This Scala notebook uses *BeakerX*, a Two Sigma Open Source project that enhances Jupyter.

http://beakerx.com/ (http://beakerx.com/)

```
In [1]:  scala.util.Properties.versionMsg

Out[1]:  Scala library version 2.11.12 -- Copyright 2002-2017, LAMP/EPFL
```

The Functional Way: Review from Lecture 1

Example 1: Buying a Book

In Lecture 1 we discussed an example of an imperative function that implements logic for buying a digital book. We drew a diagram that looked something like this:

```
(credit card number) → [ buy ] → (digital book)
                                ↓
             (side effect: card charged)
```

In contrast, when using the functional programming paradigm we try to avoid side effects. Therefore, our functional version of the same diagram for the buy function looked something like this:

```
(credit card number) → [ buy ] → (digital book, card charge event)
```

The result of buy is a *pair* rather than a single value, since it consists of both the result value (i.e., the book that was bought) as well as a value representing the change in state caused by this function. A Scala implementation of this buy function might have a signature like the following:

```
In [2]:  trait CreditCardNumer
         trait DigitalBook
         trait ChargeEvent

         def buy(creditCardNumber: CreditCardNumer): (DigitalBook, ChargeEvent) = ???

Out[2]:  defined trait CreditCardNumer
         defined trait DigitalBook
         defined trait ChargeEvent
         buy: (creditCardNumber: CreditCardNumer)(DigitalBook, ChargeEvent)
```

This is a very basic example of typically how State is modeled in functional programming: The state is represented as an explicit value (rather than an implicit union of a bunch of global variables or other resources). That explicit state value is threaded through the computation, updated copies are returned by functions that modify the state, and that modified state value then needs into be threaded to the next function.

Example 2: Stateful Stack

Since `scala.collection.immutable.Stack` was deprecated in favor of `List`, let's declare a type alias.

We'll alias `List[Int]` so that we don't have to deal with type parameters on our `Stack` type.

```
In [3]: type Stack = List[Int]

Out[3]: defined type alias Stack
```

Note that the object implementing the factory function is still `List` (we only created an alias for the type, not the companion object).

```
In [4]: val stack: Stack = List(4, 3, 2, 1)

Out[4]: [[4, 3, 2, 1]]
```

Threading State Explicitly

To simulate stack mutation in functional code, our stack operations return a pair: `(newStackState, operationResultValue)`

```
In [5]: def statePush(x: Int, stack: Stack): (Stack, Unit) = (x :: stack) -> {}

        def statePop(stack: Stack): (Stack, Int) = {
          val x :: xs = stack
          xs -> x
        }

Out[5]: statePush: (x: Int, stack: Stack)(Stack, Unit)
        statePop: (stack: Stack)(Stack, Int)
```

push followed by pop should yield the original stack, and the popped item should be the same item we pushed.

```
In [6]: val pushedItem = 5
        val pushResult @ (stack1, _) = statePush(pushedItem, stack)
        pushResult

Out[6]: (List(5, 4, 3, 2, 1),())
```

```
In [7]: val popResult @ (stack2, poppedItem) = statePop(stack1)
        popResult

Out[7]: (List(4, 3, 2, 1),5)
```

```
In [8]: stack == stack2

Out[8]: true
```

```
In [9]: pushedItem == poppedItem

Out[9]: true
```

Now let's say we want a function to swap the top two elements on the stack. We could implement it like this:

```
In [10]: def swapTopPairV1(stack: Stack): (Stack, Unit) = {
           val x :: y :: zs = stack
           (y :: x :: zs) -> {}
         }

Out[10]: swapTopPairV1: (stack: Stack)(Stack, Unit)
```

... but that's too easy!

Since we want to model imperative state changes in functional code, we'll implement our `swap` function in terms of pop and `push`:

```
In [11]: def swapTopPairV2(stack: Stack): (Stack, Unit) = {
           val (stack1, x) = statePop(stack)
           val (stack2, y) = statePop(stack1)
           val (stack3, _) = statePush(x, stack2)
           val (stack4, _) = statePush(y, stack3)
           stack4 -> {}
         }

Out[11]: swapTopPairV2: (stack: Stack)(Stack, Unit)
```

Versions 1 and 2 of the `swapTopPair` function above are equivalent.

```
In [12]: val swappedStack1 = swapTopPairV1(stack)
         val swappedStack2 = swapTopPairV2(stack)
         swappedStack1 == swappedStack2

Out[12]: true
```

It would be nice if we didn't have to explicitly thread that state around. E.g., if we accidentally used `state2` a second time in place of `state3`, we would get the wrong result.

Threading State as a Monad

If we create a new Monad type implementing `map` and `flatMap`, we can use those methods to implicitly thread the state through successive clauses of a for-expression.

In [13]:
```
trait State[S, A] { self =>
  def run(initial: S): (S, A)

  def map[B](f: A => B): State[S, B] = new State[S, B] {
    def run(state: S): (S, B) = {
      val (statePrime, result) = self.run(state)
      statePrime -> f(result)
    }
  }

  def flatMap[B](f: A => State[S, B]): State[S, B] = new State[S, B] {
    def run(state: S): (S, B) = {
      val (statePrime, result) = self.run(state)
      f(result).run(statePrime)
    }
  }
}

type StackState[A] = State[Stack, A]
```

Out[13]:
```
defined trait State
defined type alias StackState
```

Now we can re-write our push and pop operations as State Monad combinator functions:

In [14]:
```
def pop() = new StackState[Int] {
  def run(stack: Stack) = stack.tail -> stack.head
}

def push(x: Int) = new StackState[Unit] {
  def run(stack: Stack) = (x :: stack) -> {}
}
```

Out[14]:
```
pop: ()StackState[Int]
push: (x: Int)StackState[Unit]
```

Note that as of Scala 2.12, the language supports expanding lambda expressions into *Single Abstract Member* (SAM) types (https://www.scala-lang.org/news/2.12.0/#lambda-syntax-for-sam-types), similar to how lambdas work in Java 8, which eliminates some of the boilerplate above:

```
def pop(): StackState[Int] = {
  stack => stack.tail -> stack.head
}

def push(x: Int): StackState[Unit] = {
  stack => (x :: stack) -> {}
}
```

However, BeakerX currently only supports Scala 2.11, so we're stuck with the verbose syntax here.

Implementing Swapping with State Monad

Now we can implement our swap function with the StackState monad:

```
In [15]: stack
```

```
Out[15]: [[4, 3, 2, 1]]
```

```
In [16]: def swapTopPairV3(): StackState[Unit] = for {
           x <- pop()
           y <- pop()
           _ <- push(x)
           _ <- push(y)
         } yield {}

         swapTopPairV3().run(stack)
```

```
Out[16]: (List(3, 4, 2, 1),())
```

Remember that the for-expression above expands to a chain of `flatMap` and `map` calls:

```
In [17]: def desugaredSwapTopPairV3(): StackState[Unit] = {
           pop().flatMap { x =>
             pop().flatMap { y =>
               push(x).flatMap { _ =>
                 push(y).map { _ =>
                   ()
                 }
               }
             }
           }
         }

         desugaredSwapTopPairV3().run(stack)
```

```
Out[17]: (List(3, 4, 2, 1),())
```

Is that really so much simpler? Yes, the for-expression version does look much less cluttered; however, the control flow is now significantly more complex. Here's another example for comparison:

```
In [18]: // Explicit state threading
         locally {
           val (stack1, _) = swapTopPairV2(stack)
           val (stack2, _) = statePush(4, stack1)
           val (stack3, _) = swapTopPairV2(stack2)
           statePop(stack3)
         }
```

```
Out[18]: (List(4, 4, 2, 1),3)
```

```
In [19]: // Implicit state threading with the State Monad
         locally {
           for {
             _ <- swapTopPairV3()
             _ <- push(4)
             _ <- swapTopPairV3()
             result <- pop()
           } yield result
         }.run(stack)

Out[19]: (List(4, 4, 2, 1),3)
```

```
In [20]: // Or equivalently desugared
         locally {
           swapTopPairV3().flatMap { _ =>
             push(4).flatMap { _ =>
               swapTopPairV3().flatMap { _ =>
                 pop().map {
                   result => result
                 }
               }
             }
           }
         }.run(stack)

Out[20]: (List(4, 4, 2, 1),3)
```

Using the for-expression syntax with the State Monad *does* give us a nicer way to thread state in a functional way. However, it has its limitations. For example, what if I want to push a List of elements onto a stack? We might want to implement that recursively with a function that looks something like this:

```
In [21]: def pushAllSkeleton(xs: List[Int]): StackState[Unit] = xs match {
           case Nil => ???
           case y :: ys => ???
         }

Out[21]: pushAllSkeleton: (xs: List[Int])StackState[Unit]
```

We need a way to simply yield unit when we reach the base case. We'll add a new State Monad combinator for that:

```
In [22]: def inject[S, R](result: R) = new State[S, R] {
           def run(s: S) = s -> result
         }

Out[22]: inject: [S, R](result: R)State[S,R]
```

Now we can "inject" `Unit` as a no-op in the base case, and combine `push` with a recursive `pushAll` in the recursive case.

```
In [23]: def pushAll(xs: List[Int]): StackState[Unit] = xs match {
           case Nil => inject {}
           case y :: ys => for {
             _ <- push(y)
             _ <- pushAll(ys)
           } yield ()
         }

         pushAll(List(10, 11, 12)).run(stack)

Out[23]: (List(12, 11, 10, 4, 3, 2, 1),())
```

However, that's not tail recursive. The order in which the expressions are executed is also confusing. What's actually happening here is that we're recursively building a chain of `StackState` monad instances, which we would then thread a stack through later to simulate execution. This is similar to how we built parsers using the Parser Combinators, and then actually parsed the input afterward by calling `parser.parseAll(...)` on the resulting parser object.

In contrast, the version using explicit state threading is a bit uglier, but it's tail recursive (via `foldLeft`). The control flow is also much easier to understand in this case, since there is no intermediate combinator object constructed, and no delayed execution of our state-transforming functions.

```
In [24]: def pushAllExplicit(xs: List[Int], stack: Stack): (Stack, Unit) = {
           val basePair = stack -> {}
           xs.foldLeft(basePair) {
             (accPair, x) =>
               val (currentStack, _) = accPair
               (x :: currentStack) -> {}
           }
         }

         pushAllExplicit(List(10, 11, 12), stack)

Out[24]: (List(12, 11, 10, 4, 3, 2, 1),())
```

Advanced Haskell-inspired libraries like Scalaz and Cats introduce other Monad tools that might let us build a better version of `pushAll` — but with an even steeper learning curve than what we have here.

The State Monad is a neat idea, but unless you're programming in Haskell (where you have no choice because the language itself prevents writing side-effecting code) you probably won't use it. This is one of those cases where the practicality of imperative style tends to win over the pureness of the functional style.