Comp 311
Functional Programming

Eric Allen, Two Sigma Investments
Robert “Corky” Cartwright, Rice University
Sağnak Taşırlar, Two Sigma Investments
Options

- Often the result of a computation is that no satisfactory value could be found
  - Lookup in a table with a key that does not exist
  - Attempting to find a path that does not exist
abstract class Option[+A] {...}

case object None extends Option[Nothing] {...}

case class Some[+A](val contained: A) extends Option[A]
{
    ...
}
}
Options Are Monads!

```scala
abstract class Option[+A] {
  def flatMap[B](f: (A) ⇒ Option[B]): Option[B]
  def map[B](f: (A) ⇒ B): Option[B]
  def withFilter(p: (A) ⇒ Boolean):
    FilterMonadic[A, collection.Iterable[A]]
}
```
/**
 * Create a path from start to finish in G, if it exists.
 */

def findRoute(start: String, end: String, graph: Graph):
    Option[List[String]]
Reduce to Backtracking Cases

```scala
def findRoute(start: String, end: String, graph: Graph): Option[List[String]] = {
  if (start == end) Some(List(end))
  else for (route <- routeFromOrigins(graph(start), end, graph))
    yield start :: route
}
```
Recursive Sub-Problems

def routeFromOrigins(origins: List[String], destination: String, graph: Graph): Option[List[String]] = {
    origins match {
        case Nil => None
        case origin :: origins => {
            findRoute(origin, destination, graph) match {
                case None => routeFromOrigins(origins, destination, graph)
                case Some(route) => Some(route)
            }
        }
    }
}
Termination

- `routeFromOrigins` is structurally recursive:
  - terminates provided that `findRoute` terminates
- `findRoute` terminates only if graph is acyclic
Accumulating Knowledge
Accumulating Knowledge

• Remember visited nodes to prevent infinite regress

• Pass this to recursive calls via “accumulator”
def findRoute(start: String, end: String, graph: Graph, visited: List[String] = Nil):
    Option[List[String]] = {
        if (start == end) Some(List(end))
        else if (visited contains start) None
        else for (route <- routeFromOrigins(graph(start), end, graph, start :: visited))
            yield start :: route
    }
def routeFromOrigins(origins: List[String], destination: String, graph: Graph, visited: List[String] = Nil):
    Option[List[String]] = {
        origins match {
            case Nil => None
            case origin :: origins => {
                findRoute(origin, destination, graph, visited) match {
                    case None => routeFromOrigins(origins, destination, graph, origin :: visited)
                    case Some(route) => Some(route)
                }
            }
        }
    }
Accumulators

• accumulator parameter allows us to “remember” knowledge from one recursive call to another

• Often essential for correctness in generative recursion

• Also useful for saving space in structural recursion
Accumulators for Structural Recursion

• Let us define a function `fromOrigin`, which:
  
  • Takes a list of `Int` values, with each value denoting a relative distance to the point to its left
  
  • Returns a list of `Int` values denoting the absolute distances to the origin
Accumulators for Structural Recursion

becomes

2  5  10  12  20
Defining fromOrigin

def fromOrigin[T](xs: List[T]) = {
    xs match {
        case Nil => Nil
        case x :: xs => x :: (fromOrigin {xs} map {_+x})
    }
}
Defining fromOrigin

def fromOrigin (xs: List[Int]): List[Int] = {
  xs match {
    case Nil => Nil
    case x :: xs =>
      x :: (for (y <- fromOrigin(xs)) yield {y+x})
  }
}
cost of fromOrigin

fromOrigin(List(2,3,5,2,8)) ↦
  List(2,3,5,2,8) match {
    case Empty => Empty
    case x :: xs => x :: (fromOrigin {xs} map {_+x})
  } ↦
2 :: (fromOrigin(List(3,5,2,8)) map (_+2)) ↦*
2 :: (3 :: (fromOrigin(List(5,2,8) map (_+3))) map(_+2)) ↦*
2 :: (3 :: (List(5,7,15) map (_+3))) map(_+2)) ↦*
2 :: (3 :: (List(8,10,18)) map(_+2)) ↦*
2 :: (List(5,10,12,20)) ↦*
List(2, 5, 10, 12, 20)
The cost of fromOrigin

• Each recursive call map over the argument list
  • which takes $n$ steps for a list of length $n$

\[\sum_{i=1}^{n} i = \frac{(n)(1 + n)}{2} = O(n^2)\]
Big O Notation

• We say:

\[ f(x) = O(g(x)) \text{ as } x \to \infty \]

• To mean that there is a constant \( k \) and some value \( x_0 \) such that

\[ |f(x)| \leq k|g(x)| \text{ for all } x \geq x_0 \]
Big O Notation

• Typically the part:

\[ \text{as } x \to \infty \]

• is implicit

• Effectively, we are defining equivalence classes of functions
Accumulating Distance to the Origin

• We could reduce the time taken by instead accumulating the distance to the origin in a parameter
Accumulating Distance to the Origin

def fromOriginAcc(xs: List[Int]) = {
  def inner(xs: List[Int], fromOrigin: Int): List[Int] = {
    xs match {
      case Nil => Nil
      case x :: xs => {
        val xToOrigin = x + fromOrigin
        xToOrigin :: inner(xs, xToOrigin)
      }
    }
  }
  inner(xs, 0)
}
Guidelines for Using Accumulators in Functions

- Start with the standard design recipes!
- Add an accumulator *only after* the initial design attempt
Guidelines for Using Accumulators in Functions

- Recognize the benefit of having an accumulator
- Understand what the accumulator denotes
Recognizing the Benefit of an Accumulator

- If the function is structurally recursive and uses an auxiliary function, consider an accumulator

- Study hand evaluations to see if an accumulator helps in reducing time or space costs
Recognizing the Benefit of an Accumulator

```scala
def invert[T](xs: List[T]): List[T] = {
  xs match {
    case Nil => Nil
    case x :: xs => makeLastItem(x, invert(xs))
  }
}

def makeLastItem[T](x: T, xs: List[T]): List[T] = {
  xs match {
    case Nil => List(x)
    case y :: ys => y :: makeLastItem(x, ys)
  }
}
```
Recognizing the Benefit of an Accumulator

- there is nothing for invert to forget

- consider accumulating the items walked over
Recognizing the Benefit of an Accumulator

```scala
def invert[T](xs: List[T]): List[T] = {
  def inner(xs: List[T], accumulator: List[T]): List[T] = {
    xs match {
      case Nil => ...
      case y :: ys => ... inner(… ys … y … accumulator …)
    }
  }
  inner(xs, Nil)
}
```
Recognizing the Benefit of an Accumulator

- accumulator must stand for a list
- it could stand for all elements that precede \( xs \)
Recognizing the Benefit of an Accumulator

def invert[T](xs: List[T]): List[T] = {
    def inner(xs: List[T], accumulator: List[T]): List[T] = {
        xs match {
            case Nil => …
            case y :: ys => … inner(… ys … y :: accumulator)
        }
    }
    inner(xs, Nil)
}
Recognizing the Benefit of an Accumulator

• Now it is clear that the accumulator contains all the elements that precede xs in reverse order
def invert[T](xs: List[T]): List[T] = {
  def inner(xs: List[T], accumulator: List[T]): List[T] = {
    xs match {
      case Nil => accumulator
      case y :: ys => inner(ys, y :: accumulator)
    }
  }
  inner(xs, Nil)
}
Recognizing the Benefit of an Accumulator

• The key step in the design process is to establish the invariant that describes the relationship between the accumulator and the parameters of a function

• Establish appropriate accumulator invariant is an art that takes practice
Recognizing the Benefit of an Accumulator

```scala
def sum1(xs: List[Int]): Int = {
  xs match {
    case Nil => 0
    case y :: ys => y + sum1(ys)
  }
}
```
An Accumulator for Sum

- walking over elements of a list to return their sum
- obvious thing to accumulate is the sum so far
def sum2(xs: List[Int]): Int = {
  def inner(xs: List[Int], accumulator: Int): Int = {
    xs match {
      case Nil => accumulator
      case y :: ys => inner(ys, y + accumulator)
    }
  }
  inner(xs, 0)
}
An Accumulator for Sum

\[
\text{sum1(List}(5, 3, 7, 9)) \rightarrow *
\text{5} + \text{sum1(List}(3, 7, 9)) \rightarrow *
\text{5} + 3 + \text{sum1(List}(7, 9)) \rightarrow *
\text{5} + 3 + 7 + \text{sum1(List}(9)) \rightarrow *
\text{5} + 3 + 7 + 9 + \text{sum1(List}()) \rightarrow *
\text{5} + 3 + 7 + 9 + 0 \rightarrow 
\text{8} + 7 + 9 + 0 \rightarrow 
\text{15} + 9 + 0 \rightarrow 
\text{24} + 0 \rightarrow 
\text{24}
\]
An Accumulator for Sum

\[
\begin{align*}
\text{sum2}(\text{List}(5, 3, 7, 9)) & \mapsto * \\
\text{inner}(\text{List}(5, 3, 7, 9), 0) & \mapsto * \\
\text{inner}(\text{List}(3, 7, 9), 5 + 0) & \mapsto * \\
\text{inner}(\text{List}(3, 7, 9), 5) & \mapsto * \\
\text{inner}(\text{List}(7, 9), 5 + 3) & \mapsto * \\
\text{inner}(\text{List}(7, 9), 8) & \mapsto * \\
\text{inner}(\text{List}(9), 7 + 8) & \mapsto * \\
\text{inner}(\text{List}(9), 15) & \mapsto * \\
\text{inner}(\text{List}(), 9 + 15) & \mapsto * \\
\text{inner}(\text{List}(), 24) & \mapsto * \\
\end{align*}
\]

24
An Accumulator for Sum

• The key advantage of our accumulator version of sum is space

• The advantage is not a matter as to whether the space is used on the stack or in the heap as an argument!

• The ability to reduce the sum as we recur is the primary cause of space savings
This Would Not Save Space

def sum3(xs: List[Int]): Int = {
  def inner(xs: List[Int], accumulator: () => Int): Int = {
    xs match {
      case Nil => accumulator()
      case y :: ys => inner(ys, () => (y + accumulator()))
    }
  }
  inner(xs, () => 0)
}
Thoughts on Accumulators

- Accumulator-based functions are not always faster
  - Accumulator-based factorial tends to be slower
- Accumulator-based functions do not always take less space
Thoughts on Accumulators

• Accumulator-based functions are usually harder to understand

• Programmers new to functional programming are seduced by them because sometimes they can be similar to loops
Thoughts on Accumulators

• Use accumulators judiciously and understand the benefits you are trying to achieve
abstract class Tree[+T]

case object Empty extends Tree[Nothing]

case class Branch[+T](data: T, left: Tree[T], right: Tree[T]) extends Tree[T]
Accumulators and Trees

def height[T](tree: Tree[T]): Int = {
    tree match {
        case Empty => 0
        case Branch(d,l,r) => max(height(l), height(r)) + 1
    }
}
Accumulators and Trees

- One natural thing to try is to include an accumulator of type `Int`
- This accumulator can maintain the distance we have descended from the root of the tree
def height2[T](tree: Tree[T]): Int = {
    def inner(tree: Tree[T], accumulator: Int): Int = {
        tree match {
            case Empty => accumulator
            case Branch(d,l,r) => max(inner(l, accumulator + 1),
                                    inner(r, accumulator + 1))
        }
    }
    inner(tree, 0)
}
abstract class FamilyTree

case object Empty extends FamilyTree

case class Cons(father: FamilyTree, mother: FamilyTree, name: String, birthYear: Int, eyes: String) extends FamilyTree
Family Trees Revisited

- Let’s develop a method `blueEyedAncestors` that finds *all* blue-eyed ancestors in a tree
def blueEyedAncestors(tree: FamilyTree): List[String] = {
  tree match {
    case Empty => Nil
    case Cons(father, mother, name, _, eyes) => {
      val inParents = blueEyedAncestors(father) ++
                      blueEyedAncestors(mother)

      eyes match {
        case "blue" => name :: inParents
        case _ => inParents
      }
    }
  }
}
Family Trees Revisited

- We have defined a structurally recursive function that relies on an auxiliary recursive function: ++

- As discussed, functions of this form often benefit from the use of an accumulator

- We sketch a template for our accumulator-based function in the usual way
def blueEyedAncestors2(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: ...) = {
    tree match {
      case Empty => {...}
      case Cons(father, mother, name, _, eyes) => {
        val inParents = inner(...father...accumulator...) ...
        inner(...mother...accumulator...)
        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }
  inner(tree...)
}
Formulating an Accumulator Invariant

• Our accumulator should remember knowledge about the family tree lost as we descend the tree

• There are two recursive applications: To the father tree and the mother tree

• Options:
  • Denote all blue-eyed ancestors encountered so far
  • Denote all the trees we still need to look at
Option 1: Denote All Blue-Eyed Ancestors Encountered So Far

```scala
def blueEyedAncestors2(tree: FamilyTree): List[String] = {
def inner(tree: FamilyTree, accumulator: List[String]): List[String] = {
    tree match {
        case Empty => accumulator
        case Cons(father, mother, name, _, eyes) => {
            val inParents = inner(father, inner(mother, accumulator))
            eyes match {
                case "blue" => name :: inParents
                case _ => inParents
            }
        }
    }
}
inner(tree, Nil)
```
Option 1: Denote All Blue-Eyed Ancestors Encountered So Far

```scala
def blueEyedAncestors2(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: List[String]): List[String] = {
    tree match {
      case Empty => accumulator
      case Cons(father, mother, name, _, eyes) => {
        val inParents = inner(father, inner(mother, accumulator))

        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }
  inner(tree, Nil)
}
```

Return type is determined by our choice of accumulator invariant
Option 1: Denote All Blue-Eyed Ancestors Encountered So Far

```scala
def blueEyedAncestors2(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: List[String]): List[String] = {
    tree match {
      case Empty => accumulator
      case Cons(father, mother, name, _, eyes) => {
        val inParents = inner(father, inner(mother, accumulator))
        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }
  inner(tree, Nil)
}
```

We must pass in the result of one descent to the other to maintain the invariant.
def blueEyedAncestors2(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: List[String]): List[String] = {
    tree match {
      case Empty => accumulator
      case Cons(father, mother, name, _, eyes) => {
        val inParents = inner(father, inner(mother, accumulator))

        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }
  inner(tree, Nil)
}

Thus, our combining operator is function composition.
Option 1: Denote All Blue-Eyed Ancestors Encountered So Far

```scala
def blueEyedAncestors2(tree: FamilyTree): List[String] = {
    def inner(tree: FamilyTree, accumulator: List[String]): List[String] = {
        tree match {
            case Empty => accumulator
            case Cons(father, mother, name, _, eyes) => {
                val inParents = inner(father, inner(mother, accumulator))

                eyes match {
                    case "blue" => name :: inParents
                    case _ => inParents
                }
            }
        }
    }
    inner(tree, Nil)
}
```

Our choice of invariant determines what to return in the Empty case.
Option 1: Denote All Blue-Eyed Ancestors Encountered So Far

```scala
def blueEyedAncestors2(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: List[String]): List[String] = {
    tree match {
      case Empty => accumulator
      case Cons(father, mother, name, _, eyes) => {
        val inParents = inner(father, inner(mother, accumulator))
        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }
  inner(tree, Nil)
}
```

Our choice also determines the initial value of the accumulator.
Option 2: Denote All Family Trees Not Yet Processed

def blueEyedAncestors3(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: List[FamilyTree]): List[String] = {
    tree match {
      case Empty => {...}
      case Cons(father, mother, name, _, eyes) => {
        val inParents = inner(father, mother :: accumulator)

        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }

  inner(tree, Nil)
}
def blueEyedAncestors3(tree: FamilyTree): List[String] = {
    def inner(tree: FamilyTree, accumulator: List[FamilyTree]): List[String] = {
        tree match {
            case Empty => {...}
            case Cons(father, mother, name, _, eyes) => {
                val inParents = inner(father, mother :: accumulator)

                eyes match {
                    case "blue" => name :: inParents
                    case _ => inParents
                }
            }
        }
    }
    inner(tree, Nil)
}

Naturally, the only tree to process initially is tree, so our accumulator is Nil.
Option 2: Denote All Family Trees Not Yet Processed

def blueEyedAncestors3(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: List[FamilyTree]):
  List[String] = {
    tree match {
      case Empty => {...}
      case Cons(father, mother, name, _, eyes) => {
        val inParents = inner(father, mother :: accumulator)
        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }
  inner(tree, Nil)
}
Option 2: Denote All Family Trees Not Yet Processed

• When the tree is empty, we choose the next element in our accumulator to recur on
def blueEyedAncestors3(tree: FamilyTree): List[String] = {
    def inner(tree: FamilyTree, accumulator: List[FamilyTree]): List[String] = {
        tree match {
            case Empty => accumulator match {
                case Nil => Nil
                case tree :: trees => inner(tree, trees)
            }
            case Cons(father, mother, name, _, eyes) => {
                val inParents = inner(father, mother :: accumulator)

                eyes match {
                    case "blue" => name :: inParents
                    case _ => inParents
                }
            }
        }
    }
    inner(tree, Nil)
}
Tail Recursion
Tail Recursion

• Some functions defined using accumulators have a special property:

  • The recursive call occurs as the last step in the computation
abstract class Nat {
    def !(): Nat
    def *(m: Nat): Nat
    def +(m: Nat): Nat
}

Note that this is a postfix operator. (This follows from the rules for method application syntax.)
Nats

case object Zero extends Nat {
    def !(()) = Next(Zero)
    def *(m: Nat) = Zero
    def +(m: Nat) = m
}

case class Next(n: Nat) extends Nat {
  def !() = this * (n!)
  def *(m: Nat) = m + (n * m)
  def +(m: Nat) = Next(n + m)
}
Nats

Next(Next(Next(Zero)))! \rightarrow
Next(Next(Next(Zero))) \times \text{Next(Next(Zero))}! \rightarrow
Next(Next(Next(Zero))) \times \text{Next(Next(Zero))} \times \text{Next(Zero)}! \rightarrow
Next(Next(Next(Zero))) \times \text{Next(Next(Zero))} \times \text{Next(Zero)} \times \text{Zero}! \rightarrow
Next(Next(Next(Zero))) \times \text{Next(Next(Zero))} \times \text{Next(Zero)} \times \text{Next(Zero)} \rightarrow
\ldots
\text{Next(Next(Next(Next(Next(Next(Zero))))))}
Pure Recursion

def !() = this * (n!)

def !() = {
    def inner(n: Nat, acc: Nat): Nat = {
        n match {
            case Zero => acc
            case Next(m) => inner(m, n * acc)
        }
    }
    inner(this, Next(Zero))
}
Nats

Next(Next(Next(Zero)))! →
inner(Next(Next(Next(Zero))), Next(Zero)) →
inner(Next(Next(Zero)), Next(Next(Next(Next(Zero)))))) →
inner(Next(Zero), Next(Next(Next(Next(Next(Next(Zero))))))) →
inner(Zero, Next(Next(Next(Next(Next(Next(Next(Zero))))))) →
Next(Next(Next(Next(Next(Next(Next(Zero)))))))
Translating for Ints

def factorial(n: Int): Int = {
    if (n == 0) 1
    else n * factorial(n - 1)
}

def factorial2(n: Int) = {
    def inner(n: Int, acc: Int): Int = {
        if (n == 0) acc
        else inner(n - 1, n * acc)
    }
    inner(n, 1)
}
Pure Recursion with Ints

\[ 3! \rightarrow \]
\[ 3 \times 2! \rightarrow \]
\[ 3 \times 2 \times 1! \rightarrow \]
\[ 3 \times 2 \times 1 \times 0! \rightarrow \]
\[ 3 \times 2 \times 1 \times 1 \rightarrow \]
\[ \ldots \]
\[ 6 \]
Tail Recursion with Ints

3! ↦
inner(3, 1) ↦
inner(2, 3) ↦
inner(1, 6) ↦
inner(0, 6) ↦
6