Exercise sheet 2
for Monday 7 February and Wednesday 9 February 2010

Last update 2011-01-25

You will need a C# 4.0 compiler from Microsoft or the Mono project to do the first half of the exercises, and a Scala 2.8 compiler (which in turn requires a Java runtime) for the second half of the exercises.

Hand in your solutions by sending a single zip file to sestoft@itu.dk by Friday 18 February 2011. The file name must be **`SASP-02-yourname.zip`**; for instance `SASP-02-OleHansen.zip`.

Troubleshooting

Problem: Running your compiled programs, such as `Foo.exe`, generates an exception with a long complicated message involving something about security.

Diagnosis: Windows does not allow you to execute files from a network drive.

Solution (quick and dirty): Copy the `.exe` file to the local disk (C:) and run it there.

Solution (proper, VS2010): Go to Start > Control Panel > Administrative Tools > Microsoft .NET Framework 2.0 Configuration > Configure Code Access Security Policy > Adjust Zone Security > Make changes to this computer > Local Intranet > 3/4 trust. Beware that later versions of Microsoft .NET may have a different but equally unfathomable permutation of these steps.

Problem: The prompt claims that `csc` is an unknown program.

Diagnosis: You're using a Command Prompt instead of .NET Command Prompt.

Solution: Use Start > All Programs > Microsoft Visual Studio 2010 > Visual Studio Tools > Visual Studio Command Prompt.

Problem: The Func and Action types, or the Linq extension methods, are mysteriously unavailable.

Diagnosis: You're using a development environment (maybe Visual C# Express, Monodevelop) that does not reference these by default.

Solution: Add references to System.Core and System.Data.Linq to you project.

Nullables and out parameters

Exercise 2.1 Define a C# method `int? ParseInt(String s)` that attempts to parse an integer from string `s`, and returns that integer if it succeeds, and returns `null` otherwise. Note that the return type is "nullable `int`". Hint: Use method `bool System.Int32.TryParse(String s, out int result)`. You might instead use `System.Int32.Parse(String s)` and handle the exception when parsing fails, but that would be horribly slow.

Higher-order functions on enumerables

Exercise 2.2 Do the optional `Flatten` exercise from Exercise sheet 1.

Exercise 2.3 Using the same regular expression as in the `SplitLines` method in Exercise sheet 1, define a C# method:

```
static IEnumerable<String> Words(String s) { ... }
```

that splits a string `s` into a stream of "words".

Then define method `SplitLines` using only methods `Words`, `Map` (from the lecture) and `Flatten`.

Exercise 2.4 Use the `ReadFile` method from Exercise sheet 1, and the new `SplitLines` method, and the `ParseInt` method from Exercise 2.1, and write a loop to compute the sum of those "words" from a text file that can be parsed as integers (ignoring everything that cannot be parsed as an integer).

Exercise 2.5 Write a method `IEnumerable<T> Merge<T>(IEnumerable<T> xs, IEnumerable<T> ys)` that merges two sorted enumerables, without duplicates, to produce a sorted enumerable without duplicates. There should be a constraint on type parameter `T` requiring it to support comparison. The merge should be lazy, that is, should not build any intermediate arrays, lists, hashsets or similar. (Hint: This is surprisingly cumbersome; I cannot find a clean solution in less than roughly 30 lines).

Use your `Merge` method to merge the three sorted streams of multiples of 2, multiples of 3, and multiples of 5, into a single sorted sequence — this requires two applications of `Merge<int>`.

Hint: Method `System.Linq.Enumerable.Range(i,j)` creates a stream of the integers from `i` to `j` (inclusive).

Linq

Exercise 2.6 Using the `ReadFile` method from Exercise sheet 1, which creates a stream of the lines of a file on disk, write Linq queries to compute the following:

- Compute a stream (that is, an enumerable) of the lengths of the lines of a file.
- Compute the average length of the lines of a file.
- Compute the maximal length of the lines of a file.
- Compute the average length of the non-blank lines of a file read from disk.
- Compute the number of lines whose length is between 40 and 72 characters (inclusive).
- Compute, for each line length in the file, the number of lines that have that length.
- Compute, for each line length between 40 and 72 characters (inclusive), the number of lines that have that length.

Scala

Exercise 2.7 Define Scala polymorphic (generic) higher-order functions `map`, `filter` and `flatten` on type List[T].

Exercise 2.8 The lecture used the following enumeration type and case class hierarchy to represent simple arithmetic expressions:

```
object Operator extends Enumeration {
  val Add, Sub, Mul, Div, Eq, Ne, Lt, Le, Gt, Ge = Value
}

abstract class Expression
case class Variable(name : String) extends Expression
case class Constant(value : Int) extends Expression
case class BinOp(op : Operator.Value, e1 : Expression, e2 : Expression) extends Expression
```

Write a Scala function `reduce(expr : Expression) : Expression` that uses pattern matching to reduce arithmetic expressions. For instance, `BinOp(Operator.Add, Constant(0), e2)` should be reduced to `e2`. Namely, that expression represents `0+e2` which is equivalent to `e2`.

Your `reduce` function should perform reductions at least corresponding to these cases:

$0 + e$	$\longrightarrow$	e
$e + 0$	$\longrightarrow$	e
$e - 0$	$\longrightarrow$	e
$1 * e$	$\longrightarrow$	e
$e * 1$	$\longrightarrow$	e
$0 * e$	$\longrightarrow$	0
$e * 0$	$\longrightarrow$	0
$e - e$	$\longrightarrow$	0

Exercise 2.9 Use Scala `for`-expressions to compute some of the same values you did with C# Linq in Exercise 2.6 above.

Hint 1: You can get the lines of a file as an Iterator[String] using this function:

```
def readFile(name : String) = scala.io.Source.fromFile(name).getLines
```

Note that the lines include a terminating newline, so an "empty" line contains at least the character `\n` (and on Windows possible also `\n` more stuff).

Hint 2: The Scala library does not seem to offer aggregate functions, so you will need to define functions like these:

```
def average(xs : Iterator[Int]) : Double = { ... }
def max(xs : Iterator[Int]) : Int = { ... }
def groupBy[T,K](xs : Seq[T], f : T => K) : Set[(K, Seq[T])] = { ... }
```

Hint 3: The `groupBy` can be computed efficiently by using a `Map[T,Seq[K]]`, and it is perfectly OK to return the result as such a map rather than a `Set[(K, Seq[T])]`, but not a requirement.

Hint 4: If you are getting into too much trouble with type mismatches between Iterator and Iterable, then you may find it useful that `xs.toList` turns an Iterator `xs` into a List which supports Iterable. The downside is that the lazy Iterator (which reads from a file as needed) gets turned into an in-memory list of all the file's lines. Conversely, if `xs` is an Iterable, then `xs.elements` is an Iterator. Perhaps there are neater ways out of this, which we will discover as we become more experienced with Scala.