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)
Family Trees

abstract class TreeNode

case object EmptyNode extends TreeNode

case class Child(mother: TreeNode, father: TreeNode, yearOfBirth: Int, eyeColor: String) extends TreeNode
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)
}
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 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))
  }
}
Functional Update of a List

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
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

```scala
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

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

can be written as

```scala
filter(((n) => (n > 0)), xs)
```
Simplifying Function Literals

- Parentheses around a single parameter is allowed to be omitted

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

can be written as

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

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

becomes

```typescript
_ < 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))
```
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\}
def double(xs: List) = {
    xs match {
        case Empty => Empty
        case Cons(y, ys) => Cons(2 * y, double(ys))
    }
}
def negate(xs: List) = {
  xs match {
    case Empty => Empty
    case Cons(y, ys) => (-y, negate(ys))
  }
}
Negation as a Comprehension

\[ \{ -x \mid x \in xs \} \]
def map(f: Int => Int, xs: List) = {
  xs match {
    case Empty => Empty
    case Cons(y,ys) => Cons(f(y), map(f,ys))
  }
}
Mapping a Computation Over a List

```scala
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

\[
\begin{align*}
\text{val } \ xs &= \text{Cons}(1,\text{Cons}(2,\text{Cons}(3,\text{Cons}(4,\text{Cons}(5,\text{Cons}(6,\text{Empty})))))) \\
\text{map}(\_ , \ xs) \mapsto \ast \\
\text{Cons}(-1,\text{Cons}(-2,\text{Cons}(-3,\text{Cons}(-4,\text{Cons}(-5,\text{Cons}(-6,\text{Empty}))))) \\
\text{map}(x \mapsto 2 * x, \ xs) \mapsto \ast \\
\text{Cons}(1,\text{Cons}(4,\text{Cons}(9,\text{Cons}(16,\text{Cons}(25,\text{Cons}(36,\text{Empty})))))
\end{align*}
\]
Recall Our Sum Function Over Lists

```python
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

```scala
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) = {
    reduce(Int.MinValue, (x,y) => if (x > y) x else y, xs)
}

def min(xs: List) = {
    reduce(Int.MaxValue, (x,y) => if (x < y) x else y, 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
Combinations of Maps and Reductions

\[ \sum_{x \in x s} x^2 + 1 \]
Combinations of Maps and Reductions

\[
\text{reduce}(0, \_+\_, \text{map}(x \Rightarrow x \times x + 1, \text{xs}))
\]
def summation(xs: List, f: Int => Int) = 
    reduce(0, _+_ , map(f, xs))
Summation

def square(x: Int) = x * x

summation(xs, square(_)+1)
More Syntactic Sugar

• Functions defined with `def` can be passed as arguments whenever an expression of a compatible function type is expected

• What constitutes a compatible function type?
Partially Applied Functions

- If we want to pass a function as an argument, but supply some of the arguments to the function ourselves, we can wrap an application to the function in a function literal:

```javascript
map(x => x + 1, xs)
```
Partially Applied Functions

- If we want to pass a function as an argument, but supply some of the arguments to the function ourselves, we can wrap an application to the function in a function literal:

  \[ \text{map}(x \mapsto x + 1, \text{xs}) \]

  which is equivalent to

  \[ \text{map}(_ + 1, \text{xs}) \]
Partially Applied Functions

- **Eta Expansion:** Wrapping a function in function literal that takes all of the arguments of \( f \) and immediately calls \( f \) with those arguments

\[
(x:\text{Int}) \Rightarrow \text{square}(x)
\]

is equivalent to

\[
\text{square}
\]
Mapping a Computation Over a List

We can use eta expansion to pass operators as arguments:

\[
\text{map}(x \mapsto -x, \text{xs})
\]
Mapping a Computation Over a List

We can use eta expansion to pass operators as arguments:

map(_, xs)
Returning Functions as Values
We Can Define Functions That Return Other Functions as Values

```scala
def adder(x: Int): Int => Int = {
  def addX(y: Int) = x + y
  addX
}
```
def adder(x: Int): Int => Int = {
    def addX(y: Int) = x + y
    addX
}

The explicit return type is needed because Scala type inference assumes an unapplied function is an error.
We Can Define Functions That Return Other Functions as Values

```scala
def adder(x: Int) = {
  def addX(y: Int) = x + y
  addX _
}
```

Alternatively, we can eta-expand `addX` to assure the type checker that we really do intend to return a function
We Can Define Functions That Return Other Functions as Values

```scala
def adder(x: Int) = {
  def addX(y: Int) = x + y
  addX _
}
```

An underscore outside of parentheses in a function application denotes the entire tuple of arguments passed to the function.
We Can Define Functions That Return Other Functions as Values

```python
def adder(x: Int) = x + ((_: Int)
```

We can instead define add by *partially* eta-expanding the + operator. But then we need to annotate the second operand with a type.
Imports
Importing a Member of a Package

import scala.collection.immutable.List
Importing Multiple Members of a Package

import scala.collection.immutable.{List, Vector}
Importing and Renaming Members of a Package

import scala.collection.immutable.{List=>SList, Vector}
Importing All Members of a Package

import scala.collection.immutable._

Note that * is a valid identifier in Scala!
Combining Notations

import scala.collection.immutable.{_}

same meaning as:

import scala.collection.immutable._
Combining Notations

import scala.collection.immutable.{List=>SList,_}

Imports all members of the package but renames List to SList
Combining Notations

```scala
import scala.collection.immutable.{List=>_,_}
```

Imports all members of the package except for `List`
Importing a Package

```scala
import scala.collection.immutable

Now sub-packages can be denoted by shorter names:

immutable.List
```
Importing and Renaming Packages

```scala
import scala.collection.{immutable => I}

I.List
```

Allows members to be written like this:

`I.List`
Importing Members of An Object

import Arithmetic._

Allows members such as `Arithmetic.gcd` to be write like this:

```scala
gcd
```
Implicit Imports

The following imports are implicitly included in your program:

```scala
import java.lang._
import scala._
import Predef._
```
Package java.lang

- Contains all the standard Java classes
- This import allows you to write things like:
  
  Thread

  instead of:

  java.lang.Thread
Package scala

- Provides access to the standard Scala classes:
  
  BigInt, BigDecimal, List, etc.
Object Predef

- Definitions of many commonly used types and methods, such as:

  require, ensuring, assert
### Visibility Modifier Private

For a method `Arithmetic.reduce` in package `Rationals`

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>no modifier</td>
<td>public access</td>
</tr>
<tr>
<td>private</td>
<td>private to class <code>Arithmetic</code></td>
</tr>
</tbody>
</table>
Local Definitions

- As with constant definitions, we can make function definitions local to the body of a function.
- The functions can be referred to only in the body of the enclosing function.
def reduce() = {
    val isPositive =
        ((numerator > 0) & (denominator > 0)) |
        ((numerator < 0) & (denominator < 0))

def reduceFromInts(num: Int, denom: Int) = {
    require ((num >= 0) & (denom > 0))
    val gcd = Arithmetic.gcd(num, denom)
    val newNum = num/gcd
    val newDenom = denom/gcd

    if (isPositive) Rational(newNum, newDenom)
    else Rational(-newNum, newDenom)
}
reduceFromInts(Arithmetic.abs(numerator), Arithmetic.abs(denominator))

} ensuring (_ match {
  case Rational(n,d) => Arithmetic.gcd(n,d) == 1 & (d > 0)
})
Announcements

• Homework 2 Available from Piazza (Due October 1)

• Two Sigma Info Session at Huff House, 4pm Today