Comp 311
Functional Programming

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

Syntactic Sugar: Currying

- Scala provides special syntax for defining a function that immediately returns another function:

```
def f(x0:T1,…,xN:TN) = (y0:U1,…,yM:UM) => expr
```

can be written as:

```
def f(x0:T1,…,xN:TN) (y0:U1,…,yM:UM) = expr
```

- Defining a function in this way is called “currying” after the computer scientist Haskell Curry

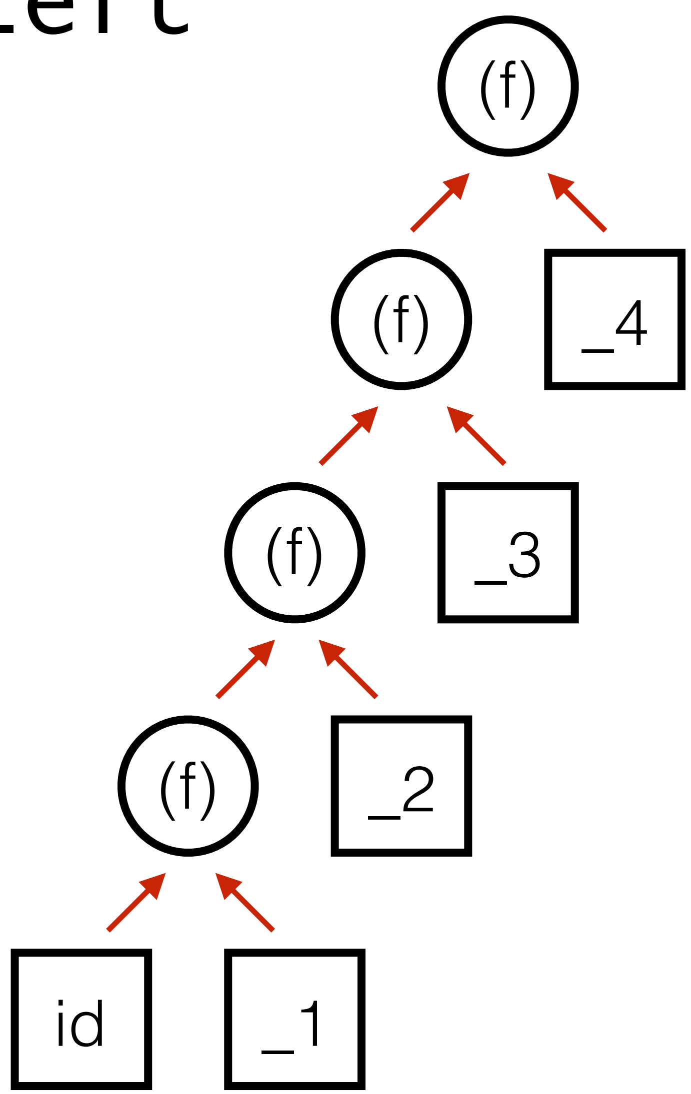
Reduce Revisited

```
abstract class List[+T] {
  …
  def foldLeft[S >: T](x: S)(f: (S, S) => S): S
  def foldRight[S >: T](x: S)(f: (S, S) => S): S
}
```

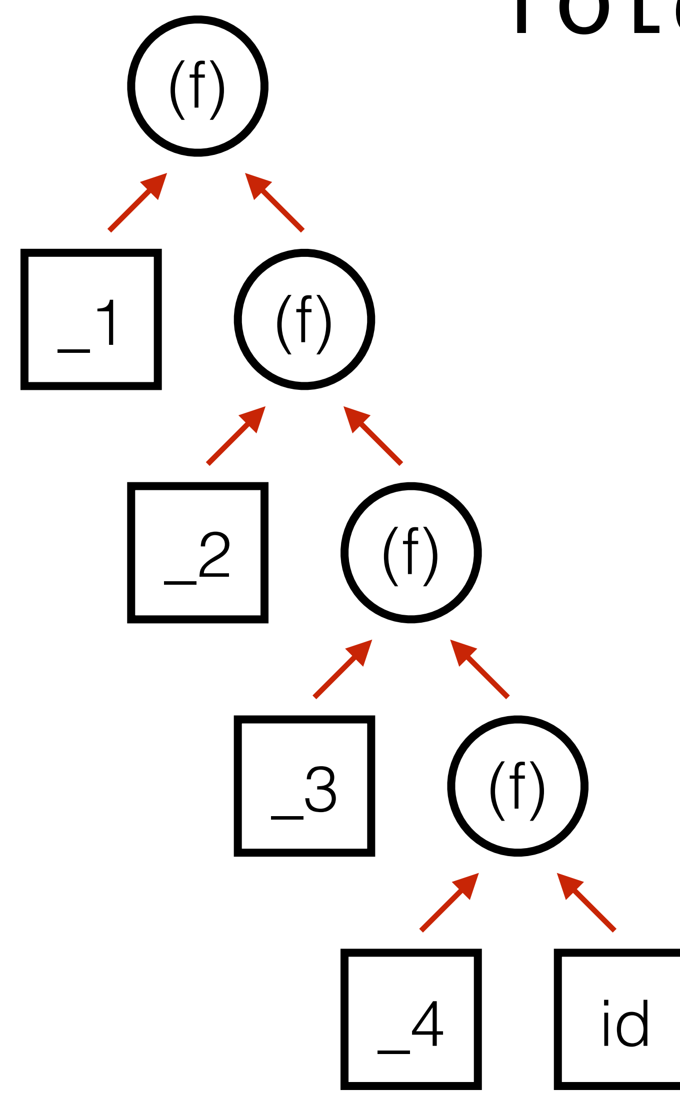
Note that these functions are curried

Reduce Revisited

`foldLeft`



`foldRight`



Reduce Revisited

```
case object Empty extends List[Nothing] {
  …
  def foldLeft[S](x: S)(f: (S, S) => S) = x
  def foldRight[S](x: S)(f: (S, S) => S) = x
}
```

Reduce Revisited

```
case class Cons[+T](head: T, tail: List[T])
extends List[T] {
  …
  def foldLeft[S >: T](x: S)(f: (S, S) => S) =
    tail.foldLeft(f(x, head))(f)

  def foldRight[S >: T](x: S)(f: (S, S) => S) =
    f(tail.foldRight(x)(f), head)
}
```

ops may not be commutative

```
f(tail.foldLeft((f(head,x))(f)
f(head, tail.foldRight(x)(f))
```

Reduce Revisited

```
def foldLeft[S >: T](x: S)(f: (S, S) => S) =
  tail.foldLeft(f(x, head))(f)
```

```
Cons(1,Cons(2,Cons(3,Empty))).foldLeft(0)(_+_) ↦
Cons(2,Cons(3,Empty)).foldLeft(0 + 1)(_+_) ↦
Cons(2,Cons(3,Empty)).foldLeft(1)(_+_) ↦
Cons(3,Empty).foldLeft(1 + 2)(_+_) ↦
Cons(3,Empty).foldLeft(3)(_+_) ↦
Empty.foldLeft(3 + 3)(_+_) ↦
Empty.foldLeft(6)(_+_) ↦
6
```

Reduce Revisited

```
def foldRight[S >: T](x: S)(f: (S, S) => S) =
  f(tail.foldRight(x)(f), head)
```

```
Cons(1,Cons(2,Cons(3,Empty))).foldRight(0)(_+_) ↦
Cons(2,Cons(3,Empty)).foldRight(0)(_+_) + 1 ↦
Cons(3,Empty).foldRight(0)(_+_) + 2 + 1 ↦
Empty.foldRight(0)(_+_) + 3 + 2 + 1 ↦
0 + 3 + 2 + 1 ↦
6
```

Reduce Revisited

```
abstract class List[+T] {
  …
  def reduce[S >: T](f: (S, S) => S): S
}
```

We can elide a zero element for the reduction provided that the list is non-empty

Reduce Revisited

```
case object Empty extends List[Nothing] {
  …
  def reduce[S](f: (S, S) => S) =
    throw ReduceError
}
```

case object ReduceError extends Error

Reduce Revisited

```
case class Cons[+T](head: T, tail: List[T])
extends List[T] {
  …
  def reduce[S >: T](f: (S, S) => S) =
    tail.foldLeft[S](head)(f)
}
```

We explicitly instantiate the type parameter to foldLeft. Without this, type inference will instantiate the type parameter based on the static type of head (which is T) and then signal an error that f is not of type (T, T) => T.

Forall and Exists

```
abstract class List[+T] {
  …
  def forall(p: T => Boolean) =
    map(p).foldLeft(true, _&&_)

  def exists(p: T => Boolean) =
    map(p).foldLeft(false, _||_)
}
```

Length

```
abstract class List[+T] { …
  def length: Int
}
case object Empty extends List[Nothing] { …
  def length = 0
}
case class Cons[+T](head: T, tail: List[T])
extends List[T] { …
  def length = map((_:T) => 1).reduce(_+_)
}
```

In what real contexts could we justify this definition of length?

Pointwise Addition

```
def pointwiseAdd(xs: List[Int], ys: List[Int]): List[Int] = {
  require (xs.length == ys.length)

  (xs, ys) match {
    case (Empty, Empty) => Empty
    case (Cons(x1, xs1), Cons(y1, ys1)) =>
      Cons(x1 + y1, pointwiseAdd(xs1,ys1))
  }
}
```

Generalizing to ZipWith

```
// in class List:
def zipWith[U,V](f: (T, U) => V)(that: List[U]): List[V] = {
  require (this.length == that.length)

  (this, that) match {
    case (Empty, Empty) => Empty
    case (Cons(x1,xs1), Cons(y1,ys1)) =>
      Cons(f(x1,y1), xs1.zipWith(f)(ys1))
  }
}
```

Defining The Zip Function

```
// in class List:
def zip[U](that: List[U]) = zipWith((_, _: U))(that)
```

Defining Flatten

```
def flatten[S](xs: List[List[S]]) = {
  xs.foldLeft(Empty)(_++_)
}
```

Defining Flatten

```
def flatten[S](xs: List[List[S]]) = {
  xs.foldLeft(Empty)(_++_)
}
```

Because of the specific type of List needed,
we define as a top level function

Defining FlatMap

```
abstract class List[+T] {
  …
  def flatMap[S](f: T => List[S]) =
    flatten(this.map(f))
  }
}
```

Defining FlatMap

```
abstract class List[+T] {
  …
  def flatMap[S](f: T => List[S]) =
    flatten(this.map(f))
  }
}
```

In contrast to flatten, our flatMap function can be defined on arbitrary lists

Defining FlatMap

- These definitions suggest that flatMap is the best thought of as the more primitive notion
- We can define flatMap as a method on lists directly and then define flatten in terms of it

Defining FlatMap

```
abstract class List[+T] { …
  def flatMap[S](f: Nothing => List[S]): List[S]
}

case object Empty extends List[Nothing] { …
  def flatMap[S](f: Nothing => List[S]) = Empty
}

case class Cons[+T](head: T, tail: List[T])
extends List[T] { …
  def flatMap[S](f: T => List[S]) =
    f(head) ++ tail.flatMap(f)
}
```

Defining Filter

```
abstract class List[+T] {
  …
  def filter[U](p: T => Boolean): List[T]
}
```

Defining Filter

```
case object Empty extends List[Nothing] {
  …
  def filter[U](p: T => Boolean) = Empty
}
```

Defining Filter

```
case class Cons[+T](head: T, tail: List[T])
extends List[T] {
  …
  def filter[U](p: T => Boolean) = {
    if (p(head)) Cons(head, tail.filter(p))
    else tail.filter(p)
  }
}
```

For Expressions

For Expressions

- As with all expressions, for expressions reduce to a value
- The value reduced to is a collection
- The type of collection produced depends on the types of collections iterated over
- Each iteration produces a value to include in the resulting collection

Many Maps and Filters Can Be Expressed Using For Expressions

```
for (x <- xs) yield square(x) + 1
```

Many Maps and Filters Can Be Expressed Using For Expressions

```
for (x <- xs) yield square(x) + 1
```

We call this a generator

Many Maps and Filters Can Be Expressed Using For Expressions

`for` *clauses* `yield` *body*

Many Maps and Filters Can Be Expressed Using For Expressions

```
for (i <- 1 to 10) yield square(i) + 1
```

Many Maps and Filters Can Be Expressed Using For Expressions

```
for (i <- 0 until 10) yield square(i) + 1
```

Does not include 10

Many Maps and Filters Can Be Expressed Using For Expressions

```
// BAD FORM
for (i <- 0 until xs.length)
  yield square(xs.nth(i)) + 1
```

Many Maps and Filters Can Be Expressed Using For Expressions

```
// Write this instead
for (x <- xs)
  yield square(x) + 1
```

For Expressions Can Also Include Filters

```
for (x <- xs if x >= 0)
  yield square(x) + 1
```

This is a filter

Filters in For Expressions

- Filters are attached to generators
- A given generator can have zero or more filters

For Expressions Can Also Include Filters

```
for (
  x <- xs
  if x >= 0
  if x % 2 == 0
) yield square(x) + 1
```

Clauses Can Be Enclosed in Braces Instead of Parentheses

```
for {
  x <- xs
  if x >= 0
  if x % 2 == 0
} yield square(x) + 1
```