Comp 311
Functional Programming

Eric Allen, Two Sigma Investments
Robert “Corky” Cartwright, Rice University
Sagnak Tasirlar, Two Sigma Investments
Family Trees

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

case object EmptyNode extends TreeNode

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

def hasBlueEyedAncestor(t: TreeNode): Boolean = {
  t match {
    case EmptyNode => false
    case Child(m,f,b,e) => ((e == "Blue") ||
                           hasBlueEyedAncestor(m) ||
                           hasBlueEyedAncestor(f))
  }
}
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 \( n \)
Binary Search Trees

abstract class BinarySearchTree {
    def contains(n: Int): Boolean
    def insert(n: Int): BinarySearchTree
}
case object EmptyTree extends BinarySearchTree {
  def contains(n: Int) = false
  def insert(n: Int) = ConsTree(n, EmptyTree, EmptyTree)
}
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 \texttt{insert} with 143?
Traversing Multiple Recursive Datatypes
Taking the First Few Elements

def take(n: Nat, xs: List): List = {
    // require n <= size(xs)
    (n, xs) match {
        case (Zero, xs) => Empty
        case (Next(m), Cons(y, ys)) => Cons(y, take(m, ys))
    }
Taking the First Few Elements

def take(n: Int, xs: List): List = {
  require ((n >= 0) && (n <= size(xs)))
  (n,xs) match {
    case (0, xs) => Empty
    case (n, Cons(y, ys)) => Cons(y, take(n-1, ys))
  }
}
Dropping the First Few Elements

```scala
def drop(n: Int, xs: List): List = {
  require ((n >= 0) && (n <= size(xs)))
  (n, xs) match {
    case (0, xs) => xs
    case (n, Cons(y, ys)) => drop(n-1, ys)
  }
}
```
def update(xs: List, i: Nat, y: Int): List = {
  require (xs != Empty) // && i < size(xs)

  (xs, i) match {
    case (Cons(z, zs), Zero) => Cons(y, zs)
    case (Cons(z, zs), Next(j)) => Cons(z, update(zs, j, y))
  }
}
def update(xs: List, i: Int, y: Int): List = {
  require ((i >= 0) && (i < size(xs)))
  assert (xs != Empty)

  (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

```scala
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

```scala
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

\[
\text{filter}(((n: \text{Int}) \Rightarrow (n > 0)), \text{xs})
\]

  can be written as

\[
\text{filter}(((n) \Rightarrow (n > 0)), \text{xs})
\]
Simplifying Function Literals

- Parentheses around a single parameter is allowed to be omitted

  \[
  \text{filter}(((n) => (n > 0)), xs) \]

  can be written as

  \[
  \text{filter}(n => (n > 0), xs) \]
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,

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

becomes

```plaintext
_ < 0
```
Passing Function Literals As Arguments

```scala
val xs = Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty))))))
filter(_ < 3, xs) => Cons(1,Cons(2,Empty))
```