CS558 Programming Languages Fall 2023 Lecture 6a

Andrew Tolmach Portland State University

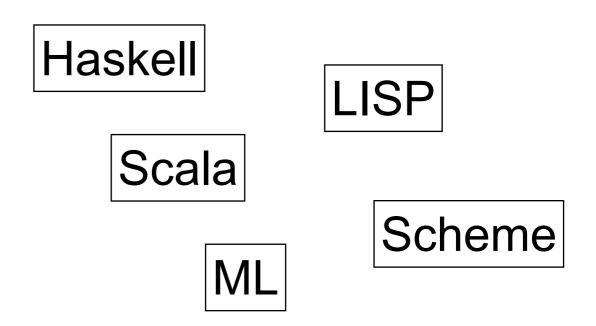
© 1994-2023

Functional Programming

Output the second se

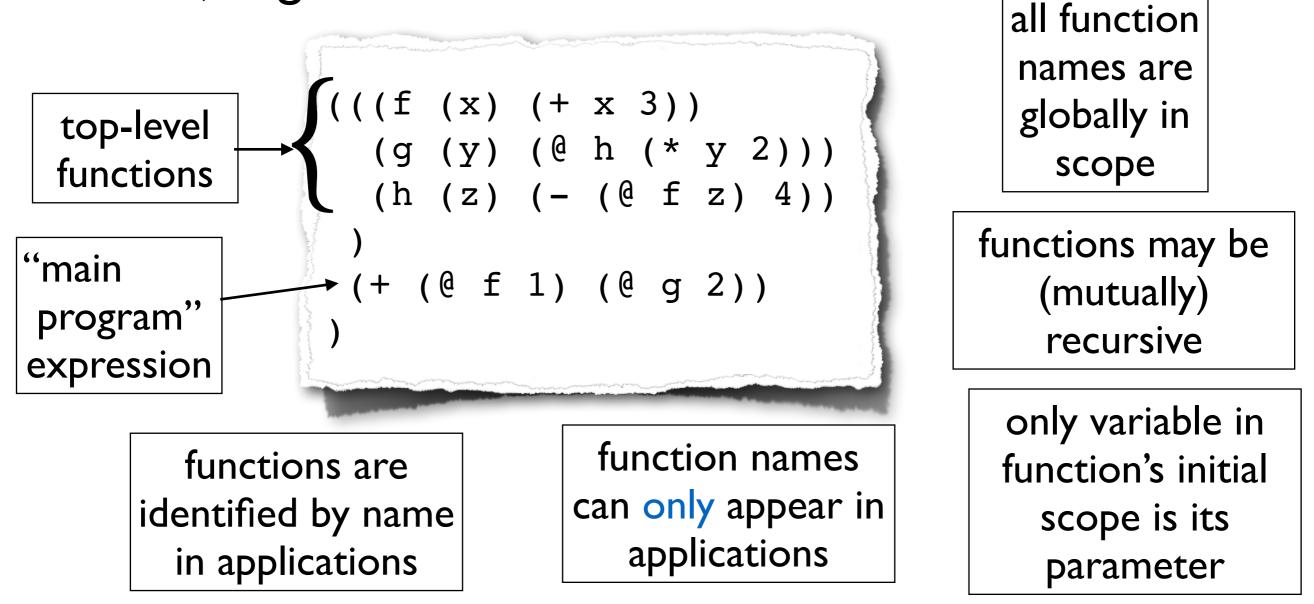
"First-class" functions

Emphasis on pure ("functional") computations (side effects restricted or prohibited)



Top-level Functions

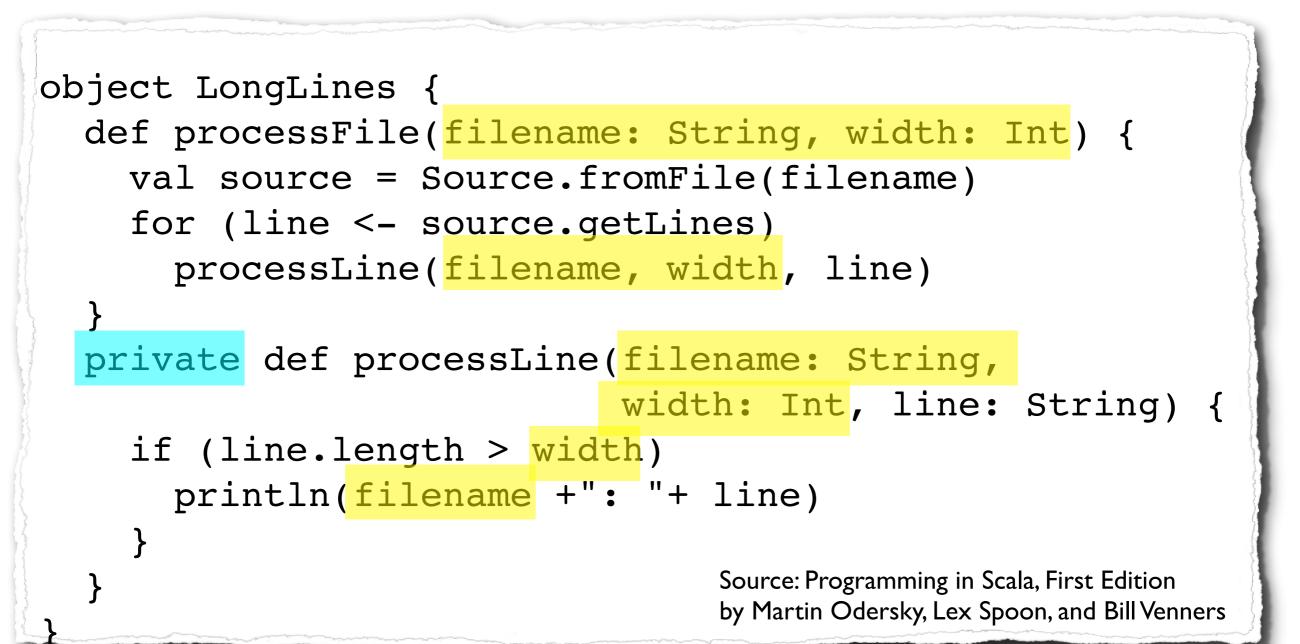
So far, we've been implicitly assuming that all functions are declared separately at program top level, e.g.



Almost Top-level Functions

Some languages (e.g. C) only allow top-level functions.

Other languages may have a top-level layer of modules or objects, with functions just inside. E.g. in Scala:



Nested Functions

Many languages let us define local functions

Inner function is only visible in scope of outer one, and can access variables bound in outer one. In Scala:

```
object LongLines {
  def processFile(filename: String, width: Int) {
    def processLine(line: String) {
       if (line.length > width)
         print(filename +": "+ line)
     }
    val source = Source.fromFile(filename)
    for (line <- source.getLines)</pre>
       processLine(line)
                                     Source: Programming in Scala, First Edition
                                     by Martin Odersky, Lex Spoon, and Bill Venners
```

First-class functions

- What happens if we treat functions as just another kind of value that we can manipulate in expressions?
- Slogan: functions are "first-class" values (just like integers or booleans or ...) if they can be:
 - bound to variables
 - passed to or from other ("higher-order") functions
 - stored in data structures

defined by anonymous program literals

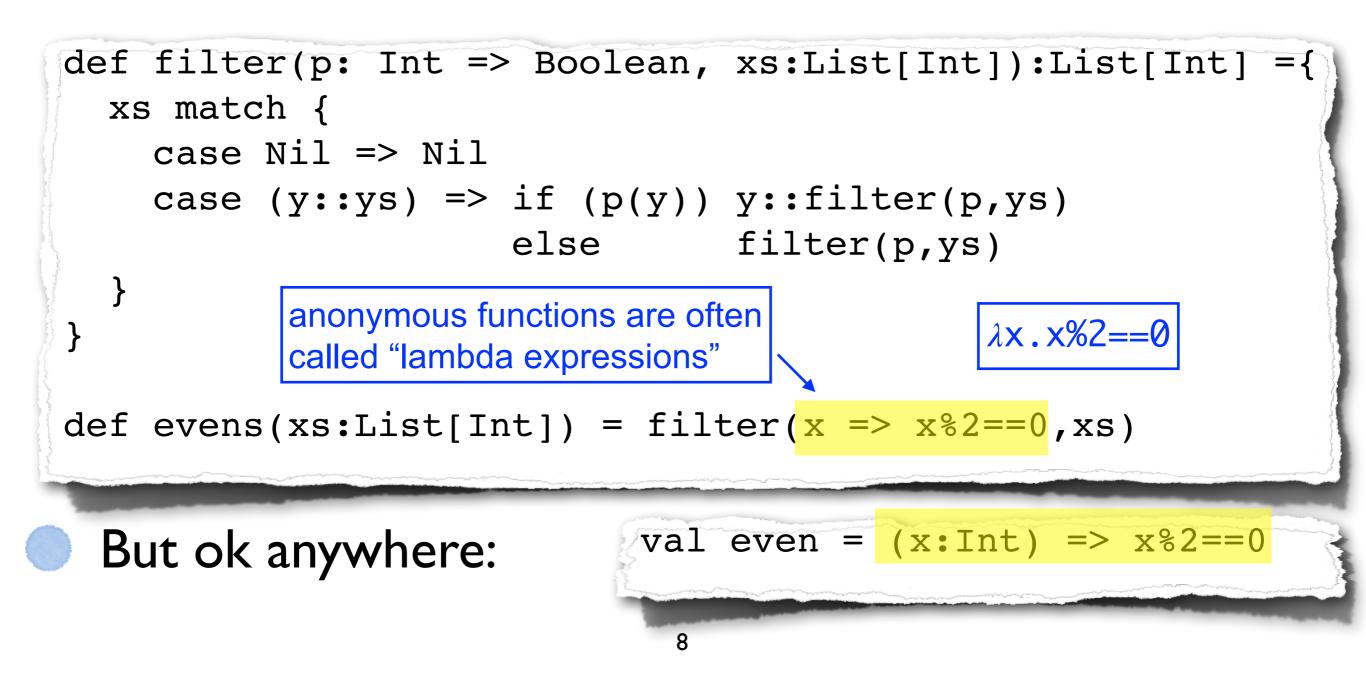
Functions as Parameters

- Allows us to parameterize by behaviors
- Particularly useful for working over collections

```
def filter(p: Int => Boolean, xs:List[Int]):List[Int] = {
    xs match {
        case Nil => Nil
        case (y::ys) => if (p(y)) y::filter(p,ys)
            else filter(p,ys)
        }
}
def even(x:Int): Boolean = x%2==0
def evens(xs:List[Int]) = filter(even,xs)
val v = evens(List(1,2,3,4)) // yields List(2,4)
```

Anonymous functions

- No need to name a function that is used just once
- Typically as an actual parameter:



Nested functions

A nested function (named or anonymous) can reference parameters of the enclosing function

```
def filter(p: Int => Boolean, xs:List[Int]):List[Int] = {
  def f(xs:List[Int]): List[Int] = xs match {
    case Nil => Nil
    case (y::ys) => if (p(y)) y::f(ys) else f(ys)
  }
 f(xs)
}
def multiplesOf(n:Int,xs:List[Int]) =
  filter(x => x%n==0, xs)
def evens(xs:List[Int]) = multiplesOf(2,xs)
def multsOf3(xs:List[Int]) = multiplesOf(3,xs)
```

Functions as results

A function can also be returned as the result of a function call. Here we use this to refactor filter:

```
def filter(p: Int => Boolean): List[Int] => List[Int] =
  def f(xs:List[Int]): List[Int] = xs match {
    case Nil => Nil
    case (y::ys) => if (p(y)) y::f(ys) else f(ys)
def multiplesOf(n:Int): List[Int] => List[Int] =
  filter(x => x n = 0)
val evens = multiplesOf(2)
val v = evens(List(1,2,3,4)) // yields List(2,4)
```

Curried Functions

- Like filter, any multi-parameter function can be coded as a nest of single-parameter functions each returning a function
- Such "Curried" functions can be either partially or fully applied
- Scala has extra syntactic sugar for them, e.g.

def compose[A](f: A=>A, g:A=>A)(x:A) = f(g(x))

val multsOf6 = compose(evens,multsOf3) _
val v = multsOf6(List.range(0,7)) // yields List(0,6)
val u = compose(evens,multsOf3)(List.range(0,7)) // same

Map

Currying is especially useful when passing partially applied functions to other higher-order functions

```
def map[A,B] (f: A => B) : List[A] => List[B] = {
  def g(xs:List[A]) : List[B] = xs match {
     case Nil => Nil
     case (y::ys) => f(y)::g(ys)
  }
  g
def pow(n:Int)(b:Int) : Int =
  if (n==0) 1 else b * pow (n-1)(b)
val a = map (pow(3)) (List(1,2,3)) // gives List(1,8,27)
```

Abstracting another pattern

```
def sum (l:List[Int]) : Int = l match {
  case Nil => 0
  case h::t => h + sum(t)
  }
```

sum of a list

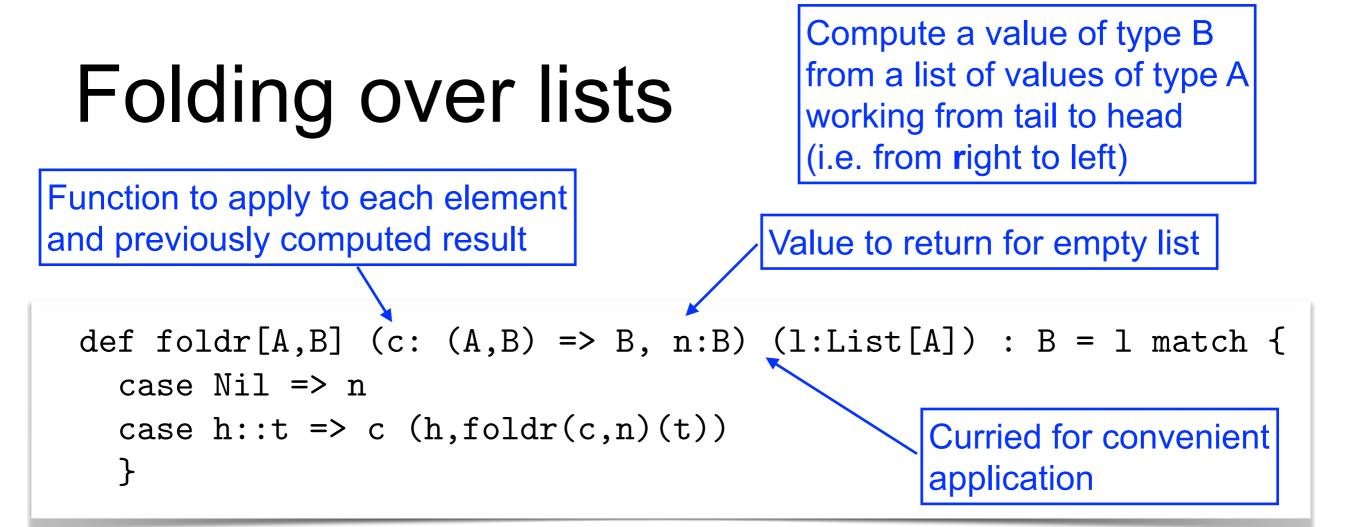
```
product of a list
def prod (l:List[Int]) : Int = 1 match {
    case Nil => 1
    case h::t => h * prod(t)
    }

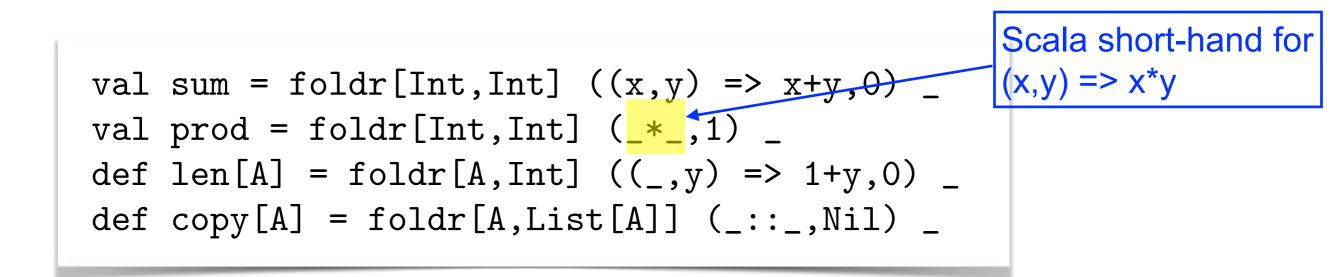
def len[A](l:List[A]) : Int = 1 match {
    case Nil => 0
    case _::t => 1 + len(t)
```

```
copy of a list
```

}

```
def copy[A](l:List[A]) : List[A] = 1 match {
  case Nil => Nil
  case h::t => h::copy(t)
  }
```





Visualizing folds

We can view foldr(c,n)(l) as replacing each ::
constructor in l by c and the Nil constructor by n

We can also define a foldl that accumulates a value from the left; this will sometimes be more efficient

In some languages fold is called reduce, because we "reduce" a list of values to a single value. Similar ideas appear in "map-reduce" frameworks for organizing massively parallel computations.