Comp 311
Functional Programming

Nick Vrvilo, Two Sigma Investments
Robert “Corky” Cartwright, Rice University

September 19, 2018

Scala Style Guide

Scala has an official style guide that you should reference while working on your homework projects:

https://docs.scala-lang.org/style/

Family Trees

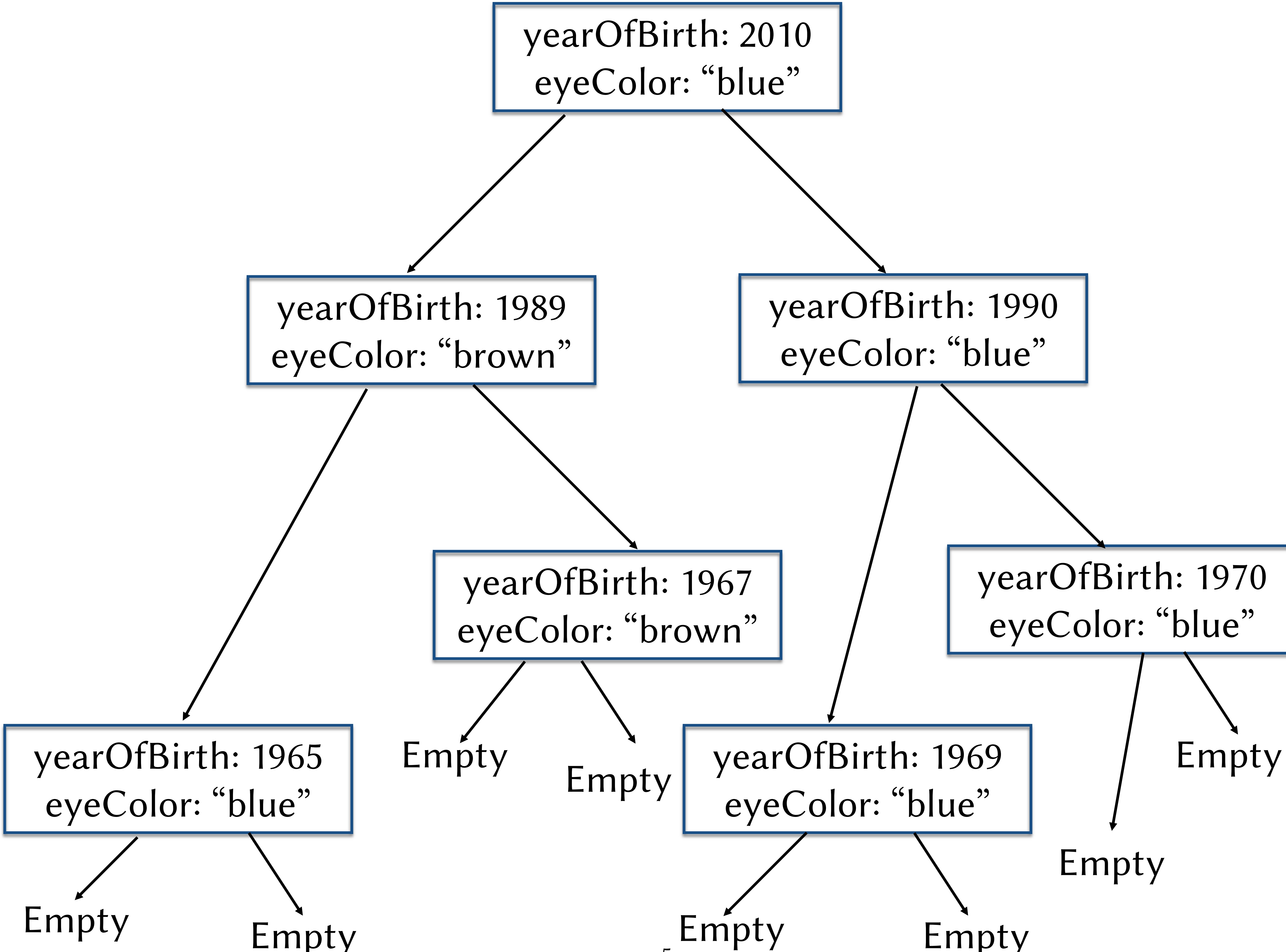
```
TreeNode ::= Empty
           | Child(TreeNode,
                   TreeNode,
                   Int,
                   String)
```

Family Trees

```
sealed abstract class TreeNode

case object EmptyNode extends TreeNode

case class Child(
    mother: TreeNode,
    father: TreeNode,
    yearOfBirth: Int,
    eyeColor: String)
  extends TreeNode
```


yearOfBirth: 2010
eyeColor: “blue”
yearOfBirth: 1989
eyeColor: “brown”
yearOfBirth: 1990
eyeColor: “blue”
yearOfBirth: 1967
eyeColor: “brown”
yearOfBirth: 1970
eyeColor: “blue”
yearOfBirth: 1965
eyeColor: “blue”
Empty
Empty
yearOfBirth: 1969
eyeColor: “blue”
Empty
Empty
Empty
Empty
Empty
Empty
Empty

Family Trees

```
def hasBlueEyedAncestor(t: TreeNode): Boolean =
  t match {
    case EmptyNode => false
    case Child(mother,father,_,eyeColor) =>
      ( (eyeColor == "Blue")
      || hasBlueEyedAncestor(mother)
      || hasBlueEyedAncestor(father) )
  }
```

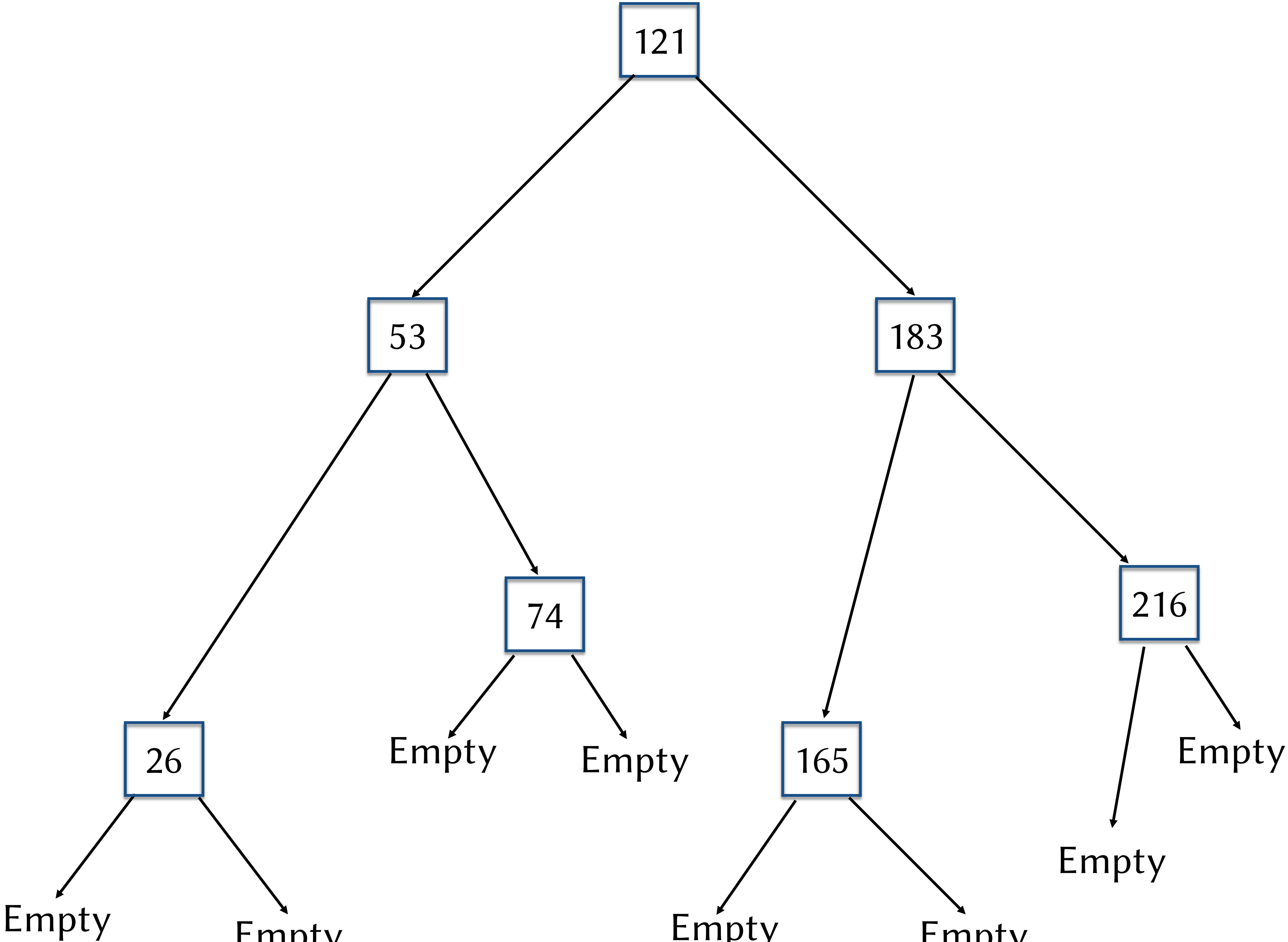
Family Trees

```
def hasBlueEyedAncestor(t: TreeNode): Boolean =
  t match {
    case EmptyNode => false
    case Child(_,_,_,"Blue") => true
    case Child(mother,father,_,_) =>
      hasBlueEyedAncestor(mother) ||
      hasBlueEyedAncestor(father)
  }
```

Binary Search Trees

Binary Search Trees

- We define trees containing only Ints
- To help us find elements quickly, we abide by the following invariant:
  - At a given node containing value *n*:
    - All values in the left subtree are less than *n*
    - All values in the right subtree are greater than *n*


121
53
183
74
216
26
Empty
Empty
165
Empty
Empty
Empty
Empty
Empty
Empty
Empty

Binary Search Trees

```
abstract class BinarySearchTree {
  def contains(n: Int): Boolean
  def insert(n: Int): BinarySearchTree
}
```

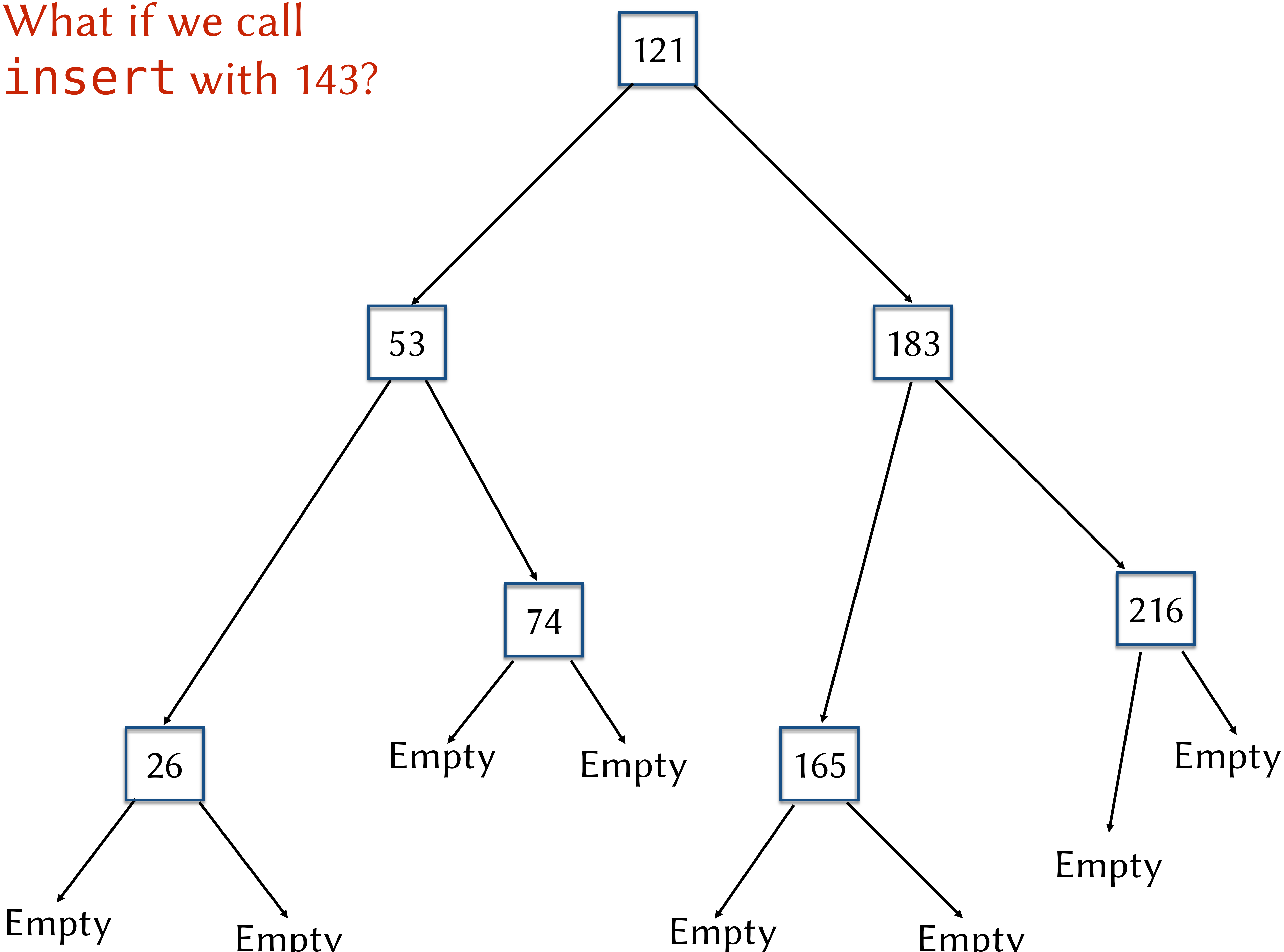
Binary Search Trees

```
case object EmptyTree extends BinarySearchTree {
  def contains(n: Int) = false
  def insert(n: Int) = ConsTree(n, EmptyTree, EmptyTree)
}
```

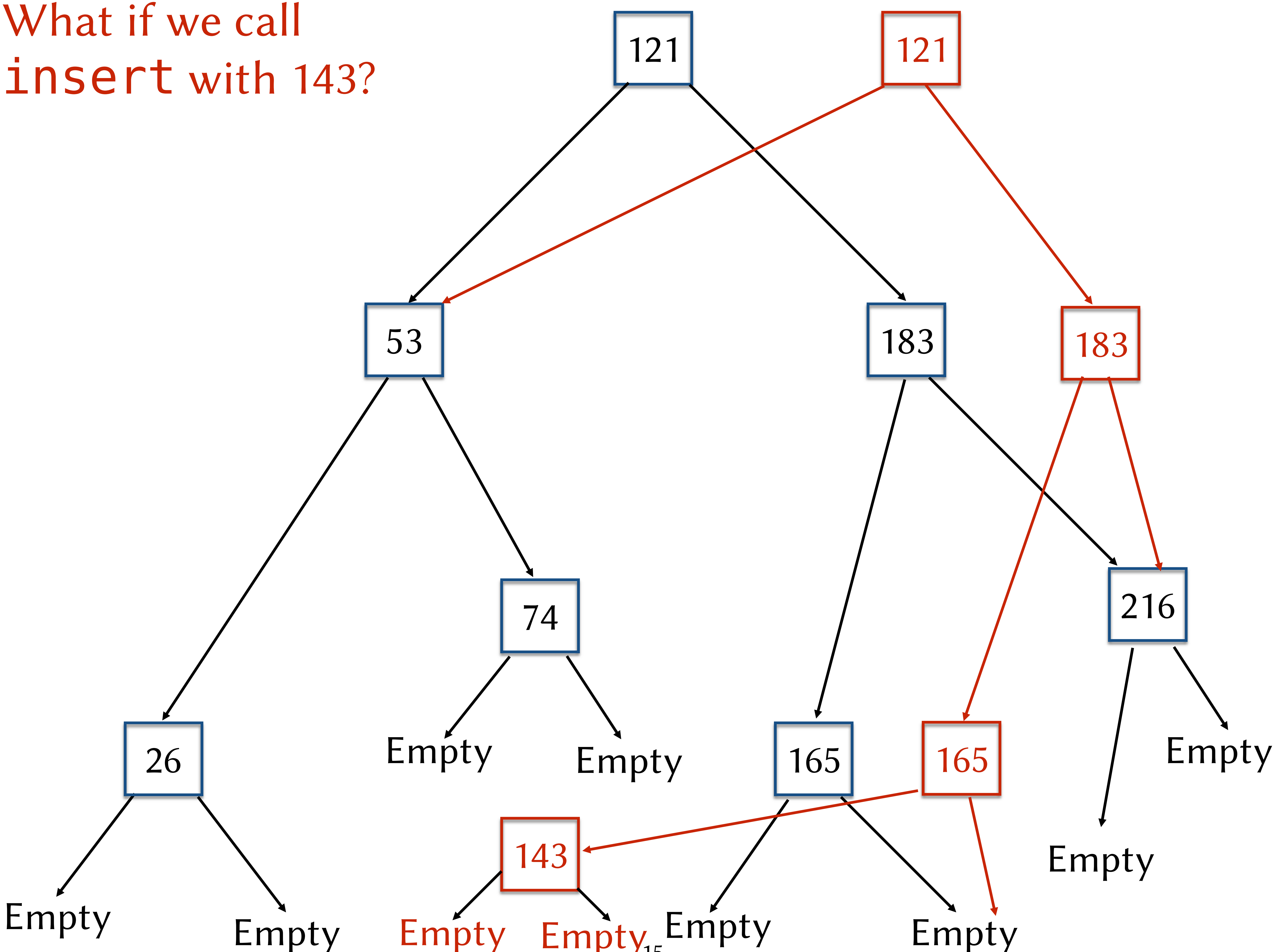
Binary Search Trees

```
case class ConsTree(
    m: Int,
    left: BinarySearchTree,
    right: BinarySearchTree)
  extends BinarySearchTree {
  def contains(n: Int): Boolean = {
    if (n < m) left.contains(n)
    else if (n > m) right.contains(n)
    else true // n == m
  }
  def insert(n: Int) = {
    if (n < m) ConsTree(m, left.insert(n), right)
    else if (n > m) ConsTree(m, left, right.insert(n))
    else this // n == m
  }
}
```

What if we call `insert` with 143?



What if we call `insert` with 143?



Traversing Multiple Recursive Parameters

Taking the First Few Elements

```
def take(n: Int, xs: List): List = {
  require(0 <= n && n <= xs.size)
  (n, xs) match {
    case (0, xs) => Empty
    case (n, Cons(y, ys)) => Cons(y, take(n-1, ys))
  }
}
```

Dropping the First Few Elements

```
def drop(n: Int, xs: List): List = {
  require(0 <= n && n <= xs.size)
  (n, xs) match {
    case (0, xs) => xs
    case (n, Cons(y, ys)) => drop(n-1, ys)
  }
}
```

Functional Update of a List

```
def update(xs: List, i: Int, y: Int): List = {
  require(0 <= i && i < xs.size)
  assume(xs != Empty) // implied by requirements

  (xs, i) match {
    case (Cons(z, zs), 0) => Cons(y, zs)
    case (Cons(z, zs), _) => Cons(z, update(zs,i-1,y))
  }
}
```

Design Abstraction

Our Function Templates Reveal Common Structure

```
def containsZero(xs: List): Boolean = xs match {
  case Empty => false
  case Cons(n, ys) => (n == 0) || containsZero(ys)
}

def containsOne(xs: List): Boolean = xs match {
  case Empty => false
  case Cons(n, ys) => (n == 1) || containsOne(ys)
}
```

Our Function Templates Reveal Common Structure

```
def contains(m: Int, xs: List): Boolean = xs match {
  case Empty => false
  case Cons(n, ys) => (n == m) || contains(m, ys)
}
```

But Sometimes the Part We Want to Abstract Is a Function

```
def below(m: Int, xs: List): List =
  xs match {
    case Empty => Empty
    case Cons(n, ys) => {
      if (n < m) Cons(n, below(m, ys))
      else below(m, ys)
    }
  }
```

But Sometimes the Part We Want to Abstract Is a Function

```
def above(m: Int, xs: List): List =
  xs match {
    case Empty => Empty
    case Cons(n, ys) => {
      if (n > m) Cons(n, above(m, ys))
      else above(m, ys)
    }
  }
```

Taking Functions As Parameters

```
def filter(f: (Int)=>Boolean, xs: List): List =
  xs match {
    case Empty => Empty
    case Cons(n, ys) => {
      if (f(n)) Cons(n, filter(f, ys))
      else filter(f, ys)
    }
  }
```

Passing Functions as Arguments

```
val xs = Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty))))))

filter(((n: Int) => (n > 0)), xs) ↦*
Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty))))))

filter(((n: Int) => (n < 0)), xs) ↦*
Empty

filter(((n: Int) => (n < 3)), xs) ↦*
Cons(1,Cons(2,Empty))
```

Passing Functions as Arguments

```
val xs = Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty))))))

filter(((n: Int) => (n > 0)), xs) ↦*
Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty))))))

filter(((n: Int) => (n < 0)), xs) ↦*
Empty

filter(((n: Int) => (n < 3)), xs) ↦*
Cons(1,Cons(2,Empty))
```

These are *function literals*

First-Class Functions

- Function literals are expressions with static arrow types that reduce to *function values*
- The value type of a function value is also an arrow type
- Function values are first-class values:
  - They are allowed to be passed as arguments
  - They are allowed to be returned as results

Simplifying Function Literals

Parameter types on function literals are allowed to be elided whenever the types are clear from context:

```
filter(((n: Int) => (n > 0)), xs)
```

can be written as

```
filter(((n) => (n > 0)), xs)
```

Simplifying Function Literals

- Parentheses around a single parameter is allowed to be omitted

```
filter(((n) => (n > 0)), xs)
```

can be written as

```
filter(n => (n > 0), xs)
```

Simplifying Function Literals

- When a single parameter is used only once in the body of a function literal:
  - We can drop the parameter list
  - We simply write the body with an _ at the place where the parameter is used

For example,

```
((x: Int) => (x < 0))
```

becomes

```
_ < 0
```

Passing Function Literals As Arguments

```
val xs = Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty))))))

filter(_ < 3, xs) ↦* Cons(1,Cons(2,Empty))
```

Guidelines On Using Function Literals

- Function literals are well-suited to situations in which:
  - The function is only used once
  - The function is not recursive
  - The function does not constitute a key concept in the problem domain

Comprehensions

$$\{2x \mid x \in xs\}$$

Mapping a Computation Over a List

```
def double(xs: List) = xs match {
  case Empty => Empty
  case Cons(y,ys) => Cons(y+y, double(ys))
}
```

Mapping a Computation Over a List

```
def negate(xs: List) = xs match {
  case Empty => Empty
  case Cons(y,ys) => Cons(-y, negate(ys))
}
```

Negation as a Comprehension

$$\{-x \mid x \in xs\}$$

Generalizing a Mapping Computation

```
def map(f: Int=>Int, xs: List): List =
  xs match {
    case Empty => Empty
    case Cons(y,ys) => Cons(f(y), map(f,ys))
  }
```

Mapping a Computation Over a List

```
val xs = Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty))))))

negate(xs) ↦*
Cons(-1,Cons(-2,Cons(-3,Cons(-4,Cons(-5,Cons(-6,Empty))))))

double(xs) ↦*
Cons(1,Cons(4,Cons(9,Cons(16,Cons(25,Cons(36,Empty))))))
```

Mapping a Computation Over a List

```
val xs = Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty))))))

map(-_, xs) ↦*
Cons(-1,Cons(-2,Cons(-3,Cons(-4,Cons(-5,Cons(-6,Empty))))))

map(x => x+x, xs) ↦*
Cons(1,Cons(4,Cons(6,Cons(8,Cons(10,Cons(12,Empty))))))
```

Recall Our Sum Function Over Lists

```
def sum(xs: List): Int = xs match {
  case Empty => 0
  case Cons(y,ys) => y + sum(ys)
}
```

In Mathematics, We Might Write this as a Summation

$$\sum_{x \in xs} x$$

And Our Product Function Over Lists

```
def product(xs: List): Int = xs match {
  case Empty => 1
  case Cons(y,ys) => y * product(ys)
}
```

In Mathematics, We Might Write this as a Product

$$\prod_{x \in xs} x$$

We Abstract to a Reduction Function Over Lists

```
def reduce(base: Int, f: (Int, Int) => Int, xs: List): Int =
  xs match {
    case Empty => base
    case Cons(y,ys) => f(y, reduce(base, f, ys))
  }
```

Example Reductions

```
val xs = Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty))))))

reduce(0, (x,y) => x + y, xs) ↦* 21

reduce(1, (x,y) => x * y, xs) ↦* 720
```

Min and Max

```
def max(xs: List): Int =
  reduce(Int.MinValue, (x,y) => if (x > y) x else y, xs)

def min(xs: List): Int =
  reduce(Int.MaxValue, (x,y) => if (x < y) x else y, xs)
```

Min and Max

Numbers in Scala have min/max binary operators:

```
def max(xs: List): Int =
  reduce(Int.MinValue, (x,y) => x max y, xs)

def min(xs: List): Int =
  reduce(Int.MaxValue, (x,y) => x min y, xs)
```

Min and Max, Simplified

```
def max(xs: List) = reduce(Int.MinValue, _ max _, xs)

def min(xs: List) = reduce(Int.MaxValue, _ min _, xs)
```

Simplifying Function Literals

- When *each* parameter is used only once in the body of a function literal, and in the order in which they are passed:
  - We can drop the parameter list
  - We simply write the body with an _ at the place where each parameter is used

For example,

```
((x: Int, y: Int) => (x + y))
```

becomes

```
_ + _
```

Example Reductions

```
val xs = Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty))))))

reduce(0, _+_, xs) ↦* 21

reduce(1, _*_, xs) ↦* 720
```

Note the multiple parameters

Min and Max, Simplified

```
def max(xs: List) = reduce(Int.MinValue, _ max _, xs)

def min(xs: List) = reduce(Int.MaxValue, _ min _, xs)
```

Combinations of Maps and Reductions

$$\sum_{x \in xs} x^2 + 1$$

Combinations of Maps and Reductions

```
reduce(0, _+_, map(x => x*x + 1, xs))
```

Summation

```
def summation(xs: List, f: Int => Int) =
    reduce(0, _+_, map(f, xs))
```

Summation

```
def square(x: Int) = x * x

summation(xs, square(_)+1)
```