ::= ‘_’ |
                  varid |
                  Literal |
                  StableId |
                  StableId ‘(’ [Patterns] ‘)’ |
                  StableId ‘(’ [Patterns ‘,’] [varid ‘@’] ‘_’ ‘*’ ‘)’ |
                  ‘(’ [Patterns] ‘)’ |
                  XmlPattern
```

Pattern Binders

```
Pattern2 ::= varid [‘@’ Pattern3] | Pattern3

def length[A](list: List[A]): Int = list match {
  case Nil => 0
  case c @ Cons(head, tail) => 1 + length(c.tail)
}
```

```
Pattern ::= Pattern1 { ‘|’ Pattern1 }
Pattern1 ::= varid ‘:’ TypePat | ‘_’ ‘:’ TypePat | Pattern2
Pattern2 ::= varid [‘@’ Pattern3] | Pattern3
Pattern3 ::= SimplePattern | SimplePattern {id [nl] SimplePattern}
SimplePattern ::= ‘_’ |
                  varid |
                  Literal |
                  StableId |
                  StableId ‘(’ [Patterns] ‘)’ |
                  StableId ‘(’ [Patterns ‘,’] [varid ‘@’] ‘_’ ‘*’ ‘)’ |
                  ‘(’ [Patterns] ‘)’ |
                  XmlPattern
```

Extractor Pattern

```
SimplePattern ::= StableId '(' [Patterns] ')' |
```

```scala
def take[A](n: Int, ls: List[A]): List[A] = ls match {
  case Cons(head, tail) if n > 0 => {
    head :: take(n - 1, tail)
  }
  case _ => Nil
}
```

```
Pattern ::= Pattern1 { '|' Pattern1 }
Pattern1 ::= varid ':' TypePat | '_' ':' TypePat | Pattern2
Pattern2 ::= varid ['@' Pattern3] | Pattern3
Pattern3 ::= SimplePattern | SimplePattern {id [nl] SimplePattern}
SimplePattern ::= '_' |
                  varid |
                  Literal |
                  StableId |
                  StableId '(' [Patterns] ')' |
                  StableId '(' [Patterns ','] [varid '@'] '_' '*' ')' |
                  '(' [Patterns] ')' |
                  XmlPattern
```

Custom Extractors

- Remember? We wanted to use the following extractor syntax:

```
val list = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
list match { case head :: tail => ... }
```

Custom Extractors

- Remember? We wanted to use the following extractor syntax:

```
val list = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
list match { case head :: tail => ... }
```

- We can do this by defining a custom extractor!

Pattern Matching with `unapply`

- Remember? We wanted to use the following extractor syntax:

```
list match { case head :: tail => ... }
```

- We need to define an `unapply` method

```
object :: {
  def unapply[A](cons: Cons[A]): Option[(A, List[A])] =
    Some((cons.head, cons.tail))
}
```

A Scalable Language

- Scala grows with the demands of its users
  - E.g. by defining your own DSLs

A Scalable Language

- Scala grows with the demands of its users
  - E.g. by defining your own DSLs
  - Scala supports a huge number of **advanced programming constructs**, like:

A Scalable Language

- Scala grows with the demands of its users
  - E.g. by defining your own DSLs
  - Scala supports a huge number of **advanced programming constructs**, like:
    - **Higher-order Functions**

A Scalable Language

- Scala grows with the demands of its users
  - E.g. by defining your own DSLs
  - Scala supports a huge number of **advanced programming constructs**, like:
    - **Higher-order Functions**
    - **Call-by-name** Parameters

A Scalable Language

- Scala grows with the demands of its users
  - E.g. by defining your own DSLs
  - Scala supports a huge number of **advanced programming constructs**, like:
    - **Higher-order Functions**
    - **Call-by-name** Parameters
    - **Lazy Evaluation**

A Scalable Language

- Scala grows with the demands of its users
  - E.g. by defining your own DSLs
  - Scala supports a huge number of **advanced programming constructs**, like:
    - **Higher-order Functions**
    - **Call-by-name** Parameters
    - **Lazy Evaluation**
    - **Implicit** Conversions

A Scalable Language

- Scala grows with the demands of its users
  - E.g. by defining your own DSLs
  - Scala supports a huge number of **advanced programming constructs**, like:
    - **Higher-order Functions**
    - **Call-by-name** Parameters
    - **Lazy Evaluation**
    - **Implicit** Conversions
    - **Actors** for Parallelization

A Scalable Language

- Scala grows with the demands of its users
  - E.g. by defining your own DSLs
  - Scala supports a huge number of **advanced programming constructs**, like:
    - **Higher-order Functions**
    - **Call-by-name** Parameters
    - **Lazy Evaluation**
    - **Implicit** Conversions
    - **Actors** for Parallelization
    - **Structural Subtyping** (aka Ducktyping)

A Scalable Language

- Scala grows with the demands of its users
  - E.g. by defining your own DSLs
  - Scala supports a huge number of **advanced programming constructs**, like:
    - **Higher-order Functions**
    - **Call-by-name** Parameters
    - **Lazy Evaluation**
    - **Implicit** Conversions
    - **Actors** for Parallelization
    - **Structural Subtyping** (aka Ducktyping)
    - **Tailrecursion**

A Scalable Language

- Scala grows with the demands of its users
  - E.g. by defining your own DSLs
  - Scala supports a huge number of **advanced programming constructs**, like:
    - **Higher-order Functions**
    - **Call-by-name** Parameters
    - **Lazy Evaluation**
    - **Implicit** Conversions
    - **Actors** for Parallelization
    - **Structural Subtyping** (aka Ducktyping)
    - **Tailrecursion**
    - **Infix-Notation**

A Scalable Language

- Scala grows with the demands of its users
  - E.g. by defining your own DSLs
  - Scala supports a huge number of **advanced programming constructs**, like:
    - **Higher-order Functions**
    - **Call-by-name** Parameters
    - **Lazy Evaluation**
    - **Implicit** Conversions
    - **Actors** for Parallelization
    - **Structural Subtyping** (aka Ducktyping)
    - **Tailrecursion**
    - **Infix-Notation**
    - Build-in **Dependency Injection**

Scala Enthusiasts Braunschweig

- Since June 16th 2014: `scala-bs.de`

Scala Enthusiasts Braunschweig

- Since June 16th 2014: `scala-bs.de`
- We are **programming enthusiasts**
  - We do **not** only code Scala!
  - *Ruby, Node.JS, Groovy, JavaScript, …*

Scala Enthusiasts Braunschweig

- Since June 16th 2014: `scala-bs.de`
- We are **programming enthusiasts**
  - We do **not** only code Scala!
  - *Ruby, Node.JS, Groovy, JavaScript, …*
- We provide a forum for **speaking and learning about programming**
  - We chose Scala as a common ground


Scala Enthusiasts Braunschweig

- Since June 16th 2014: `scala-bs.de`
- We are **programming enthusiasts**
  - We do **not** only code Scala!
  - *Ruby, Node.JS, Groovy, JavaScript, …*
- We provide a forum for **speaking and learning about programming**
  - We chose Scala as a common ground
- We **meet** every **2nd month**, every **2nd Tuesday** and **talk** about Scala, its libraries, and programming concepts in general
  - Normally we have **two 30 minutes talks** and open discussions afterwards