CS558 Programming Languages

Fall 2023
Lecture 6a

Andrew Tolmach
Portland State University

Functional Programming

- An alternative paradigm to imperative programming
- “First-class” functions
- Emphasis on pure (“functional”) computations (side effects restricted or prohibited)

Haskell

LISP

Scala

Scheme

ML

Top-level Functions

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

top-level functions

"main program" expression

```
(((f (x) (+ x 3))
  (g (y) (@ h (* y 2)))
  (h (z) (- (@ f z) 4))
 )
 (+ (@ f 1) (@ g 2))
)
```

all function names are globally in scope

functions may be (mutually) recursive

functions are identified by name in applications

function names can only appear in applications

only variable in function's initial scope is its parameter

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:

```
object LongLines {
  def processFile(filename: String, width: Int) {
    val source = Source.fromFile(filename)
    for (line <- source.getLines)
      processLine(filename, width, line)
  }
  private def processLine(filename: String,
                          width: Int, line: String) {
    if (line.length > width)
      println(filename +": "+ line)
    }
  }
}
```

Source: Programming in Scala, First Edition
by Martin Odersky, Lex Spoon, and Bill Venners

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)
      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:

```
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)
  }
}
```

anonymous functions are often called "lambda expressions"

λx.x%2==0

```
def evens(xs:List[Int]) = filter(x => x%2==0,xs)
```

- But ok anywhere:

```
val even = (x:Int) => x%2==0
```

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)
  }
  f _
}

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 = l match {
  case Nil => 1
  case h::t => h * prod(t)
  }
```

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

length of a list

copy of a list

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

Folding over lists

Compute a value of type B from a list of values of type A working from tail to head (i.e. from **right** to left)

Function to apply to each element and previously computed result

Value to return for empty list

```
def foldr[A,B] (c: (A,B) => B, n:B) (l:List[A]) : B = l match {
  case Nil => n
  case h::t => c (h,foldr(c,n)(t))
  }
```

Curried for convenient application

```
val sum = foldr[Int,Int] ((x,y) => x+y,0) _
val prod = foldr[Int,Int] (_*_,1) _
def len[A] = foldr[A,Int] ((_,y) => 1+y,0) _
def copy[A] = foldr[A,List[A]] (_::_,Nil) _
```

Scala short-hand for (x,y) => x*y

Visualizing folds

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

$l=$ `x1 :: (x2 :: (… :: (xn :: Nil)…))`

`foldr(_+_,0)(`l`)=  x1 + (x2 + (… + (xn + 0)…))`

- 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.