Exercises — Week Eight

The functional circus continues…

Spring term 2016

You need to have installed the current Scala distribution and the current Java distribution before commencing these exercises. Don't forget the documentation for both distributions as they will come in very useful.

For some of these exercises you will need the test code you used in a previous exercise sheet, namely the classes below the `atomicscala` folder on the repo. Any source code examples are available under the `week08` folder.

There are four broad areas begin examined:

1. Generics,
2. Consolidation of existing concepts,
3. Sequences, and
4. Higher-Order Functions.

Generics

1. Given the following type:

```
sealed trait IntList
final case object End extends IntList
final case class Pair(head: Int, tail: IntList) extends IntList
```

   (a) Change the name to `LinkedList` and make the type of data stored in the list generic.

   (b) Now implement `length`, returning the length of the `LinkedList`. Some test cases are below:

```
val example = Pair(1, Pair(2, Pair(3, End())))
assert(example.length == 3)
assert(example.tail.length == 2)
assert(End().length == 0)
```

   (c) Now implement a method `contains` that determines whether or not a given item is in the list. Ensure your code works with the following test cases:

```
val example = Pair(1, Pair(2, Pair(3, Empty())))
assert(example.contains(3) == true)
assert(example.contains(4) == false)
assert(Empty().contains(0) == false)
// This should not compile
// example.contains("not an Int")
```

(d) Implement a method `apply` that returns the nth item in the list. Ensure your solu on works with the following test cases:

```
val example = Pair(1, Pair(2, Pair(3, Empty())))
assert(example(0) == 1)
assert(example(1) == 2)
assert(example(2) == 3)
assert(try {
        example(3)
        false
} catch {
        case e: Exception => true
})
```

[Yes, Scala does have java type exceptions — although we try to avoid them.]

(e) Throwing an exception isn't very clean. Whenever we throw an exception we lose type safety as there is nothing in the type system that will remind us to deal with the error. It would be much better to return some kind of result that encodes we can *succeed* or *fail*. Consider the following type:

```
sealed trait Result[A]
case class Success[A](result: A) extends Result[A]
case class Failure[A](reason: String) extends Result[A]
```

Change your `apply` method so it returns a `Result`, with a failure case indicating what went wrong. Here are some test cases to help you:

```
assert(example(0) == Success(1))
assert(example(1) == Success(2))
assert(example(2) == Success(3))
assert(example(3) == Failure("Index out of bounds"))
```

2. Consider the following code:

```
sealed trait IntList {
  def length: Int =
    this match {
      case End => 0
      case Pair(hd, tl) => 1 + tl.length
    }

  def double: IntList =
    this match {
      case End => End
      case Pair(hd, tl) => Pair(hd * 2, tl.double)
    }
```

```
  def product: Int =
    this match {
      case End => 1
      case Pair(hd, tl) => hd * tl.product
    }
  def sum: Int = this match {
    case End => 0
    case Pair(hd, tl) => hd + tl.sum
  }
}
```

All of these methods have the same general pattern, which is not surprising as they all use structural recursion. It would be nice to be able to remove the duplication. Below we have sketched out an abstraction over `sum`, `length`, and `product`

```
def abstraction(end: Int, f: ???): Int =
        this match {
                case End => end
                case Pair(hd, tl) => f(hd, tl.abstraction(f, end))
        }
```

(a) Rename this function to `fold`, which is the name it is usually known as, and complete the implementation.

(b) Now reimplement `sum`, `length`, and `product` in terms of `fold`.

(c) Is it more convenient to rewrite methods in terms of `fold` if they were implemented using pattern matching or polymorphism? What does this tell us about the best use of `fold`?

(d) Why can't we write our `double` method in terms of `fold`? Is it feasible we could if we made some change to `fold`?

(e) Implement a generalised version of `fold` and rewrite `double` in terms of it.

Consolidation

3. Given the following code:

```
object One {
  def sqrtIter(guess: Double, x: Double): Double =
    if (isGoodEnough(guess, x)) guess
    else sqrtIter(improve(guess, x), x)

  def improve(guess: Double, x: Double) =
    (guess + x / guess) / 2

  def isGoodEnough(guess: Double, x: Double): Boolean =
    abs(square(guess) - x) < 0.001
```

```
  def sqrt(x: Double) = sqrtIter(1.0, x)

  def square(x: Double) = x * x

  def abs(x: Double) = if (x > 0) x else -x
}
```

The `isGoodEnough` test is not very precise for small numbers and might lead to non-termination (why?). Design a different version of `isGoodEnough` which does not have these problems.

4. Given the following function:

   ```
   def factorial(n: Int): Int = if (n == 0) 1 else n * factorial(n - 1)
   ```

   Design a tail-recursive version of `factorial`.

5. Can you write a tail-recursive version of a function `sum`, that computes the sum of the values of a function at points over a given range, by filling in the `???`'s?

   ```
   def sum(f: Int => Int)(a: Int, b: Int): Int = {
           def iter(a: Int, result: Int): Int = {
                   if (???) ???
                   else iter(???, ???)
           }
           iter(??, ??)
   }
   ```

6. Write a function `product` that computes the product of the values of a function at points over a given range.

7. Write `factorial` in terms of `product`.

8. Can you write a more general function which generalises both `sum` and `product`?

9. Given the following code:

   ```
   trait IntSet {
     def incl(x: Int): IntSet

     def contains(x: Int): Boolean
   }

   case class EmptySet() extends IntSet {
     override def contains(x: Int): Boolean = false

     override def incl(x: Int): IntSet =
       new NonEmptySet(x, new EmptySet, new EmptySet)
   }

   case class NonEmptySet(elem: Int, left: IntSet, right: IntSet)
   ```

```
  extends IntSet {
  override def contains(x: Int): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true

  override def incl(x: Int): IntSet =
    if (x < elem) new NonEmptySet(elem, left incl x, right)
    else if (x > elem) new NonEmptySet(elem, left, right incl x)
    else this
}
```

Write methods `union` and `intersection` to form the union and intersection between two sets.

Add a method

```
def excl(x: Int)
```

to return the given set without the element `x`. To accomplish this, it is useful to also implement a test method

```
def isEmpty: Boolean
```

for sets.

10. Write an implementation `Integer` of integer numbers. The implementation should support all operations of class `Nat`:

```
abstract class Nat {
  def isZero: Boolean

  def predecessor: Nat

  def successor: Nat

  def +(that: Nat): Nat

  def -(that: Nat): Nat
}

class Succ(x: Nat) extends Nat {
  override def isZero: Boolean = false

  override def predecessor: Nat = x

  override def successor: Nat = new Succ(this)

  override def +(that: Nat): Nat = x + that.successor

  override def -(that: Nat): Nat = if (that.isZero) this
```

```
  else x - that.predecessor
}

object Zero extends Nat {
  override def isZero: Boolean = true

  override def predecessor: Nat = sys.error("negative number")

  override def successor: Nat = new Succ(Zero)

  override def +(that: Nat): Nat = that

  override def -(that: Nat): Nat = if (that.isZero) Zero
  else sys.error("negative number")
}
```

while adding two methods:

```
def isPositive: Boolean
def negate: Integer
```

The first method should return `true` if the number is positive. The second method should negate the number. Do not use any of Scala's standard numeric classes in your implementation.

(*Hint:* There are two possible ways to implement `Integer`. One can either make use the existing implementation of `Nat`, representing an integer as a natural number and a sign. Alternatively one can generalise the given implementation of `Nat` to `Integer`, using the three subclasses `Zero` for 0, `Succ` for positive numbers and `Pred` for negative numbers.)

Sequences

11. Consider the following definitions representing trees of integers. These definitions can be seen as an alternative representation of `IntSet`:

```
abstract class IntTree
case object EmptyTree extends IntTree
case class Node(elem: Int, left: IntTree, right: IntTree) extends IntTree
```

Complete the following implementations of function `contains` and `insert` for `IntTree`'s.

```
def contains(t: IntTree, v: Int): Boolean = t match { ...
        ...
}
def insert(t: IntTree, v: Int): IntTree = t match { ...
        ...
}
```

12. As an example of how lists can be processed, consider sorting the elements of a list of numbers into ascending order. One simple way to do so is *insertion sort*, which works as follows:

    > To sort a non-empty list with first element `x` and rest `xs`, sort the remainder `xs` and insert the element `x` at the right position in the result. Sorting an empty list will yield the empty list.

    Expressed as Scala code:

    ```
    def isort(xs: List[Int]): List[Int] =
            if (xs.isEmpty) Nil
            else insert(xs.head, isort(xs.tail))
    ```

    Provide an implementation of the missing function `insert`.

13. Given the following function, which computes the length of a list:

    ```
    def length: Int = this match {
            case Nil => 0
            case x :: xs => 1 + xs.length
    }
    ```

    Design a tail-recursive version of `length`.

14. Consider a function which squares all elements of a list and returns a list with the results. Complete the following two equivalent definitions of `squareList`:

    ```
    def squareList(xs: List[Int]): List[Int] = xs match {
            case List() => ??
            case y :: ys => ??
    }
    def squareList(xs: List[Int]): List[Int] =
            xs map ??
    ```

15. Considering the following methods of class `List`:

    ```
    def filter(p: A => Boolean): List[A] = this match {
            case Nil => this
            case x :: xs => if (p(x)) x :: xs.filter(p) else xs.filter(p)
    }

    def forall(p: A => Boolean): Boolean =
            isEmpty || (p(head) && (tail forall p))

    def exists(p: A => Boolean): Boolean =
            !isEmpty && (p(head) || (tail exists p))
    ```

    Define `forall` and `exists` in terms of `filter`.

```
object ListMain extends App {
  // val xs = List(1, 2, 3, 4, 5)
  val xs = List(1, 2, 3, 4, 5, 0)

  def forall[A](p: A => Boolean)(xs: List[A]): Boolean = {
    def filter[A](p: A => Boolean)(xs: List[A]): List[A] = xs match {
      case Nil => Nil
      case y :: ys =>
        if (p(y)) y :: ys.filter(p)
        else Nil
    }

    // filter(p)(xs).length > 0
    filter(p)(xs).length == xs.length
  }

  def exists[A](p: A => Boolean)(xs: List[A]): Boolean = {
    def filter[A](p: A => Boolean)(xs: List[A]): List[A] = xs match {
      case Nil => xs
      case y :: ys =>
        if (p(y)) List(y) // found; no need for further recursion
        else ys.filter(p)
    }

    filter(p)(xs).length > 0
  }

  assert(exists[Int](x => x > 3)(xs))
  assert(!exists[Int](x => x < 0)(xs))

  // assert(forall[Int](x => x > 0)(xs))
  assert(forall[Int](x => x >= 0)(xs))
  assert(!forall[Int](x => x > 3)(xs))
}
```

16. Consider the problem of writing a function `flatten`, which takes a list of element lists as arguments. The result of `flatten` should be the concatenation of all element lists into a single list. Here is an implementation of this method in terms of `:\`:

    ```
    def flatten[A](xs: List[List[A]]): List[A] =
            (xs :\ (Nil: List[A])) {(x, xs) => x ::: xs}
    ```

    Consider replacing the body of `flatten` by

    ```
    ((Nil: List[A]) /: xs) ((xs, x) => xs ::: x)
    ```

17. Complete the missing expressions in the following definitions of some basic list-manipulation operations as *fold* operations.

```
def mapFun[A, B](xs: List[A], f: A => B): List[B] =
        (xs :\ List[B]()){ ?? }

def lengthFun[A](xs: List[A]): Int =
        (0 /: xs){ ?? }
```

18. Given a chess board of size `n`, place `n` queens on it so that no queen is in check with another. Consider the following code:

```
def queens(n: Int): List[List[Int]] = {
  def placeQueens(k: Int): List[List[Int]] =
    if (k == 0) List(List())
    else for {queens <- placeQueens(k - 1)
              column <- List.range(1, n + 1)
              if isSafe(column, queens, 1)} yield column :: queens
  placeQueens(n)
}
```

    Write the function

```
def isSafe(col: Int, queens: List[Int], delta: Int): Boolean
```

    which tests whether a queen in the given column `col` is *safe* with respect to the queens already placed. Here, `delta` is the difference between the row of the queen to be placed and the row of the first queen in the list.

19. Define the following function in terms of `for`:

```
def flatten[A](xss: List[List[A]]): List[A] =
  (xss :\ (Nil: List[A])) ((xs, ys) => xs ::: ys)
```

Higher-Order functions

20. Translate

```
for (b <- books; a <- b.authors if a startsWith "Bird") yield b.title
for (b <- books if (b.title indexOf "Program") >= 0) yield b.title
```

    to higher-order functions.

21. (a) Given the following sum type for modelling optional data:

```
sealed trait Maybe[A]
final case class Full[A](value: A) extends Maybe[A]
final case class Empty[A]() extends Maybe[A]
```

        Implement `fold` for `Maybe`.

    (b) Given the following generic sum type:

```
sealed trait Sum[A, B]
final case class Left[A, B](value: A) extends Sum[A, B]
final case class Right[A, B](value: B) extends Sum[A, B]
```

Implement `fold` for `Sum`.

22. A binary tree can be defined as follows:

    A Tree of type A is a Node with a left and right Tree or a Leaf with an element of type A.

    (a) Implement this algebraic data type along with a `fold` method.

    (b) Using `fold` convert the following `Tree` to a `String`.

```
val tree: Tree[String] =
  Node(Node(Leaf("To"), Leaf("iterate")),
    Node(Node(Leaf("is"), Leaf("human,")),
      Node(Leaf("to"), Node(Leaf("recurse"), Leaf("divine")))))
```