Lecture 4

1 Folding

The four functions in fig. 7.1 are examples of *folding functions*. A folding function "collapses" a list into a single value. A fold is characterized by a binary operation used to combine values and a base case for the empty list.

- In `sum`, the binary operation is `+` and the base value is `0`:

  ```
  sum(List(10, 20, 30)) = 10 + (20 + (30 + 0))
  ```

- In `product`, the binary operation is `*` and the base value is `1`:

  ```
  product(List(2, 3, 4)) = 2 * (3 + (4 * 1))
  ```

- In `concat`, the binary operation is `+` (string concatenation) and the base value is `""`:

  ```
  concat(List("X", "Y", "Z")) = "X" + ("Y" + ("Z" + ""))
  ```

- In `totalLen`, the binary operation is the following function:

  ```
  def binop (hd : String, tailLen : Int) : int = hd.length + tailLen
  ```

  and the base value is `0`:

  ```
  totalLen(List("hi", "cs220", "student")) = binop("hi", binop("cs220", binop("student" 0)))
  ```

```
def sum(lst: List[Int]): Int = lst match {
  case Nil => 0
  case hd :: tl => hd + sum tl
}
```

(a) Add all numbers in a list

```
def product(lst: List[Int]): Int = lst match {
  case Nil => 1
  case hd :: tl => hd + product tl
}
```

(b) Multiply all numbers in a list

```
def concat(lst: List[String]): String = lst match {
  case Nil => ""
  case hd :: tl => hd + concat tl
}
```

(c) Concatenate all strings in a list

```
def totalLen(lst: List[String]): Int = lst match {
  case Nil => 0
  case hd :: tl => hd.length + totalLen tl
}
```

(d) Sum the lengths of all strings in a list

Figure 7.1: Several functions that *fold* over a list.

Now that we've seen the common pattern, it's clear that we abstract these functions to a higher-order function that takes three arguments (1) the original list, (2) a two-argument function instead of a fixed binary operator, and (3) a value for the base-case, instead of a fixed base-case:

```
def fold(baseCase: ???, binOp: (???, ???) => ???, lst: List[???]): ??? = lst match {
  case Nil => baseCase
  case head :: tail => binOp(head, fold(baseCase, binOp, tail))
}
```

We can rewrite the functions in fig. 7.1 more succinctly as follows:

```
def sum(lst : List[Int]) = fold(0, (x: Int, y: Int) => x + y, lst)

def product(lst : List[Int]) = fold(1, (x: Int, y: Int) => x * y, lst)

def concat(lst : List[String]) = fold("", (x: String, y: String) => x + y, lst)

// binOp was defined above
def totalLen(lst : List[Int]) = fold(0, binOp, lst)
```

Note that in `binOp`, the types of the two arguments are different. The type of the first argument is the same as the type of the elements in the list. The type of the second argument is the same as the type of the result of the function. This is because when `binOp` is applied (internally within `fold`), the first argument to `binOp` is an element of the list, and the second argument is an intermediate result. Therefore, we can fill in the types in `fold` as follows:

```
def fold[A,B](baseCase: B, binOp: (A, B) => B, lst: List[A]): B = lst match {
  case Nil => baseCase
  case head :: tail => binOp(head, fold(baseCase, binOp, tail))
}
```

2 Functions are Values

Functions can be passed as arguments to functions. The last lecture introduced functions can be passed as arguments to other functions. For example, the function `filter(f, lst)` produces a list of all the items of `lst` for which `f` is `true`:

```
def filter[A](f: A => Boolean, lst: List[Int]): List[A] = lst match {
  case Nil => Nil
  case Cons(head, tail) =>
    f(head) match {
      case true => head :: filter(f, tail)
      case false => filter(f, tail)
    }
}
```

This allowed us to write several functions very easily, for example:

```
def isEven(n: Int): Boolean = n % 2 == 0

def onlyEvens(lst: List[Int]): List[Int] = {
  filter(isEven, lst)
}

def nonZero(n: Int): Boolean = n != 0

def onlyNonZero(lst: List[Int]): List[Int] = {
  filter(nonZero, lst)
}
```

Functions can be nested within other functions. If our goal was to write the list-processing functions, then the functions `isEven` and `nonZero`, are cluttering our code. Another programmer may think that they are signficant, when them simply exist as trivial helper functions.

Functions can be nested within other functions:

```
def onlyEvens(lst: List[Int]): List[Int] = {
  def isEven(n: Int): Boolean = n % 2 == 0
  filter(isEven, lst)
}

def onlyNonZero(lst: List[Int]): List[Int] = {
  def nonZero(n: Int): Boolean = n != 0
  filter(nonZero, lst)
}
```

In particular, the names `isEven` and `nonZero` are *not defined* outside their respective functions.

Functions can produce functions. Functions can also produce other functions. For example:

```
def makeAdder(x: Int): Int => Int = {
  def addX(y: Int): Int = x + y
  addX(x)
}

val addThree = makeAdder(3)
test("add3 test") {
  assert(addThree(10) == 13)
}
```

Functions do not need to be named. Unlike other values, functions seem to have the following special property: every function has a name, but the other kinds of values do not. For example, we can simply write `1 :: 2 :: 3 :: Nil` and don't need to give this list a name. So far, all of the functions we've seen start with `def functionName`.

It turns out that this is just a convention. As with other values, functions don't need names. For example, here is a function that adds two numbers:

```
((x: Int, y: Int) => x + y)
```

This function does not have a name, but it can be applied just like any other function:

```
((x: Int, y: Int) => x + y)(10, 20)
```

The code above is not easy to read. It will be a lot clearer of we give the function a name. You already know how to name a function using `def`. But, all other values are named using `val`. In fact, we can use `val` to name functions too, just like any other type of value:

```
val adder = ((x: Int, y: Int) => x + y)

adder(10, 20)
```

You should think of `def` as a convenient shorthand for naming functions. That is, these two definitions are equivalent:

```
val adder = ((x: Int, y: Int) => x + y)

def adder(x: Int, y: Int) = x + y
```

In general, it is a good idea to name your functions. But, there are certain situations where a short, anonymous function can make your code easier to read and write.

For example, we earlier defined the `doubleAll` function, which doubles every number in a list of numbers. Here is simple, one-line definition using an anonymous function as an argument to `map`:

```
def doubleAll(lst: List[Int]) = map((n: Int) => n * 2, lst)
```

The anonymous function itself is extremely simple and the name of the enclosing function, `doubleAll`, really makes it clear what it does.

For another example, suppose we want to remove all the odd numbers in a list. We can do this using `filter` and a short, anonymous function:

```
def removeOdds(lst: List[Int]) = filter((n: Int) => n % 2 == 0, lst)
```

In these kinds of situations, anonymous functions can be very helpful.

Functions can be stored in data structures. The following datatype can stores two functions inside it:

```
case class Foo(m1: Int => Int, m2: Int => Int)
```

We can create values of type `Foo` in the following way:

```
val myFoo = Foo((x: Int) => x + 1, (y: Int) => y * 20)

assert(myFoo.m1(10) == 11)
assert(myFoo.m2(10) == 200)
```

Admittedly, this isn't very useful, but notice that the function applications look a lot like method calls.

2.1 Some Definitions

Here are some terminology that gets thrown around when comparing programming languages and programming techniques.

- *Higher-order functions* are functions that consume or return other functions as values. You can tell if a function is higher-order by inspecting its type. Does it have any nested uses of `=>` in the type? If so, it is a higher-order function.
- *First-class functions* is a property of a programming language. For a programming language to have first-class functions, it must treat functions as values, with all the rights and privileges that other values have. You must be able to use functions as arguments, produce functions as results, and store functions in data-structures.

3 Scope and Substitution

Global vs. Local Variables Why does the following program raise an error?

```
def f(x: Int) = {
  val y = x + 10
  y
}
f(11)
y
```

Although the program defines a variable called `y`, the *scope* of the variable is limited to the function `f`. Therefore, we cannot refer to the variable outside the function, which is why we get an error.

Substitution What does the following program produce?

```
val x = 20
def f(x: Int): Int = {
  x + 5
}
f(10) + x
```

The result is `35`. Within the body of `f`, the the name `x` refers to the argument to the function, which *shadows* the global variable `val x = 20`. Therefore, `f(10) == 15`. However, outside the function `x` refers to the global variable `x == 20`, and `15 + 20 == 35`.

We can make this argument more precise by *substituting* variables with their values:

- Given the orignal program above, we first substitute the global `x` with its value `20`, to get the following program:

  ```
  def f(x: Int): Int = {
    x + 5
  }
  f(10) + 20
  ```

  Notice that we substituted the `x` on the last line with `20`, but left the `x` within `f` unchanged, since it referred to the argument `x`.

- Next, we can apply the function `f` by substituting its argument `x` with the value `10`:

  ```
  (10 + 5) + 20
  ```

- Finally, we are left with a simple arithmetic expression that is trivial to evaluate.

Here is a more compact way of making the same argument:

	Expression	Reasoning
	val x = 20; def f(x: Int): Int = { x + 5 }; f(10) + x	Original expression
=	def f(x: Int): Int = { x + 5 }; f(10) + 20	Substitute x with 20
=	(10 + 5) + 20	Inline f and substitute x with 10
=	15 + 20	Evaluate 10 + 5
=	35	Evaluate 15 + 20

Nested Functions Scope can appear tricker when working with higher-order functions. But, we can use the same substitution principle to reason about nested functions.

For example, we wrote the `makeAdder` higher-order function before and used it to create the `add10` function:

```
def makeAdder(x: Int): Int => Int = {
  def add(y): Int = { x + y }
  add
}

val add10 = makeAdder(10)
```

The following calculate starts with the definition above (written in a single line for brevity) and shows that it is equivalent to the more obvious definition of `add10`:

	Expression	Reasoning
	def makeAdder(x: Int): Int =>Int = def add(y): Int = x + y ; add ; val add10 = makeAdder(10)	
=	val add10 = { def add(y): Int = { x + 10 }; add }	Inline makeAdder and substitute y with 10
=	val add10 = { val add = ((y: Int) =>{ x + 10 }); add }	Rewrite add using val notation
=	val add10 = ((y: Int) =>{ x + 10 })	Substitute add with its definition
=	def add10(y: Int) = x + 10	Rewrite add using def notation

In fact, every line in this calculation is a valid Scala program (which you should check!) and all the lines are truly equivalent to each other.

It is always possible to perform these kinds of calculations to simplify expressions with higher-order functions. For large programs, a detailed calculation may be infeasibly wrong. But, if you understand how these calculuations work in detail on small programs, you'll be able to reason carefully about larger programs without needing to actually do the calculuates in detail.

4 Storing Functions in Data Structures

Although we've seen that functions can be stored in data structures, we haven't seen any good examples that suggest why we may want to do so. For example, here is yet another type that stores functions in its fields:

```
case class Point(
  getX: () => Double,
  getY: () => Double,
  magnitude: () => Double,
  add: Point => Point)
```

Above, `getX`, `getY`, and `magnitude` are fields that store functions that consume zero arguments and produce `Double`s.

Consider the following higher-order function, which consumes two coordinates and produces a point:

```
def makePoint(x: Double, y: Double): Point = {
  def getX(): Double = x
  def getY(): Double = y
  def magnitude(): Double = math.sqrt(x * x + y * y)
  def add(other: Point): Point = makePoint(x + other.getX(), y + other.getY())
  Point(getX, getY, magnitude, add)
}
```

We can now construct points using `makePoint` and invoke the functions that it defines. For example:

```
val pt1 = makePoint(3.0, 4.0)
assert(pt1.magnitude() == 5.0)
val pt2 = pt1.add(makePoint(10.0, 10.0))
assert(pt2.getX() == 13.0)
assert(pt1.getY() == 4.0)
```

Note that this looks an awful lot like vanilla object-oriented programming. In fact, you could say that `makePoint` is a *constructor* that produces object that have the *interface* `Point` and that `getX`, `getY`, etc. are *methods* that the interface defines. Furthermore, `x` and `y` are *private fields*, since they are not in scope outside the `makePoint` function.[1]

[1] However, `x` and `y` are in scope for the methods, since they are local to `makePoint`.

This example shows that you can encode object-oriented programs using higher-order functions. In fact, this example makes essential use of scope, functions stored in data structures, functions that return functions (stored in data structures), and all the other characteristics of first-class functions. With just a little more effort, other object-oriented ideas like inheritence can be encoded using higher-order functions too. Conversely, it turns out that higher-order functions can be encoded using objects, which we will see later in the course. The point of this exercise is that functional programming and object-oriented programming are two sides of the same coin.