1 Support for command line arguments

All the different command line tools produced (`tangle`, `weave`, the one-step-compiler) have some things in common: They take as input either markup files or literate files. Also, it would be quite useful to specify more than one of them, for example to create a woven document out of several input files. The class `LiterateSettings` will provide exactly this functionality. It is not as evolved as the compiler settings and will just remember arguments that it does not know about.

⟨ ** ⟩ ≡

```
package scalit.util
import scalit.markup._

object LiterateSettings {
  <getting the arguments>
}

class LiterateSettings(val settings: Map[String,List[String]],
                                     ls: List[Stream[Line]]) {
  <Constructor with argument list>

  <a reference to output>
  <getting the filters>
  <getting the lines>
  <getting the blocks>
  <getting the chunks>
}
```

We have two fields here: One is for all the settings that we got and the other is for the input files. But we won't really use this constructor: We'd rather directly take the arguments given to the application.

1.1 Parsing the command line arguments

The usual way to call `LiterateSettings` is with an argument list in form of an array of strings. Inside this constructor, we'll obtain the value for settings and lines with a call to a recursive function.

⟨ *Constructor with argument list* ⟩ ≡

```
def this(p: (Map[String,List[String]],List[Stream[Line]])) =
  this(p._1, p._2)
def this(args: Array[String]) =
  this(LiterateSettings.getArgs(args.toList,Map(),Nil))
```

Now to get the arguments, we go element for element through the list, trying to obtain something: This function has to be defined outside of the class `LiterateSettings`, because it will be called before the object exists.

⟨*getting the arguments*⟩ ≡

```
def getArgs(args: List[String], settings: Map[String,List[String]],
                lines: List[Stream[Line]]) :
    (Map[String,List[String]],List[Stream[Line]]) = args match {
```

If any filename is prefixed by an argument consisting of `-m`, then we parse some lines from markup input:

⟨*getting the arguments*⟩ + ≡

```
case "-m" :: markupfile :: xs ⇒ {
  val mlines = conversions.linesFromMarkupFile(markupfile)
  getArgs(xs,settings,mlines ::: lines)
}
```

We might also read from standard input. With the command line option of `-li`, we try to read a literate program, with `-mi`, we try to read markup input:

⟨*getting the arguments*⟩ + ≡

```
case "-li" :: Nil ⇒ {
  val llines = conversions.linesFromLiterateInput(System.in)
  (settings, lines.reverse ::: List(llines))
}
case "-mi" :: Nil ⇒ {
  val mlines = conversions.linesFromMarkupInput(System.in)
  (settings,lines.reverse ::: List(mlines))
}
```

1.1.1 Additional command line arguments

If the frontmost element of `args` begins with `-`, that means we are dealing with an option that takes one argument (all the other cases were dealt with before). We append it to the list of arguments already given with this option

⟨*getting the arguments*⟩ + ≡

```
case opt :: arg :: xs ⇒
  if( opt(0) == '−' )
    getArgs(xs,settings +
                  (opt → (arg :: settings.getOrElse(opt,Nil))),
                lines)
  else {
    val llines = conversions.linesFromLiterateFile(opt)
    getArgs(arg :: xs, settings, llines :: lines)
  }
```

We have to treat the case where we get ``‘‘-o’’`` specially: in this case, we'll have to provide output to a file:

⟨*a reference to output*⟩ ≡

```
lazy val output: java.io.PrintStream =
  settings get "−o" match {
    case None ⇒ System.out
    case Some(List(file)) ⇒ new java.io.PrintStream(
      new java.io.FileOutputStream(file)
    )
  }
```

Finally, if there is only one argument left, then it has to be an input from a literate file, otherwise it would have been treated:

⟨*getting the arguments*⟩ + ≡

```
case litfile :: xs ⇒ {
  val llines = conversions.linesFromLiterateFile(litfile)
  getArgs(xs,settings,llines :: lines)
}
case Nil ⇒ (settings,lines.reverse)
}
```

1.2 Filters

We can also, on the command line specify filters to be applied to the markup or block phase. Filters are just classes extending from `MarkupFilter` or `BlockFilter`. The following two fields hold reference to these filters:

⟨*getting the filters*⟩ ≡

```
import scalit.util.{MarkupFilter,BlockFilter}
val markupFilters: List[MarkupFilter] =
  settings get ("-lfilter") match {
    case None ⇒ Nil
    case Some(xs) ⇒ {
```

If we have some names, then we will try to load them using reflection, creating an instance for each class.

⟨*getting the filters*⟩ + ≡

```
xs map {
  name ⇒
    try {
      val filterClass = Class.forName(name)
      filterClass.newInstance.asInstanceOf[MarkupFilter]
    } catch {
      case ex ⇒
        Console.err.println("Could not load filter " + name)
        System.exit(1)
```

Somehow, System.exit is not enough for the type checker, therefore we will have to give back some dummy class if that happens.

⟨*getting the filters*⟩ + ≡

```
                new util.tee
            }
          }
        }
      }
```

With block filters, it's exactly the same story:

⟨*getting the filters*⟩ + ≡

```
    lazy val blockFilters: List[BlockFilter] =
      settings get "-bfilter" match {
        case None ⇒ Nil
        case Some(xs) ⇒ xs map {
          name ⇒
            try {
              val filterClass = Class.forName(name)
              filterClass.newInstance.asInstanceOf[BlockFilter]
            } catch {
              case e ⇒
                Console.err.println("Could not load" +
                " block filter " + name)
                System.exit(1)
                new util.stats
            }
        }
      }
```

1.3 Getting the content of the lines in different formats

With the settings in place, it is very easy to get the actual content of the files in different formats. We will just have to take care of the filters:

⟨*getting the lines*⟩ ≡

```
lazy val lines: List[Stream[Line]] = ls map {
  markupStream: Stream[Line] ⇒ markupStream
    (markupFilters foldLeft markupStream) {
      (acc: Stream[Line], f: MarkupFilter) ⇒ f(acc)
    }
}
```

To build the blocks, we'll just have to instantiate a block builder for every stream of lines. Also, we'll have to apply the filters in order, of course.

⟨*getting the blocks*⟩ ≡

```
val blocks: List[(Stream[markup.Block],String)] = lines map {
  l ⇒ {
    val bb = BlockBuilder(l)
    val filteredBlocks: Stream[Block] =
      (blockFilters foldLeft bb.blocks) {
        (acc: Stream[Block],f: BlockFilter) ⇒ f(acc)
      }
    (filteredBlocks,bb.filename)
  }
}
```

The chunk collections work in a very similar way.

⟨*getting the chunks*⟩ ≡

```
import scalit.tangle.emptyChunkCollection
lazy val chunkCollections = blocks map {
  case (bs,name) ⇒
  emptyChunkCollection(name) addBlocks conversions.codeblocks(bs)
}
```