CS558 Programming Languages

Fall 2015
Lecture 2b

Andrew Tolmach
Portland State University

Top-level Functions

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

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

top-level functions

body expression

all function names are globally in scope

functions may be (mutually) recursive

only variable in function's initial scope is its parameter

functions are identified by name in applications

function names can only appear in applications

Almost Top-level Functions

- Some languages (e.g. C) only allow top-level functions.
- Other languages may have a top-level layer of, e.g., 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
  - defined by unnamed program literals
  - stored in data structures

Functions as Parameters

- Let's us 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)
  }
}

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)

def 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))
```

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

Curried Functions

- Currying is most 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)
```

Semantics of first-class functions

- What's in the "value" of a first-class function `f`?
- Roughly speaking, just `f`'s definition (its parameters and body expression)
- But nested functions can have free variables defined in an enclosing scope, and the behavior of the function depends on their values.
- To find those values, it suffices to record the environment surrounding the declaration of `f`
  - Store this in a "closure" representing `f`
- Later: can this semantics be implemented efficiently?

Semantics of first-class functions

```
case class ClosureV(x:String,b:Expr,e:Env) extends Value

def interpE(expr:Expr,env:Env) : Value = expr match {
  case Num(n) => NumV(n)
  case Add(l,r) => (interpE(l,env),interpE(r,env)) …
  case Fun(x,e) => ClosureV(x,e,env)
  case App(f,e) => interpE(f,env) match {
     case ClosureV(x,b,cenv) =>
       val v = interpE(e,env)
       interpE(b,cenv + (x -> v))
     case _ => throw InterpException (…)
     }
  case Var(x) => (env get x) match …
}
```