Comp 311
Functional Programming

Eric Allen, PhD
Vice President, Engineering
Two Sigma Investments, LLC
Announcements

- Homework 2 Available from Piazza (Due October 1)
- Two Sigma Info Session at Huff House, 4pm Today
Traversing Multiple Recursive Datatypates
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 <= 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
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

```scala
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)
        }
    }
}
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}(\(((n) \Rightarrow (n > 0)), \ xs)\)
\]

can be written as

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

\[((x: \text{Int}) => (x < 0))\]

becomes

\[_ < 0\]