Functional Programming

3 Functional Data Structures

Functional Data Structures

- are **immutable**
  - once created they cannot be changed

- Based on **algebraic data types**
  - data types in functional programming languages
  - allow pattern matching

- Operations create **new data objects**
  - reusing existing data objects

- Show similar properties as **value types**
  - 2 + 1 creates new value 3
  - Fract(1, 2) + Fract(1,4) creates new value Fract(3, 4)
  - Set(1, 2) + 3 creates new value Set(1, 2, 3)

- Efficient concepts for **persistent collections** exist
  - e.g. persistent lists, hashtables, red-black-trees, finger frees, ...

3 FUNCTIONAL DATA STRUCTURES

Algebraic Data Types and Pattern Matching

- Case classes
- Pattern matching

Basic ADTs

- Tuples
- Option

Functional Collections

- Functional list case study
- Scala's immutable collection library

Algebraic Data Types (ADTs)

- Fundamental concept for defining data types in FP
- Consist of
  - **tagged records**: data tag and field types
  - **variants** of records

also called **tagged unions**

In Haskell

data type definitions:

```
data Point  =  Pt Int Int
```

```
data Shape =   Rect Point Int Int
             | Circle Point Int
```

```
data Day  =  Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

value construction:

```
Pt 10 20
```

```
Rect (Pt 10 10) 20 40
Circle (Pt 30 10) 30
```

```
Mon
Tue
...
```

- Properties
  - immutable
  - data tags identifies value variants
  - variants are closed, i.e., no more variants of a type possible
  - no subtyping

Scala's Case Classes

Case classes implement ADTs by classes and inheritance

- **case class** with **class parameters** for **tagged records**
  - **class name** is **tag**
  - **class parameters** are **fields**

```
case class Point(x: Int, y: Int)
```

value construction:

```
Point(10, 20)
```

- **abstract base type** with **case classes as subtypes** for **variants**
  - keyword **sealed** for closing type shape (only those defined in same file are allowed)

```
sealed trait Shape
case class Rect(pos: Point, w: Int, h: Int) extends Shape
case class Circle(pos: Point, radius: Int) extends Shape
```

```
Rect(Point(10, 10), 20, 40)
Circle(Point(30, 30), 10)
```

- Properties
  - class parameters are public **val** (final)
  - **equal** und **hashCode** based on class parameters
  - no subtypes of case classes

Case Classes

Case classes are special classes

- **class parameters** are **public final** fields
- **equal** and **hashCode** defined based on class parameters
- **toString** based on class parameters

```
sealed trait Expr
case class Var(name: String) extends Expr
case class Lit(value: Double) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr) extends Expr
```

<u>Haskell:</u>

```
data Expr =
      Var String
    | Lit Double
    | BinOp String Expr Expr
```

- instantiation

```
val x = Var("x")
```

```
val expr = BinOp("*", BinOp("+", Var("x"), Var("y")), Lit(2))
```

$(x+y) * 2$

- access to class parameters

```
println ( x.name )
```

```
val left  = expr.left
val right = expr.right
```

- allow **pattern matching**

Java: Java's Record Classes

Java's record classes analogous to Scala's case classes

- class definitions with components

components

```
public record Point(int x, int y) { }
```

allowed variants

```
public sealed interface Shape permits  Shape.Rect, Shape.Circle  {
    record Rect(Point pos, int w, int h) implements Shape {}
    record Circle(Point pos, int r) implements Shape {}
}
```

- value creation

```
Point point = new Point(10, 10);
```

```
Shape shape;
shape = new Shape.Rect(new Point(10, 10), 20, 40);
shape = new Shape.Circle(new Point(20, 20), 10);
```

- access functions for components

```
System.out.format("Point = %d/%d", point.x(), point.y());
System.out.format("Shape position = %d/%d", shape.pos().x(), shape.pos().y());
```

- **equals**, **hashCode**, **toString** implementations based on record components

Scala's Enums

Enum classes

- enum classes are types which allow cases

```
enum Day :
  case Mon, Tue, Wed, Thu, Fri, Sat, Sun
```

```
val day = Day.Fri;
```

- enum classes are a short variant for defining ADTs

```
enum Shape :
  case Rect(pos: Point, w: Int, h: Int) extends Shape
  case Circle(pos: Point, radius: Int) extends Shape
```

```
val shape = Shape.Circle(Point(20, 20), 10)
```

inner classes of Shape

extends Shape optional

3 Functional Data Structures

Algebraic Data Types and Pattern Matching

- Case classes
- Pattern matching

Basic ADTs

- Tuples
- Option

Functional Collections

- Functional list case study
- Scala's immutable collection library

Pattern Matching

Pattern matching analogous to Haskell

- Syntax
  - keyword **match**
  - keyword **case** with patterns

data element

cases

expressions

```
x match {
  case 1   => 1
  case n   => n * fact(n - 1)
}
```

pattens

Haskell:

```
case x of
       1 ->   1
       n ->   n * fact (n - 1)
```

Pattern Matching

Pattern matching works by

- checking for type
- testing pattern literals for equality with (immutable) values of class parameters
- binding pattern variables to values of class parameters

match expression = branching based on patterns

```
def area(shape: Shape) : Double =
  shape match {
    case Circle(pos, r)   => r * r * Math.PI
    case Rect(pos, w, h)  => w * h
  }
```

```
def isInOrigin(shape: Shape) : Boolean =
  shape match {
    case Circle(Point(0, 0), _)    => true
    case Rect(Point(0, 0), _, _)   => true
    case _                         => false
  }
```

don't care (always matches)

Patterns are built by

- **class name**
- pattern variables for fields
- or subpatterns for matching field values

Pattern Matching

Example: Matching Expr

```scala
abstract sealed class Expr
case class Var(name: String) extends Expr
case class Lit(value: Double) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr) extends Expr
```

```scala
expr match {
  case Var(n)                  => println("Variable: " ++ n)
  case Lit(1.0)                => println("Value one")
  case Lit(x)                  => println("Value = " + x)
  case BinOp("*", l, r)        => println("Addition")
  case BinOp(op, Lit(0.0), r)  => println("A binary operation with zero")
}
```

Patterns can be arbitrarily nested !

Pattern Matching

Example: Symbolic differentiation

```scala
abstract sealed class Expr
case class Var(name: String) extends Expr
case class Lit(value: Double) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr) extends Expr
```

```scala
object Expr :
  def deriv(expr : Expr, dx : Var) : Expr =
    expr match {
      case Var(n) if dx.name == n => Lit(1.0)

      case Var(_)              => Lit(0.0)

      case Lit(_)              => Lit(0.0)

      case BinOp("+", u, v)  => BinOp("+", deriv(u, dx), deriv(v, dx))

      case BinOp("*", u, v)  => BinOp("+",
                                  BinOp("*", u, deriv(v, dx)),
                                  BinOp("*", v , deriv(u, dx))
                                )
    }
```

$$\frac{dx}{dx} = 1$$

$$\frac{dc}{dx} = 0$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(u*v)}{dx} = u*\frac{dv}{dx} + v*\frac{du}{dx}$$

Patterns Summary

- Values

```
case 1
case "Hans"
```

- Variables

```
case x
```

- Default

```
case _
```

- Typetests

```
case p : Person
```

- Guards

```
case (x, y) if x == y
```

- Case classes

```
case Var(n)
case Some(x)
case None
```

- Lists

```
case List(1, 2, 3, _*)
```

- Additional variable bindings with @

```
case add0@BinOp("+", zero@Lit(0), r)
```

- ... *some more patterns* ...

Java's Pattern Matching Expressions

Similar to pattern matching in Scala

- with type patterns

type patterns

```
public double area(Shape shape) {
    return switch (shape) {
                case Shape.Rect   rect   -> rect.w() * rect.h();
                case Shape.Circle cicle  -> circle.r() * circle.r() * Math.PI;
            };
}
```

- with record patterns

```
public static double area2(Shape shape) {
  return switch (shape) {
    case Shape.Rect(Point p, int w, int h)  -> w * h;
    case Shape.Circle(Point p, int r)  -> r * r * Math.PI;
  };
}
```

record patterns
➔ must type pattern variables

- but no patterns with literals

3 Functional Data Structures

Algebraic Data Types and Pattern Matching

- Case classes
- Pattern matching

Basic ADTs

- Tuples
- Option

Functional Collections

- Functional list case study
- Scala's immutable collection library

Tuples

Generic tuple data types with multiple elements

- generic types

```
(A, B)
```
```
(A, B, C)
```
... up to 22 elements

short form for types

co-variant type parameters

```
Tuple2[+A, +B]
```
```
Tuple3[+A , +B , +C]
```

- values

```
val personInfo = ("Franz", "Kafka", 1883, "male")
```
```
: (String, String, Int, String)
```

- access operations: _1, _2, ...

```
val first = personInfo._1
val born = personInfo._3
```

- pattern matching
  - in assignments

```
val (first2, last, born2, sex) = personInfo
```

  - in match-expressions

```
personInfo match {
  case ("Franz", "Kafka", year, _) => println(s"Franz Kafka is born in $year")
  case (first, last, year, _)      => println(s"$first $last is born in $year")
}
```

3 Functional Data Structures

Algebraic Data Types and Pattern Matching

- Case classes
- Pattern matching

Basic ADTs

- Tuples
- Option

Functional Collections

- Functional list case study
- Scala's immutable collection library

Option

Analogous to Haskell's **Maybe** type!
Analogous to Java's **Optional** type!

Option[+A] for expressing possibly empty values

- Defined as case classes with two variants **Some** and **None**

```
sealed abstract class Option[+A]
case final class Some[+A](x : A) extends Option[A]
case object None extends Option[Nothing]
```

Some has value **x**
None has no value

- **Option** as return value
  - Example: find for lists

```
val optPrime : Option[Int] = list123.find(x => isPrime(x))
```

- Pattern matching with **Option**

```
optPrime match {
  case Some(p)   => println("The prime found is " + p)
  case None      => println("No prime found")
}
```

JKU

Option

Important operations

see Section Functional exception handling

```
sealed abstract class Option[+A] extends IterableOnce[A] with Product with Serializable {
  final def isEmpty: Boolean = this eq None
  final def isDefined: Boolean = !isEmpty

  def get: A
  final def getOrElse[B >: A](default: => B): B = if (isEmpty) default else this.get
  final def orElse[B >: A](alternative: => Option[B]): Option[B] =if (isEmpty) alternative else this
  def flatMap[B](f: A => Option[B]): Option[B] = if (isEmpty) None else f(this.get)
  def map[B](f: A => B): Option[B] = if (isEmpty) None else Some(f(this.get))
  def filter(p: A => Boolean): Option[A] = if (isEmpty || p(this.get)) this else None
  ...
}
```

```
final case class Some[+A](value: A) extends Option[A] {
  def get: A = value
}
```

```
case object None extends Option[Nothing] {
  def get: Nothing = throw new NoSuchElementException("None.get")
}
```

3 Functional Data Structures

Algebraic Data Types and Pattern Matching

- Case classes
- Pattern matching

Basic ADTs

- Tuples
- Option

Functional Collections

- Functional list case study
- Scala's immutable collection library

Functional Collections

- List, Set, Maps which are immutable

- Operations like adding, removing elements create new data objects
  - operations return new data structures
  - which reuse current data structure

```
val l21 = 2 :: List(1)
```

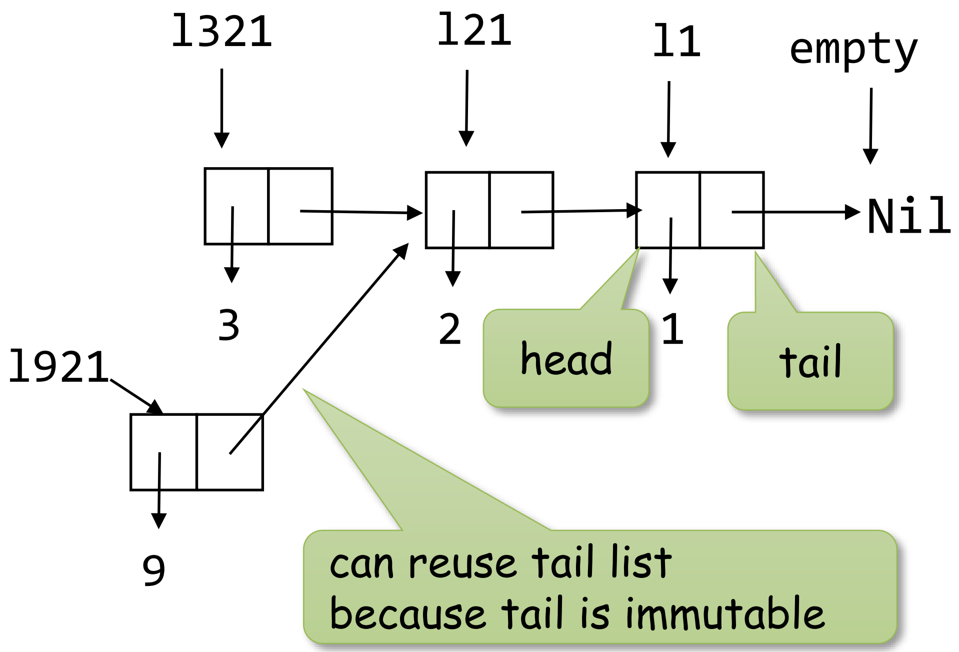
l21 is **l1** plus first element **2**;
l1 is reused

- Efficient concepts for persistent data structures exist
  - e.g. persistent lists, hashtables, red-black-trees, finger trees, ...

  see: Chris Okasaki: Purely Functional Data Structures, Cambridge University Press, 1998

Functional List

- Recursive structure with
  - **head** pointing to **first element**
  - **tail** pointing to **rest list**
  - and **empty list Nil** at end
- can **add** elements **in front**




```
val empty : List[Int] = Nil;
val l1 : List[Int] = empty.prepended(1);
val l21 : List[Int] = l1.prepended(2);
val l321 : List[Int] = l21.prepended(3);

val l921 : List[Int] = l21.prepended(9);
```

Implementation of Functional List in Scala

- sealed trait **FList** and
- case classe **FNil** and **FCons**
- with covariant type parameter **A**

prefix **F** for distinguishing them from standard List type

```
sealed trait FList[+A]

case object FNil extends FList[Nothing]

case class FCons[+A](head: A, tail: FList[A]) extends FList[A]
```

```
val empty = FNil
val l1 = FCons(1, empty)
val l21 = FCons(2, l1)
val l321 = FCons(3, l21)
```

Implementation of Functional List in Scala

- Basis functions **isEmpty** und **size**
  - as dynamically bound values

```
sealed trait FList[+A] :
  val isEmpty : Boolean
  val size : Int

case object FNil extends FList[Nothing] :
  val isEmpty : Boolean = true
  val size = 0

case class FCons[+A](head: A, tail: FList[A]) extends FList[A] :
  val isEmpty : Boolean = false
  val size = tail.size + 1
```

Implementation of Functional List in Scala

- Higher-order functions using pattern matching

```scala
sealed trait FList[+A] :
  def isEmpty : Boolean
  val size : Int

  def find(pred : A => Boolean) : Option[A] =
    this match {
      case FNil => None
      case FCons(hd, tl) => if (pred(hd)) then Some(hd) else tl.find(pred)
    }

  def filter(pred : A => Boolean) : FList[A] =
    this match {
      case FNil => FNil
      case FCons(hd, tl) =>
        if (pred(hd)) then FCons(hd, tl.filter(pred))
        else tl.filter(pred)
    }

  ...
```

Implementation of Functional List in Scala

- Higher-order functions using pattern matching

```
sealed trait FList[+A] :
  ...
  def map[B](fn : A => B) : FList[B] =
    this match {
      case FNil => FNil
      case FCons(hd, tl) => FCons(fn(hd), tl.map(fn))
    }

  def all(pred: A => Boolean) : Boolean =
    find(a => !pred(a)).isEmpty

  def any(pred: A => Boolean) : Boolean =
    find(a => pred(a)).isDefined

  ...
```

FUNCTIONAL LIST: EXAMPLE APPLICATION

Backtracking algorithm: N-Queens problem

- imperative solution with resetting state (in Java)

```
// imperative solution !
boolean solveNQueens(boolean[][] board, int col) {
  if (col >= board.length) {
    return true;
  } else {
    for (int i = 0; i < board.length; i++) {
      if (isSafe(board, i, col)) {
        board[i][col] = true;
        if (solveNQueens(board, col + 1))
          return true;
        board[i][col] = false;  // Backtracking
      }
    }
    return false;
  }
}
```

must reset state change
to previous state

Functional List: Example Application

Backtracking algorithm: N-Queens problem

- functional solution: no resetting of state required

```
def solve(board: List[Pos], col: Int) : Option[List[Pos]] = {
  if (col >= N) {
    return Some(board)
  } else {
    for (i <- 0 to N) {
      if (isSafe(board, i, col)) {
        val optSol = solve(board.prepended(Pos(i, col)), col + 1)
        if (optSol.isDefined) return optSol
      }
    }
    return None
  }
}
```

Expanding solution does not change current solution!

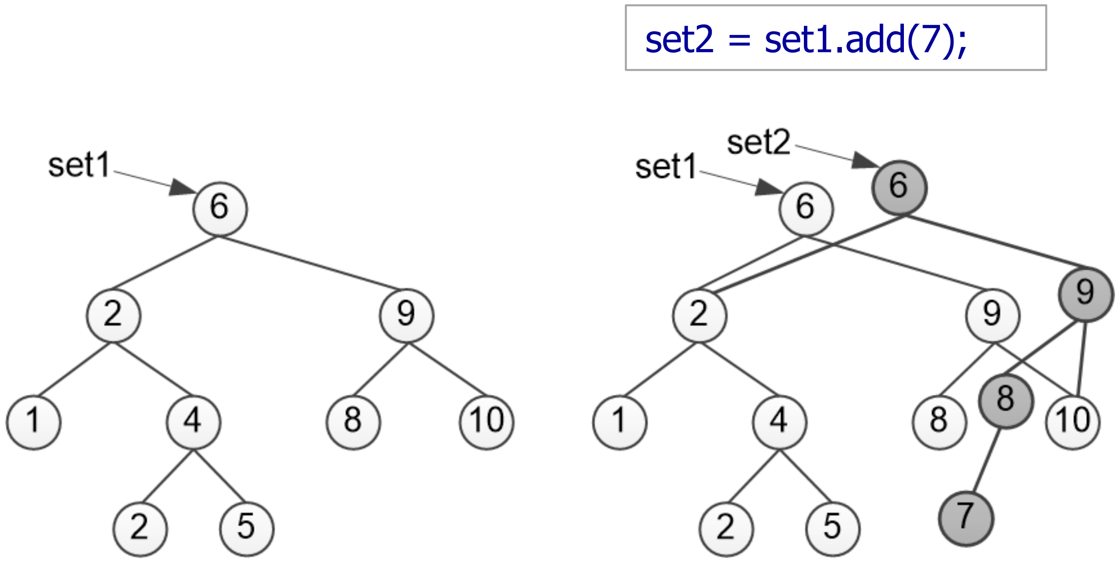
no backtracking change needed

Persistent Data Structures

Principal approach

- minimal copying, maximal reuse

Example: add element in binary tree



Pros and Cons of Persitent Data Structures

Pros

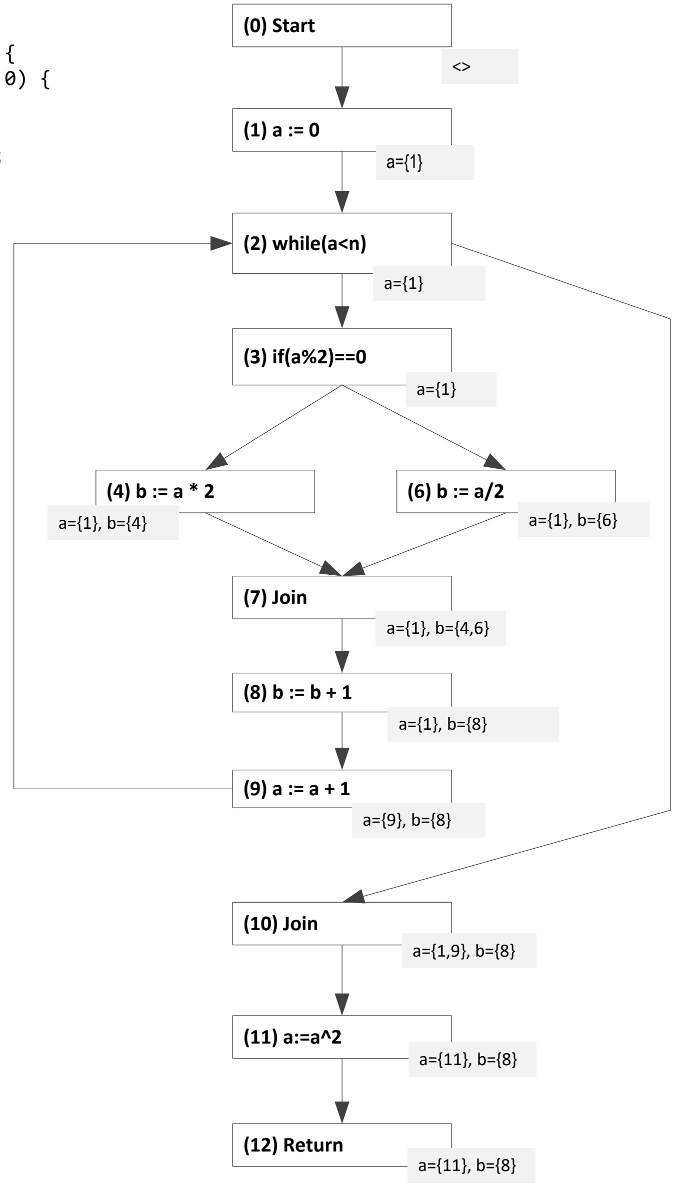
- easy to use
  - no need for copying
- reliable
  - no side effects
- thread-safe
  - very helpful in concurrent and parallel programs

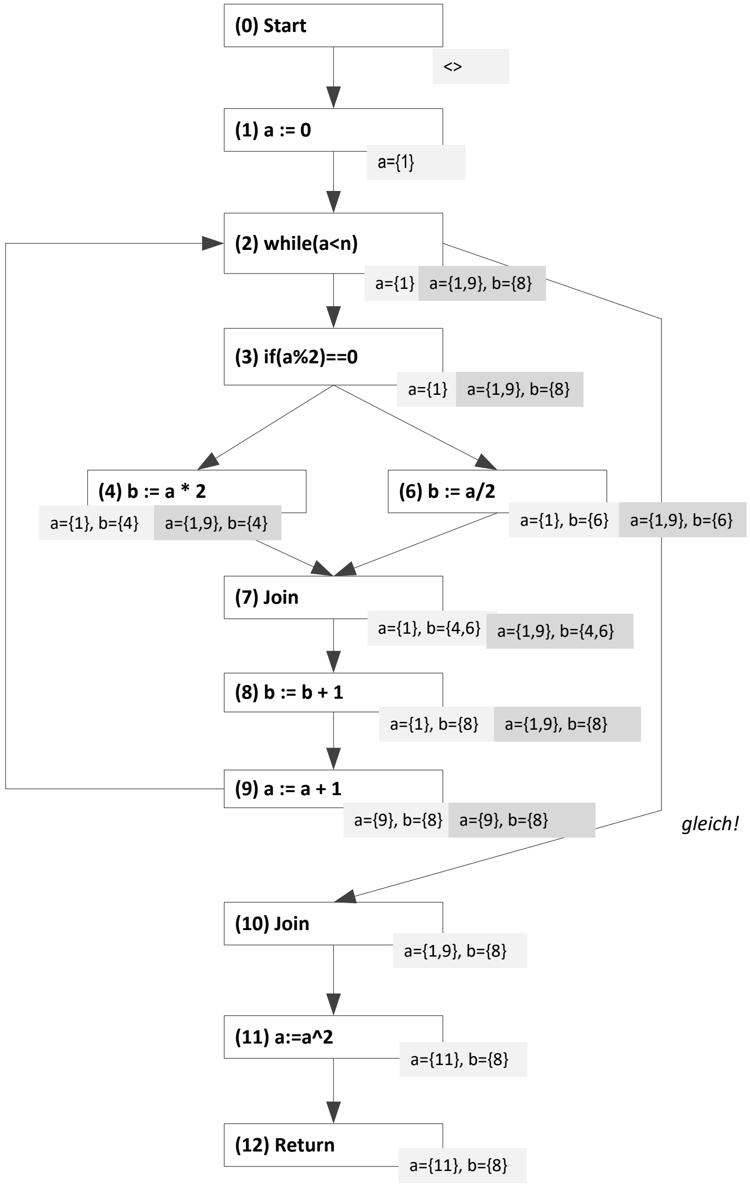
Cons

- some overhead in memory and run time

EXAMPLE APPLICATION: ASSIGNMENTS

```
 1: a := 0;
 2: while (a < n) {
 3:  if (a % 2 == 0) {
 4:    b := a * 2
 5:  } else {
 6:    b := a * 3;
 7:  }
 8:  b := b + 1;
 9:  a := a + 1;
10: }
11: a:=a*a;
```





3 Functional Data Structures

Algebraic Data Types and Pattern Matching

- Case classes
- Pattern matching

Basic ADTs

- Tuples
- Option

Functional Collections

- Functional list case study
- Scala's immutable collection library

Introduction to Immutable Collections

Scala provides a powerful library of immutable collections

- Package **scala.collections.immutable**

Abstract types

Iterable[T]	basic trait for collections
Seq[T]	defined sequence of elements
IndexedSeq[T]	direct acces
LinearSeq[T]	linear iteration
Set[T]	sets
SortedSet[T]	sorted sets
Map[K, V]	maps
SortedMap[K, V]	map sorted key K

Collection Implementation Classes

Concrete Implementations of immutable collections

Iterable[T]	
Seq[T]	
IndexedSeq[T]	
`Vector[T]`	Finger tree
`Range`	Range of Ints
LinearSeq[T]	
`List[T]`	List with Nil and Cons
`Stream[T]`	List with lazy access
`Stack[T]`	Stack
`Queue[T]`	Queue
Set[T]	
`HashSet[T]`	Hash-Table
`LinkedHashSet`	Hash-Table + LinkedList
SortedSet[T]	
`TreeSet[T]`	Red-Black-Tree
Map[K, V] extends *Iterable*[(K, V)]	
`HashMap[K, V]`	Hash-Table
SortedMap[K, V]	
`TreeMap[K, V]`	Red-Black-Tree

List

Generic list data type List[T]

- with two variants
  - ☐ empty list

```
Nil
```

  - ☐ cons operator

```
first :: rest
```

- construction with `::`

```
val list123 : List[Int] = 1 :: 2 :: 3 :: Nil
```

- construction with List constructors

```
val list123 = List(1, 2, 3)
```

```
val empty : List[Int] = List()
```

List() = Nil

- access operations: head, tail ...

```
val first = list123.head
val rest = list123.tail
```

Pattern Matching with Lists

Pattern matching with lists

- in assignments
  - with `::` patterns

```
val (first :: rest2) = list123
```

matches lists with at least 1 element

  - with **List** patterns

```
val List(first, second, third) = list123
```

matches lists with exactly 3 elements

```
val List(first, second, _*) = list123
```

matches lists with at least 2 elements

- in match expressions

binding rest to xs

```
list123 match {
  case List(1, 2, xs @ _*)  => println("first elements are 1, 2, rest is" + xs)
  case (1 :: xs)            => println("first element is 1")
  case List()               => println("empty list")
  case _                    => println("something else")
}
```

Pattern Matching with Lists

Example: equalLists

```scala
def equalLists[A](xs : List[A], ys : List[A]) : Boolean =
  (xs, ys) match {
    case (List(), List())               => true
    case (_     , List())               => false
    case (List(), _)                    => false
    case (x::xs , y::ys)  if x == y     => equalLists(xs, ys)
    case (x::xs , y::ys)                => false
  }
```

Haskell:

```haskell
equalLists :: Eq a => [a] -> [a] -> Bool
equalLists []     []               = True
equalLists _      []               = False
equalLists []     _                = False
equalLists (x:xs) (y:ys) | x == y = equalLists xs ys
equalLists (x:xs) (y:ys)           = False
```

COLLECTION: PACKAGES UND IMPORTS

Packages

- `scala.collections` base types
- `scala.collections.immutable` immutable collections
- `scala.collections.mutable` mutable collections
- `scala.collections.generic` internal

also mutable collections: e.g. ListBuffer

Implicit imports (mit `scala` und `Predef`)

- Those types are implicitly imported and always available unqualified
  - `List` → `scala.collection.immutable.List`
  - `Vector` → `scala.collection.immutable.Vector`
  - `Set` → `scala.collection.immutable.Set`
  - `Map` → `scala.collection.immutable.Map`
  - `Iterable` → `scala.collection.Iterable`
  - `Seq` → `scala.collection.Seq`
  - `IndexedSeq` → `scala.collection.IndexedSeq`
  - …

Standard Implementations

- The following types are created by default for trait types

```
Iterable(4, 5, 7)   → Implementation : scala.collection.immutable.List
Seq(3, 9, 2)        → Implementation : scala.collection.immutable.List
IndexedSeq(3, 9, 2) → Implementation : scala.collection.immutable.Vector
Set(1, 2, 3)        → Implementation : scala.collection.immutable.HashSet
Map(1 -> "eins")    → Implementation : scala.collection.immutable.HashMap
…
```

- Examples:

```
// without imports

val trav = Iterable[String]()                    scala.collection.immutable.List[String]
val iter = Seq("A", "B")                         scala.collection.immutable.List[String]
val iseq = IndexedSeq("F", "H", "P", "U")        scala.collection.immutable.Vector[String]
val set = Set("A", "B")                          scala.collection.immutable.HashSet[String]
val sortedSet = SortedSet("A", "B")              scala.collection.immutable.TreeSet[String]
val map = Map(1 -> "A", 2 -> "B")                scala.collection.immutable.HashMap[Int, String]
val sortedMap = SortedMap(1 -> "A", 2 -> "B")    scala.collection.immutable.TreeMap[Int, String]
```

ITERABLE[T]: IMPORTANT METHODS

Concatenation	`++ (xs2: Trav..)`	`val coll = Iterable(1, 2) ++ Iterable(3, 4)`
Test if empty	`isEmpty`	`val empty = coll.isEmpty`
Siez	`size`	`val n = coll.size`
first element	`head`	`val first = coll.head`
rest list	`tail`	`val rest = coll.tail`
last element	`last`	`val last = coll.last`
alle but last	`init`	`val firstElems = coll.init`
first n element	`take(n : T)`	`val firstNElems = coll take 3`
element after n elements	`drop(n : T)`	`val lastElems = coll drop 2`
String with separator	`mkString(sep)`	`val str = coll.mkString(", ")`
Iteration	`foreach(f: T => U)`	`coll foreach (x => println(x))`
map elements	`map(f: T => U)`	`val squares = coll.map(x => x * x)`
All quantifier	`forall(pred:..)`	`val allEven = coll.forall(x => x % 2 == 0)`
exists quantifier	`exists(pred:..)`	`val existsOdd = coll.exists(x => x % 2 == 0)`
Filtering elements	`filter(pred:..)`	`val evenElems = coll.filter(x => x % 2 == 0)`
Number with property	`count(pred:..)`	`val nEven = coll.count (x => x % 2 == 0)`
Partitioning elements	`partition(pred:..)`	`val (even, odds) = coll partition (x => x % 2 == 0)`
	`...`	`...`

Higher-order Functions in Iterable

```scala
val numbers = Iterable(1, 2, 3, 4, 5, 6, 7)
```

- Examples
  - map

    ```scala
    val squared = numbers.map((x) => x * x)
    ```

  - filter

    ```scala
    val evenNumbers = numbers.filter((x) => x % 2 == 0)
    ```

  - exists

    ```scala
    val existsNegative : Boolean = numbers.exists((x) => x < 0)
    ```

  - forall

    ```scala
    val allPositive : Boolean = numbers.forall((x) => x >= 0)
    ```

  - foreach

    ```scala
    numbers.foreach((x) => println(x))
    ```

  - partition

    ```scala
    val (smaller5, greaterEqual5) = numbers.partition((x) => x < 5)
    ```

  - reduceLeft

    ```scala
    val sum = numbers.reduceLeft((x, y) => x + y)
    ```

Iterable[T]: More Methods

- Building pairs
- last n elements
- elements without last n
- ...

```
zip[B](xs: Iterable[B])
takeRight(n: Int)
dropRight(n: Int)
...
```

```
val itrble = Iterable(...)
val pairs = itrble zip (1 to itrble.size)
val last2 = itrble takeRight 2
val exceptLast2 = itrble dropRight 2
...
```

Seq[T]: Important Methods

		`var seq = coll.toSeq`
acces by	`apply(i: Int)`	`val x = seq(i)`
Test if index defined	`isDefinedAt(i: Int)`	`if (seq isDefinedAt i)`
Index set	`indices`	`val indices = seq.indices`
Index of element	`indexOf(x: T)`	`val idx = seq indexOf x`
add in front	`+:(x : T)`	`seq = x +: seq`
add at end	`:+(x : T)`	`seq = seq :+ x`
add until len reached	`padTo(len: Int, x: T)`	`seq = seq.padTo(10, 0)`
change values	`updated(i: Int, x: T)`	`seq = seq.updated(i, y)`
containment	`contains(x : Any)`	`val containedX = seq.contains (x)`
reverse	`reverse()`	`val rev = seq.reverse`
intersection	`intersect(ys: Seq[T])`	`val intersection = seq intersect other`
…	…	

opeations create new sequences

Set[T]: Important Methods

```
var set = Set(1, 2, 3)
```

add element	`+ (x: T)`	`set = set + 4`
add some elements	`+ (x: T*)`	`set = set + (5, 6)`
add elements	`++(x: Iterable[T])`	`set = set ++ Iterable(7, 8)`
remove an element	`- (x : T)`	`set = set - 2`
remove elements	`- (x : T*)`	`set = set – (3, 5)`
	`--(x: Iterable[T])`	`set = set -- Iterable(7, 8)`
intersection set	`& (s: Set[T])`	`set = Set(1,2, 3) & Set(2, 3, 4)`
union set	`| (s: Set[T])`	`set = Set(1,2, 3) | Set(2, 3, 4)`
set difference	`&~ (s: Set[T])`	`set = Set(1,2, 3) &~ Set(2, 3, 4)`
containment	`contains(x: T)`	`val xContained = set contains x`
subset test	`subsetOf(set: Set[T])`	`val isSubset = set subsetOf Set(1,2,3,4,5)`

Map[T]: Important Methods

```
var map = Map(1 -> "A", 2 -> "B")
```

access by key	`get(k: K): Option[V]`	`val vOpt : Option[String] = map.get(2)`
access by key with default	`getOrElse(k: K, d: V): V`	`val v = map.getOrElse(2, "")`
access (exception if not contained)	`apply(k: K) : V`	`val v = map(2)`
test if containd	`contains(k : K) : Boolean`	`if (map.contains(2)) { …`
keys	`keys : Iterable[K]`	`for (k <- map.keys) { …`
values	`values : Iterable[V]`	`for (v <- map.values) { …`
add key value pair	`+ (k: K, v: V)`	`map = map + (3 -> "C")`
remove entry with key	`- (k : K)`	`map = map - 3`
…	…	

for Loop

- for loop for iteration over collections
  - Iteration over collection

```
for (variable <- collection) { code }
```

  - Iteration over collection with elements collected and returned

```
val results = for (variable <- collection) yield { result }
```

```
val filesInDir = java.io.File(".").listFiles

for (file <- filesInDir) {
  println(file.toString)
}

val fileNames: Seq[String] = for (f <- filesInDir) yield f.getName()
```

FOR LOOP

- with filtering of elements

```
for (variable <- collection if condition ) { code }
```

```
for (variable <- collection if condition1; if condition2 ) { code }
```

separated by ;

```
for (  file <- filesInDir
       if file.isFile;
       if file.getName.endsWith(".scala")
) println(file.toString)
```

- nested iteration

```
def fileLines(file: java.io.File) = {
  scala.io.Source.fromFile(file).getLines().toList
}
for (
  file <- filesInDir
  if file.isFile;
  line <- fileLines(file)
  if line.trim.matches(pattern)
) println(file.getName + ": " + line.trim)
```

Iteration over files

Iteration over lines of files

for Loop

- Iteration over number ranges
  - inclusive upper limit

```
for (i <- from to to) { code }
```

  - exclusive upper limit

```
for (i <- from until to) { code }
```

Examples:

```
for (i <- 0 to 10 ) { println(i + " " ) }
```

```
0 1 2 3 4 5 6 7 8 9 10
```

```
for (i <- 1 until 10) { println(i) }
```

```
0 1 2 3 4 5 6 7 8 9
```

Remark: **`to`** und **`until`** are methods for Int providing **Range** collections

```
val range1_10 = 1.until(10)
```

```
for (i <- range1_10) {
  println(i)
}
```

Infix-Notation:

```
val range1_10 = 1 until 10
```