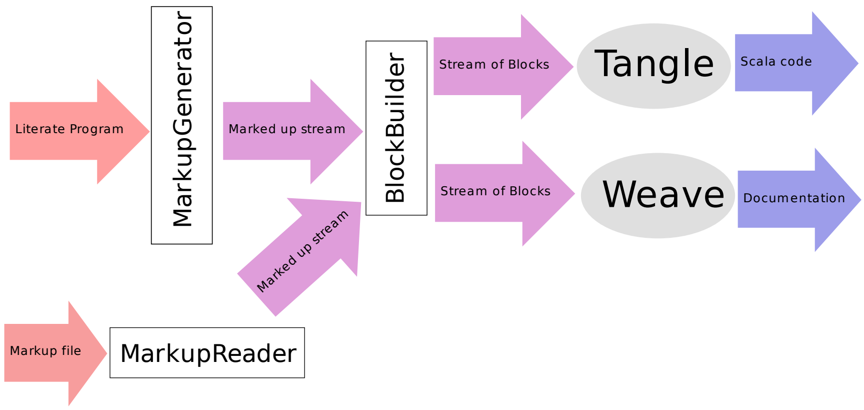
1 The noweb markup format

While the final goal of the literate programming tools is to extract the source and documentation from a `noweb` file, we will not work with the raw input source, but with an intermediary format called `markup`, where the function of each line is explicitly noted. The following classes will convert a `noweb` file into a marked up file.

After we have converted the input from this format (or after we have directly read the intermediate format), another stage will convert our stream of lines in a stream of blocks. Only this format will then be used for `tangle` which generates the code and `weave` which generates the documentation.

The next graphic illustrates the stages that we will go through:



1.1 Outline

The Noweb markup that we will use is very light and only consists of very few different cases, however, they all interact, so the conversion part is still quite difficult. A noweb file will contain:

- source chunks (whose beginning is indicated by an @ sign)
- quoted text (indicated by `[[` and `]]`)
- code chunks between ⟨⟨ and ⟩⟩`=`
- use directives (also between ⟨⟨ and ⟩⟩, but without an = afterwards

More details of the format can be found in the file `markup.nw` in the noweb distribution[1]

[1]Noweb home page: http://www.eecs.harvard.edu/nr/noweb/

This will then be converted in the intermediary format. To get a rough feeling for the intermediary format, we will first treat this format and write the conversion classes to this intermediary format afterwards:

⟨*⟩ ≡

```
package scalit.markup
<line types>
<read in the intermediary format>
<markup reader>
<conversion from noweb to markup>
<markup generator>
```

1.2 The line types of the intermediary format

There are about a dozen different line types. Let us first define these line types and then consider how to read in a file in this format.

⟨*line types*⟩ ≡

```
abstract class Line
```

Now for the subclasses

A line of text Might occur inside documentation chunks but also inside code chunks.

⟨*line types*⟩ + ≡

```
case class TextLine(content: String) extends Line {
  override def toString = "@text " + content
}
```

A new line Because we will be splitting up one line in different lines describing the structure of the file, we will have to indicate when a newline character occured in the source file. This is done with the following class:

⟨*line types*⟩ + ≡

```
case object NewLine extends Line {
  override def toString = "@nl"
}
```

Quotes Section of the text will be quoted and will appear verbatim in the documentation. They are situated betwen beginning and end quote tokens:

⟨*line types*⟩ +≡

```
case object Quote extends Line {
  override def toString = "@quote"
}
case object EndQuote extends Line {
  override def toString = "@endquote"
}
```

Code chunks Like quotes, we will designate code chunks by their beginning and end. Another important information will be the *number* of this chunk: Code and documentation chunks are numbered consecutively.

⟨*line types*⟩ +≡

```
case class Code(number: Int) extends Line {
  override def toString = "@begin code " + number
}
case class EndCode(number: Int) extends Line {
  override def toString = "@end code " + number
}
```

As already mentioned, code chunks will be given names so that they can be used as part of other code chunks. This information will be the first line inside a code section:

⟨*line types*⟩ +≡

```
case class Definition(chunkname: String) extends Line {
  override def toString = "@defn " + chunkname
}
```

Now that we have a line indicating which code chunk is defined, we can also envisage to use it in other code chunks:

⟨*line types*⟩ +≡

```
case class Use(chunkname: String) extends Line {
  override def toString = "@use " + chunkname
}
```

Documentation Like code chunks, documentation chunks will be given a number. Otherwise, they behave the same:

⟨*line types*⟩ +≡

```
case class Doc(number: Int) extends Line {
  override def toString = "@begin docs " + number
}
case class EndDoc(number: Int) extends Line {
  override def toString = "@end docs " + number
}
```

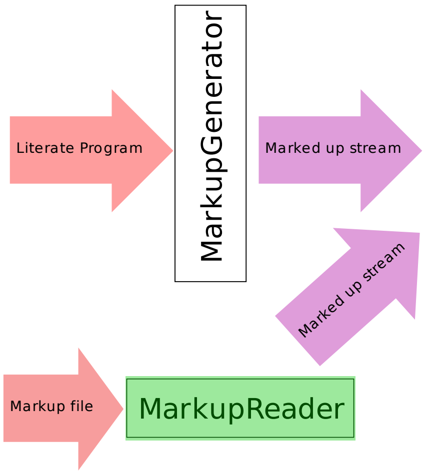
Special indicators We will have two special indicators: One telling us which file we are reading and one to indicate the end of our input.

⟨*line types*⟩ +≡

```
case class File(filename: String) extends Line {
  override def toString = "@file " + filename
}
case object LastLine extends Line
```

To test whether the intermediary format was completely described, we will build a small parser for it in the next section.

1.3 Reading in the intermediary format



A parser for the intermediary format will be much simpler than the one we will have to build for the noweb source format: It suffices to parse the beginning of a line to get the function. This simple parser will consist of two methods, one that gets the next token and one that represents all the tokens of a file as a stream.

⟨*read in the intermediary format*⟩ ≡

```
import scala.util.parsing.input.StreamReader
class MarkupParser(in: StreamReader) {
    <get next token of markup file>
    <token stream of markup file>
}
```

Note that the MarkupParser takes a stream reader representation of the input file as constructor argument. With a StreamReader, we have the possibility to record our position in the file while reading.

Let us first look at the stream representation: We know we have read the last token when we see `LastLine`

⟨*token stream of markup file*⟩ ≡

```
def lines: Stream[Line] = {
  def lines0(input: StreamReader): Stream[Line] =
  nextToken(input) match {
    case (LastLine,rest) ⇒ Stream.empty
    case (line,rest) ⇒ Stream.cons(line, lines0(rest))
  }
  lines0(in)
}
```

After we have read the token, we will have advanced in the stream, therefore we will read in the next token from the returned position.

This stream representation will be quite useful when we are processing the input with tangle and weave. But let us move on to the method that actually gives us the token:

⟨*get next token of markup file*⟩ ≡

```
def nextToken(input: StreamReader): (Line,StreamReader) = {
  def readLine(inreader: StreamReader,
               acc: List[Char]): (List[Char],StreamReader) =
    if( inreader.atEnd || inreader.first == '\n' )
      (inreader.first :: acc.reverse, inreader.rest )
    else readLine(inreader.rest, inreader.first :: acc )
```

This little function reads in the whole line and returns the position afterwards.

⟨*get next token of markup file*⟩ + ≡

```
    in.first match {
      case '@' ⇒ {
        val (line,rest) = readLine(input.rest,Nil)
        val directive = <match on the directive to produce the line>
        (directive,rest)
      }
      case _ ⇒
        System.err.println("Found a line not beginning" +
                           " with @, but with " +
                           input.first)
        exit(1)
    }
  }
```

When having read a line, we take the directive which is everything before the first space character. So we split along the spaces. However, strangely enough the readLine function (and therefore StreamReader) seems to insert a strange character at the beginning (this might as well be a bug), so we will have to continue after that.

⟨*match on the directive to produce the line*⟩ ≡

```
  (line.tail mkString "" split ' ').toList match {
    case "file" :: sth ⇒ File(sth mkString " ")
```

The file line has one more argument: The filename. We thus simply reconstruct it from the argument list.

After the file token, we will usually find a beginning of either a code or a documentation chunk. The only argument this takes is the chunk number.

⟨*match on the directive to produce the line*⟩ + ≡

```
    case "begin" :: "docs" :: number :: Nil ⇒
      Doc(Integer.parseInt(number))
    case "begin" :: "code" :: number :: Nil ⇒
      Code(Integer.parseInt(number))
```

The beginning directive will be matched by the same end directive:

⟨*match on the directive to produce the line*⟩ + ≡

```
case "end" :: "docs" :: number :: Nil =>
  EndDoc(Integer.parseInt(number))
case "end" :: "code" :: number :: Nil =>
  EndCode(Integer.parseInt(number))
```

Inside this documentation block, we may find text parts and newlines, primarily, which we will match in the same way. A quick note on the text: This is whitespace sensitive (the line will quite likely end in whitespace), therefore we do pass it the line and not the split.

⟨*match on the directive to produce the line*⟩ +≡

```
case "text" :: content =>
  TextLine(line drop 6 mkString "")
case "nl" :: Nil =>
  NewLine
```

Inside code chunks, we will also find indications on how the chunk is called:

⟨*match on the directive to produce the line*⟩ +≡

```
case "defn" :: chunkname => Definition(chunkname mkString " ")
```

Use directives work the same way:

⟨*match on the directive to produce the line*⟩ +≡

```
case "use" :: chunkname => Use(chunkname mkString " ")
```

Quote and EndQuote do not take any parameters:

⟨*match on the directive to produce the line*⟩ +≡

```
case "quote" :: Nil => Quote
case "endquote" :: Nil => EndQuote
```

If we see an empty string, then we will have arrived at the last line. Otherwise, do mark the token as unrecognized.

⟨*match on the directive to produce the line*⟩ +≡

```
    case "" :: Nil ⇒ LastLine
    case unrecognized ⇒ {
      System.err.println("Unrecognized directive: " +
                         unrecognized.head.size)
      println(input.pos)
      LastLine
    }
  }
```

1.3.1 Testing the markup format

Like the Markup generator, we also want to test the markup reader. We will output what we have read in to compare it to the original file: If we get the same result, then we are fine.

⟨*markup reader*⟩ ≡

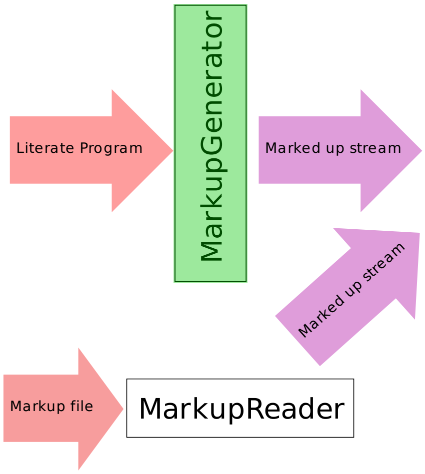
```
object MarkupReader {
  import java.io.{FileReader,BufferedReader,Reader}
  import java.io.InputStreamReader

  def usage =
    println("Usage: scala markup.MarkupReader [infile]")

  def main(args: Array[String]) = {
    val input: Reader = args.length match {
      case 0 ⇒ new InputStreamReader(System.in)
      case 1 ⇒ new BufferedReader(new FileReader(args(0)))
      case _ ⇒ usage; exit
    }
    val markupReader = new MarkupParser(StreamReader(input))
    markupReader.lines foreach {
      line ⇒ println(line)
    }
  }
}
```

Very basic: We either get a file name as argument or we treat standard input. For this test, we will just print the lines stream.

1.4 Converting from the input



Now that we know enough about the intermediary format, we are ready to treat the conversion. The final product will be a stream of markup lines (like we used them above). The first token will be the filename:

⟨*conversion from noweb to markup*⟩ ≡

```
class MarkupGenerator(in: StreamReader, filename: String) {
  def lines: Stream[Line] =
    Stream.cons(File(filename),
    Stream.cons(Doc(0),documentation(in,0)))
  <in documentation mode>
  <in quote mode>
  <in code mode>

  <read chunk name>
  <read use name>

  <tab handling>
}
```

The second token is also already given: The start of the first documentation block. A noweb file will always start with a block like this. The next section defines how the documentation mode works.

1.4.1 The documentation mode

The documentation mode will accumulate TextLines and NewLines as long as we do not see a combination that would terminate this documentation chunk:

- Another documentation chunk (at sign on a new line)
- A quoted section
- Beginning of a code block

To memorize the line content, we will use an accumulator.

⟨*in documentation mode*⟩ ≡

```
def documentation(inp: StreamReader,
                  docnumber: Int): Stream[Line] = {
  def docAcc(input: StreamReader,
             acc: List[Char]): Stream[Line] =
    <accumulate in doc mode>

  docAcc(inp,Nil)
}
```

Now let us look at how to produce some lines:

⟨*accumulate in doc mode*⟩ ≡

```
input.first match {
  case '[' ⇒ input.rest.first match {
    case '[' ⇒
      val (content,continue) = quote(input.rest.rest)
      acc match {
        case Nil ⇒
          Stream.concat(
            Stream.cons(Quote,content),
            Stream.cons(EndQuote,
                        docAcc(continue,Nil)))
        case _  ⇒
          Stream.concat(
            Stream.cons(TextLine(acc.reverse mkString ""),
            Stream.cons(Quote,
                        content)),
            Stream.cons(EndQuote,
                        docAcc(continue,Nil)))
      }
  case _ ⇒ docAcc(input.rest, input.first :: acc)
}
```

If we encounter the beginning of a quoted section, then we will call the quote mode just to continue parsing afterwards. If we encounter the at sign at the beginning of a line, we will start a new documentation chunk:

⟨*accumulate in doc mode*⟩ + ≡

```
case '@' ⇒ acc match {
  case Nil ⇒
    if( input.rest.first == '\n' ||
                  input.rest.first == ' ' )
      Stream.cons(EndDoc(docnumber),
      Stream.cons(Doc(docnumber + 1),
                  documentation(input.rest.rest,docnumber + 1)))
    else
      docAcc(input.rest, List('@'))
  case _ ⇒ docAcc(input.rest, '@' :: acc)
}
```

A documentation section can also be terminated by the beginning of a code chunk. This chunk will be between ⟨⟨ and ⟩⟩=. If a code chunk seems to be opened but a newline follows before it was closed, we have to report this error:

⟨*accumulate in doc mode*⟩ + ≡

```
case '<' => input.rest.first match {
  case '<' => acc match {
    case x :: xs =>
          error("Unescaped << in doc mode")
    case Nil =>
        val (chunkName,continue) = chunkDef(input.rest.rest)
        Stream.cons(EndDoc(docnumber),
        Stream.cons(Code(docnumber + 1),
          code(continue,chunkName,docnumber+1)))
  }
  case _ => docAcc(input.rest, '<' :: acc)
}
```

If we were able to read a chunk name, we will open a new code section with this information.

⟨*accumulate in doc mode*⟩ + ≡

```
case c =>
  if( c == '\n' ) {
    Stream.cons(TextLine(acc.reverse mkString ""),
    Stream.cons(NewLine,docAcc(input.rest,Nil)))
  } else {
    if( !input.atEnd ) docAcc(input.rest,input.first :: acc)
    else acc match {
      case Nil => Stream.cons(EndDoc(docnumber),
                              Stream.empty)
      case _ =>
        Stream.cons(TextLine(acc.reverse mkString ""),
        Stream.cons(NewLine,
        Stream.cons(EndDoc(docnumber),Stream.empty)))
    }
  }
}
```

As a general rule, all markup files will have a newline at the end. If no newline is there, then we will add one.

1.4.2 The quote mode

In quote mode, we will ignore all normal control characters up until the point where we encounter the close quote `]]`. Note, however, that additional `]`s have to be taken into account: The quote is only closed with the last `]` in a row.

⟨*in quote mode*⟩ ≡

```
def quote(inp: StreamReader): (Stream[Line], StreamReader) = {
  def quoteAcc(input: StreamReader, acc: List[Char]):
      (Stream[Line], StreamReader) =
 input.first match {
    case ']' ⇒ input.rest.first match {
      case ']' ⇒ input.rest.rest.first match {
        case ']' ⇒ quoteAcc(input.rest,']' :: acc)
        case _ ⇒ acc match {
          case Nil ⇒ (Stream.empty,input.rest.rest)
          case _ ⇒ (Stream.cons(TextLine(acc.reverse mkString ""),
                                            Stream.empty),
                         input.rest.rest)
        }
      }
      case _ ⇒ quoteAcc(input.rest,']' :: acc)
    }
    case '\n' ⇒ acc match {
      case Nil ⇒ val (more,contreader) = quoteAcc(input.rest,Nil)
                    (Stream.cons(NewLine,more),contreader)
      case _ ⇒ val (more,contreader) = quoteAcc(input.rest,Nil)
                  (Stream.cons(TextLine(acc.reverse mkString ""),
                  Stream.cons(NewLine, more)),contreader)
    }
    case c ⇒ quoteAcc(input.rest, c :: acc)
 }
        quoteAcc(inp,Nil)
}
```

1.4.3 The code mode

Code chunks are a bit different from documentation chunks in the fact that they are named. The following method reads the name of a documentation chunk and returns it:

⟨*read chunk name*⟩ ≡

```
def chunkDef(inp: StreamReader): (String, StreamReader) = {
  def chunkAcc(input: StreamReader, acc: List[Char]):
  (String, StreamReader) =
  input.first match {
    case '>' => input.rest.first match {
      case '>' => input.rest.rest.first match {
      case '=' => ((acc.reverse mkString ""),input.rest.rest.rest)
      case _ => System.err.println("Unescaped"); exit
    }
    case _ => chunkAcc(input.rest, '>' :: acc)
  }
  case c => chunkAcc(input.rest, c :: acc)
  }

  chunkAcc(inp, Nil)
}
```

Now that we know how we can read in the chunk name, let us look on how to read code sections:

⟨*in code mode*⟩ ≡

```
def code(inp: StreamReader, chunkname: String, codenumber: Int):
  Stream[Line] = {
   <detect new code chunk>
   <detect new use directive>
  def codeAcc(input: StreamReader, acc: List[Char]):
    Stream[Line] = input.first match {
      case '<' => input.rest.first match {
        case '<' =>
          acc match {
            case Nil =>
            if( isNewCodeChunk(input.rest.rest) ) {
              val (chunkName,continue) =
                chunkDef(input.rest.rest)
              Stream.cons(EndCode(codenumber),
              Stream.cons(Code(codenumber + 1),
                code(continue,
                chunkName,
                codenumber + 1)))
            } else if( isNewUseDirective(input.rest) ) {
              val (usename,cont) = use(input.rest.rest)
              Stream.cons(Use(usename),
                  codeAcc(cont,Nil))
            } else {
              codeAcc(input.rest,'<' :: acc)
            }
```

If we are not at the beginning of a line (`acc` is not empty), then we don't have to worry about new code chunks, just use directives:

⟨*in code mode*⟩ + ≡

```
          case _ ⇒
          if( isNewUseDirective(input.rest) ) {
            val (usename,cont) = use(input.rest.rest)
            Stream.cons(TextLine(acc.reverse mkString ""),
            Stream.cons(Use(usename),
              codeAcc(cont, Nil)))
          } else {
            codeAcc(input.rest, '<' :: acc)
          }
        }
        case _ ⇒ codeAcc(input.rest, '<' :: acc)
      }
```

In a code section, we might also encounter ⟨⟨. But here, it might either be the beginning of a new code section or a use directive: Unfortunately, we can't know without scanning ahead, so we use our little utility function `isNewCodeChunk`:

⟨*detect new code chunk*⟩ ≡

```
def isNewCodeChunk(input: StreamReader): Boolean =
  input.first match {
    case '>' ⇒ input.rest.first match {
      case '>' ⇒ input.rest.rest.first match {
        case '=' ⇒ true
        case _ ⇒ false
      }
      case _ ⇒ isNewCodeChunk(input.rest)
    }
    case c ⇒
      if( c == '\n' )
        false
      else isNewCodeChunk(input.rest)
}
```

To detect whether we are treating a new use directive is very similiar:

⟨*detect new use directive*⟩ ≡

```
def isNewUseDirective(input: StreamReader): Boolean =
input.first match {
    case '>' ⇒ input.rest.first match {
        case '>' ⇒ input.rest.rest.first match {
            case '=' ⇒ false
            case _   ⇒ true
        }
        case _ ⇒ isNewUseDirective(input.rest)
    }
    case c ⇒
        if( c == '\n' ) false
        else isNewUseDirective(input.rest)
}
```

This can tell us whether we really have a new code chunk before us, but not whether we really have a use directive.

A code block is also finished upon seeing an at sign:

⟨*in code mode*⟩ + ≡

```
case '@' ⇒
    if( input.rest.first == ' ' ||
        input.rest.first == '\n' )
      acc match {
          case Nil ⇒ Stream.cons(EndCode(codenumber),
                            Stream.cons(Doc(codenumber + 1),
                            documentation(input.rest.rest,
                                          codenumber + 1)))
          case _ ⇒ codeAcc(input.rest, '@' :: acc)
      }
    else codeAcc(input.rest, '@' :: acc)
```

If none of these special cases occurred, we can simply continue parsing lines.

⟨*in code mode*⟩ + ≡

```
    case c ⇒
      if( c == '\n' ) {
        val tl = TextLine(acc.reverse mkString "")
        Stream.cons(tl,
        Stream.cons(NewLine,codeAcc(input.rest,Nil)))
```

This newline thing is quite peculiar: If we encounter two newlines without any text in between, we will still interleave an empty text node.

⟨*in code mode*⟩ + ≡

```
      } else {
        if( input.atEnd )
          acc match {
            case Nil ⇒ Stream.cons(EndCode(codenumber),
                                   Stream.empty)
            case _ ⇒
              Stream.cons( TextLine(acc.reverse mkString ""),
              Stream.cons(NewLine,
              Stream.cons(EndCode(codenumber),Stream.empty)))
          }
        else if( c == '\t' )
          codeAcc(input.rest,tab ::: acc )
        else
          codeAcc(input.rest,c :: acc )
      }
    }

    Stream.cons(Definition(chunkname),
    Stream.cons(NewLine,
            codeAcc(inp.rest,Nil)))
    }
```

A little strangeness comes from the tab character: markup expands this character to eight spaces, so we'll do that too:

⟨*tab handling*⟩ ≡

```
      val tab = (1 to 8 map { x ⇒ ' ' }).toList
```

1.4.4 Reading a use name

There is still a small utility function missing: the one to read which chunk to use:

⟨*read use name*⟩ ≡

```
def use(inp: StreamReader): (String, StreamReader) = {
  def useAcc(input: StreamReader, acc: List[Char]):
        (String,StreamReader) = input.first match {
    case '>' => input.rest.first match {
      case '>' => (acc.reverse mkString "",input.rest.rest)
      case _ => useAcc(input.rest, '>' :: acc)
    }
    case c => useAcc(input.rest, c :: acc)
  }
  useAcc(inp,Nil)
}
```

1.5 The command line application

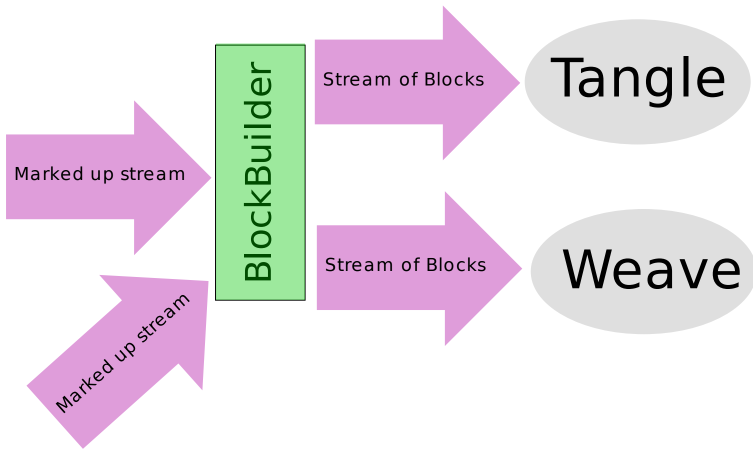
As we have seen, the data structures produced during the markup step can be directly used by further stages. To gain some compatibility with noweb (and access to other filters provided by it), we can also output the intermediary format. This will be done when we call the application `Markup`. We use the literate settings defined in `util/commandline.nw`

⟨*markup generator*⟩ ≡

```
object Markup {
  def usage: Unit = {
    System.err.println("Usage: scala markup.Markup [infile]\n")
  }
  def main(args: Array[String]) = {
    import util.LiterateSettings
    val settings = new LiterateSettings(args)
    val listlines: List[Stream[Line]] = settings.lines
    listlines foreach {
      linestream => linestream foreach println
    }
  }
}
```

2 Extracting blocks from marked up files

While the markup intermediary format[2] provides a base to write all sorts of filters, both for tangle and weave we will be required to have a more high-level view of a literate program. The following classes will provide this view in form of blocks.



⟨*⟩+≡

[2]described in the file `markup/markup.nw`

```
package scalit.markup
<Combining strings and references>
<The block format>
<Build blocks>
<Test the block format>
```

2.1 The block format

The aim of the block format is to store the information associated with a block (their name, chunk number, line number) while providing easy access to the string representation their content for weave. The only thing common between code and documentation blocks are:

- Block number
- Beginning line number

⟨*The block format*⟩ ≡

```
sealed abstract class Block(blocknumber: Int, linenumber: Int,
                            content: Stream[Line]) {
  <body of the block class>
}
```

For the string representation, we run into a problem: While during tangling we want to extract references to other code blocks, this is not the case when we want to create documentation: Here we only want to see the name. Another problem arises with quoted strings (that occur in documentation blocks): Their content will be output verbatim in the documentation and deserves another treatment. The solution here is to have a stream that can contain either strings, references to other blocks or quoted strings.

⟨*Combining strings and references*⟩ ≡

```
object StringRefs {
  sealed abstract class StringRef
  case class RealString(content: String,
                        from: Int,
                        to: Int) extends StringRef
  case class QuotedString(content: String) extends StringRef
  case class BlockRef(referenced: CodeBlock) extends StringRef

  implicit def realString2string(rs: RealString) = rs.content
}
```

With these three different contents, we are able to define a method that, given a map of code blocks (for dereference) will give us a stream of `StringRef`:

⟨*body of the block class*⟩ ≡

```
import StringRefs._

def stringRefForm(codeBlocks: Map[String,CodeBlock]): Stream[StringRef]
```

2.1.1 Code blocks

With this class, we can now represent the content of code blocks. One special field is the reference to the next block: We will not know this in the beginning, but when everything is read in, it can be calculated.

Given a map of code blocks and their associated name, we can also easily give back the stream of `StringRef`s:

⟨*The block format*⟩ + ≡

```
case class CodeBlock(blocknumber: Int, linenumber: Int,
     content: Stream[Line], blockname: String)
       extends Block(blocknumber,linenumber,content) {
  import StringRefs._
  override def stringRefForm(
    codeBlocks: Map[String,CodeBlock]): Stream[StringRef] = {
```

This is done by accumulating the string as long as we do not have a reference. When a reference occurs, we terminate the current string part and intersperse a use name. In parallel, we'll have to store the offset inside the code block as to know which lines the string reference represents:

⟨*The block format*⟩ +≡

```
    def cbAcc(ls: Stream[Line], acc: String,
              begin: Int, off: Int): Stream[StringRef] =
  ls match {
    case Stream.cons(first,rest) ⇒ first match {
      case NewLine ⇒ cbAcc(rest, acc + "\n", begin, off + 1)
      case TextLine(content) ⇒
        cbAcc(rest, acc + content, begin, off)
      case Use(usename) ⇒ {
        val cb = codeBlocks get usename match {
          case Some(codeBlock) ⇒ codeBlock
          case None ⇒
            System.err.println("Did not find block " +
                               usename)
            exit(1)
        }
        Stream.cons(RealString(acc,begin,off),
        Stream.cons(BlockRef(cb),cbAcc(rest,"",off,off)))
      }
      case other ⇒ error("Unexpected line: " + other)
    }
```

We will also have to handle the case where we are finished with reading. Nothing special here.

⟨*The block format*⟩ +≡

```
    case Stream.empty ⇒ acc match {
      case "" ⇒ Stream.empty
      case s  ⇒ Stream.cons(RealString(s,begin,off),Stream.empty)
    }
  }

    cbAcc(content,"",linenumber,linenumber)
    }
  }
```

2.1.2 Documentation blocks

For documentation blocks, we do not have to take care of eventual references. However, quoted blocks will need to be identified.

⟨*The block format*⟩ + ≡

```
case class DocuBlock(blocknumber: Int, linenumber: Int,
content: Stream[Line]) extends
    Block(blocknumber,linenumber,content) {
    import StringRefs._
    override def stringRefForm(codeBlocks: Map[String,CodeBlock]):
        Stream[StringRef] = {
            srContent
        }
    <define the string content value>
}
```

Because we do not really depend on the code Blocks, we will be able to lazily initialize a value holding the whole Stream. At the moment, we'll not even store the line numbers of documentation: What for?

⟨*define the string content value*⟩ ≡

```
    lazy val srContent: Stream[StringRef] = {
        def srcAcc(ls: Stream[Line], acc: String): Stream[StringRef] =
            ls match {
                case Stream.empty ⇒
                    Stream.cons(RealString(acc,−1,−1),
                    Stream.empty)
                case Stream.cons(first,rest) ⇒ first match {
                    case NewLine ⇒ srcAcc(rest,acc + "\n")
                    case TextLine(content) ⇒ srcAcc(rest, acc + content)
```

Like in the code case, these two are relatively trivial. We will need to invoke another function for quotes.

⟨*define the string content value*⟩ + ≡

```
    case Quote => {
      val (quoted,continue) = quote(rest,"")
      Stream.cons(RealString(acc,-1,-1),
      Stream.cons(quoted,srcAcc(continue,"")))
    }
    case other => error("Unexpected line in doc: " + other)
        }
      }
        <quote accumulation>

        srcAcc(content,"")
    }
```

We still need the quote accumulation: Until the end of the quote, we will just concatenate the string and then return where to continue and the content:

⟨*quote accumulation*⟩ ≡

```
    def quote(ls: Stream[Line],
              acc: String): (QuotedString,Stream[Line]) =
      ls match {
        case Stream.empty => (QuotedString(acc),Stream.empty)
        case Stream.cons(first,rest) => first match {
          case NewLine => quote(rest, acc + "\n")
          case TextLine(content) => quote(rest, acc + content)
          case EndQuote => (QuotedString(acc),rest)
          case other => error("Unexpected inside quote: " + other)
        }
      }
```

2.2 Building blocks

The final document will consist of a number of blocks as defined above, so the next step will be to parse these blocks. We will define a block builder class like this:

⟨*Build blocks*⟩ ≡

```
case class BlockBuilder(lines: Stream[Line]) {
  def blocks: Stream[Block] = lines match {
    case Stream.cons(_,beg @ Stream.cons(Doc(0),_)) ⇒ {
      selectNext(beg,0)
    }
    case _ ⇒ error("Unexpected beginnig: " + lines.take(2).toList)
  }
```

The filename has to be extracted separately because it will not be part of any block.

⟨*Build blocks*⟩ + ≡

```
  def filename: String = lines.head match {
    case File(fname) ⇒ fname
    case other ⇒ error("Unexpected first line: " + other)
  }
  <define how to read up to a line type>
  <define documentation and code splitting>
}
```

Basicall, documentation and code splitting use one common part: Read up to `EndCode` or `EndLine`, all while incrementing line numbers. This functionality can be extracted:

⟨*define how to read up to a line type*⟩ ≡

```
def readUpToTag(ls: Stream[Line],
                acc: Stream[Line],
                linenumber: Int,
                endTag: Line):
  (Stream[Line],Stream[Line],Int) = ls match {
    case Stream.empty =>
      error("Expected end tag but found end of stream")
    case Stream.cons(first,rest) =>
      if( first == endTag )
        (acc.reverse,rest,linenumber)
      else first match {
        case NewLine =>
          readUpToTag(rest,
            Stream.cons(first,acc),
            linenumber + 1,endTag)
        case other =>
          readUpToTag(rest,
            Stream.cons(first,acc),
            linenumber,endTag)
      }
}
```

This would be quite a bit more flexible if we could just check for a specific type, but somehow erasure prevents me from doing that.

The real work will be done with the two methods, documentation and code (which will call one another via `selectNext`): They split the content along the lines. First the function selectNext:

⟨*define documentation and code splitting*⟩ ≡

```
def selectNext(ls: Stream[Line],
                 linenumber: Int): Stream[Block] =
  ls match {
    case Stream.empty ⇒ Stream.empty
    case Stream.cons(first,rest) ⇒ first match {
      case Doc(n) ⇒ documentation(rest,n,linenumber)
      case Code(n) ⇒ code(rest,n,linenumber)
      case other ⇒ error("Expected begin code or begin doc" +
                          "but found " + other)
    }
  }
```

Nothing too spectacular here. For documentation, we will pass everything up to `EndDoc(n)` to `DocuBlock`.

⟨*define documentation and code splitting*⟩ + ≡

```
def documentation(ls: Stream[Line],
                    blocknumber: Int,
                    linenumber: Int): Stream[Block] =
  {
```

With the function readUpToTag, this becomes quite simple:

⟨*define documentation and code splitting*⟩ + ≡

```
ls match {
  case Stream.empty ⇒ error("Unexpected empty doc block")
  case s @ Stream.cons(first,rest) ⇒ {
    val (blockLines,cont,nextline) =
    readUpToTag(s,Stream.empty,linenumber,EndDoc(blocknumber))
    Stream.cons(
    DocuBlock(blocknumber,
              linenumber,
              blockLines),
     selectNext(cont,nextline))
   }
 }
}
```

The code splitting will work in exactly the same way, but we have to take care of another element: The name of the code block.

⟨*define documentation and code splitting*⟩ +≡

```
def code(ls: Stream[Line],
         blocknumber: Int,
         linenumber: Int): Stream[Block] = {
```

The format requires that the first element inside a code block is the chunk name that is defined. Also, we eat the newline that comes directly after that. Because we eat this, we'll also have to update the information on from which line we actually have content.

⟨*define documentation and code splitting*⟩ +≡

```
val Stream.cons(defline,Stream.cons(nline,cont)) = ls
val chunkname = defline match {
  case Definition(name) => name
  case other => error("Expected definition but got " + other)
}
val cont2 = nline match {
  case NewLine => cont
  case _ => Stream.cons(nline,cont)
}
val linenumber2 = linenumber + 1
```

With this information, we can accumulate the content:

⟨*define documentation and code splitting*⟩ +≡

```
    ls match {
      case Stream.empty ⇒ error("Unexpected empty code block")
      case Stream.cons(first,rest) ⇒
        val (lines,continue,lnumber) =
          readUpToTag(cont2,Stream.empty,
                      linenumber2,EndCode(blocknumber))

      Stream.cons(
        CodeBlock(blocknumber,
                  linenumber2,
                  lines,
                  chunkname),selectNext(continue,lnumber))
    }
  }
```

2.3 Testing the block format

The following application will read in a literate program and output each element of the stream of blocks.

⟨*Test the block format*⟩ ≡

```
object Blocks {
  def usage: Unit = {
    System.err.println("Usage: scala markup.Blocks [infile]\n")
  }

  def main(args: Array[String]) = {
    import util.conversions._

    val blocks = args.length match {
      case 0 ⇒ blocksFromLiterateInput(System.in)
      case 1 ⇒ blocksFromLiterateFile(args(0))
      case _ ⇒ usage; exit
    }

    blocks foreach {
      b ⇒ println(b)
    }
  }
}
```

3 How to extract source code from a literate program

`tangle` is the tool that extracts source code from a given literate program. For this extraction, we base ourselves on the stream of blocks generated in the file `markup/blocks.nw`

3.1 Overview

Tangle will first extract a map of code blocks, from which it will output the sources in the right format the routines for output are directly associated with the map and will be explained in section **??**.

⟨*⟩ +≡

> ***package*** *scalit.tangle*
> ***import*** *markup._*
>
> *<code chunks>*
>
> *<puzzle code chunks together>*
>
> *<output the source>*

3.2 Puzzling code blocks together

3.2.1 Code chunks

While the block generator already provides us with a stream of blocks, several of these might be describing the same code chunk. So a pointer to the next code block describing this chunk has to be provided. We do this using an `Option`:

⟨*code chunks*⟩ ≡

```
case class CodeChunk(bn: Int, ln: Int,
                     cont: Stream[Line],bname: String,
                     next: Option[CodeChunk]) extends
CodeBlock(bn,ln,cont,bname) {

  import StringRefs._
```

The append method will be useful when we will actually construct the chunks.

⟨*code chunks*⟩ + ≡

```
  def append(that: CodeBlock): CodeChunk = next match {
    case None ⇒
      CodeChunk(this.blocknumber,
                this.linenumber,
                this.content,
                this.blockname,
                Some(CodeChunk(that.blocknumber,
                    that.linenumber,
                    that.content,
                    that.blockname,None)))
    case Some(next) ⇒
      CodeChunk(this.blocknumber,
                this.linenumber,
                this.content,
                this.blockname,
                Some(next append that))
  }
```

With this linked-list-like definition in place, we can also redefine the string reference form by simply appending the output of the next element:

⟨*code chunks*⟩ + ≡

```
    override def stringRefForm(codeChunks: Map[String,CodeBlock]):
      Stream[StringRef] = next match {
        case None ⇒ super.stringRefForm(codeChunks)
        case Some(el) ⇒ Stream.concat(
          super.stringRefForm(codeChunks),
          el.stringRefForm(codeChunks))
      }
  }
```

3.2.2 A collection of chunks

In a chunk collection, we accumulate chunks on in a map. Also very important is the file name.

⟨*puzzle code chunks together*⟩ ≡

```
  import scala.collection.immutable.{Map,HashMap}
  case class ChunkCollection(cm: Map[String,CodeChunk],
                             filename: String) {

    import StringRefs._
```

To get the stream of code is now as simple as calling serialize. Flatten will convert it to a string.

⟨*puzzle code chunks together*⟩ +≡

```
    def serialize(chunkname: String): String =
      cm get chunkname match {
        case None ⇒ error("Did not find chunk " + chunkname)
        case Some(el) ⇒ flatten(el.stringRefForm(cm))
      }
```

From the stream of blocks, we will receive the code blocks. We will have to generate code chunks out of them.

⟨*puzzle code chunks together*⟩ +≡

```
def addBlock(that: CodeBlock): ChunkCollection =
  cm get that.blockname match {
    case None => ChunkCollection(cm +
      (that.blockname ->
       CodeChunk(that.blocknumber,
                 that.linenumber,
                 that.content,
                 that.blockname,
                 None)),filename)
    case Some(el) => ChunkCollection(cm +
      (that.blockname ->
       el.append(that)),filename)
  }
```

While adding one block is useful, we will want to do this for a whole stream of blocks:

⟨*puzzle code chunks together*⟩ +≡

```
def addBlocks(those: Stream[CodeBlock]): ChunkCollection =
  (those foldLeft this) {
    (acc: ChunkCollection, n: CodeBlock) =>
      acc.addBlock(n)
  }
```

Finally, we'll have to define how to output a string containing the whole code. In a first step, we'll have to expand references:

⟨*puzzle code chunks together*⟩ +≡

```
    def expandRefs(str: Stream[StringRef]): Stream[RealString] =
      str match {
        case Stream.empty => Stream.empty
        case Stream.cons(first,rest) =>
          first match {
            case r @ RealString(_,_,_) =>
              Stream.cons(r,expandRefs(rest))
            case BlockRef(ref) =>
              Stream.concat(
                expandRefs(cm(ref.blockname).stringRefForm(cm)),
                expandRefs(rest))
            case other => error("Unexpected string ref: " + other)
          }
      }
    def expandedStream(chunkname: String): Stream[RealString] =
      cm get chunkname match {
        case None => error("Did not find chunk " + chunkname)
        case Some(el) => expandRefs(el.stringRefForm(cm))
      }
```

After this expansion, the string form is quite easily made:

⟨*puzzle code chunks together*⟩ +≡

```
    private def flatten(str: Stream[StringRef]): String = {
      val sb = new StringBuffer
      expandRefs(str) foreach {
        case RealString(content,_,_) => sb append content
      }
      sb.toString
    }
  }
```

With the chunk collection logic in place, we will often have to access to the empty chunk collection of a particular file name:

⟨*puzzle code chunks together*⟩ +≡

```scala
case class emptyChunkCollection(fn: String)
    extends ChunkCollection(Map(),fn)
```

3.3 The main program

With serialize defined, we can now accomplish the task of printing the tangled source to standard output. Under `util/commandline.nw`, we defined a class for command line parsing that will be used here. At the moment, we print out everything to standard output, one chunk collection after another.

The following options can be given to tangle:

-r chunkname Tries to extract the chunk with the name `chunkname`

⟨*output the source*⟩ ≡

```scala
object Tangle {
  def main(args: Array[String]) = {
    import util.LiterateSettings

    val ls = new LiterateSettings(args)

    val chunksToTake = ls.settings get "-r" match {
      case None => Nil
      case Some(cs) => cs.reverse
    }

    val out = ls.output

    chunksToTake match {
      case Nil =>
        ls.chunkCollections foreach {
          cc => out.println(cc.serialize("*"))
        }
```

If we have specified some chunks to extract, we iterate over all the files that we are given, extracting the specific chunk.

⟨*output the source*⟩ +≡

```
                **case** cs ⇒
                    cs foreach {
                        chunk ⇒
                            ls.chunkCollections foreach {
                                cc ⇒
                                    **try** {
                                        out.println(cc.serialize(chunk))
                                    } **catch** {
                                        **case** e ⇒ ()
                                    }
                            }
                    }
            }
        }
    }
```

4 A closer integration in the Scala compiler

In the file `tangle.nw`, we outlined on how one could puzzle together a compileable file out of a stream of blocks. This is very useful on its own, but a closer compiler integration is desirable: This way, we can tell the compiler the real line numbers of our literate program and not of the tangled source - we will be able to debug more efficiently. Also, we can access information provided by the compiler: What was defined by this block and what variables will be used.

Concretely, we will be implementing `CoTangle` which will call the compiler with the code taken from the stream of blocks. `LitComp` is the command line application that directly compiles an input literate program.

Additionally, we provide an object that directly exposes `LiterateProgramSourceFiles`. This way, we can even use `scalac` to work with literate programs.

⟨*⟩ + ≡

```
package scalit.tangle
<CoTangle − send tangled to compiler>
<A source file format for literate programs>
<A new position type>
<LitComp − the command line application>
<LiterateCompilerSupport − object for scalac>
```

4.1 CoTangle

The following class will pass the source files that we get to the compiler (and maybe a destination directory), so a first step is to include the compiler class. Later, we'll outline how to actually compile.

⟨*CoTangle - send tangled to compiler*⟩ ≡

```
class CoTangle(sourceFiles: List[LiterateProgramSourceFile],
               destination: Option[String]) {
  <Include a compiler>
  <Compile a literate program>
}
```

The compiler will also have to have a place to report errors to, so we will need a reporter. The usual way would be to have an object overriding some behavior for this, but we will just instantiate a standard reporter.

⟨*Include a compiler*⟩ ≡

```
  import scala.tools.nsc.{Global,Settings,reporters}
  import reporters.ConsoleReporter
```

At the moment, there is only one setting that we might give to the compiler: If we have a destination directory, then we'll pass it

⟨*Include a compiler*⟩ + ≡

```
val settings = new Settings()
destination match {
  case Some(dd) ⇒ {
    settings.outdir.tryToSet(List("−d",dd))
  }
  case None ⇒ ()
}
val reporter =
  new ConsoleReporter(settings,null,
                      new java.io.PrintWriter(System.err))
val compiler = new Global(settings,reporter)
```

4.2 Literate program as a source file

After we have instanciated a compiler, we want to feed him a source file. In this file, we'd like to conserve original line numbering etc.

Much of the functionality will be close to a batch source file. The main thing that changes is the mapping position → position in literate file.

⟨*A source file format for literate programs*⟩ ≡

```
import scala.tools.nsc.util.{BatchSourceFile,Position,LinePosition}
class LiterateProgramSourceFile(chunks: ChunkCollection)
  extends BatchSourceFile(chunks.filename,
                          chunks.serialize("∗").toArray) {
  <line mappings>
  <find a line>
  <the original source file>
  <position in ultimate source file>
}
```

4.3 Position in ultimate source file

The tangling process rearranges lines from the original literate program so that they are a valid source input. For compiling purposes, we want to find out to which original line a line in the tangled file corresponds. As this will be done quite often, we want to cache the results:

⟨*line mappings*⟩ ≡

val *lines2orig* = **new** *scala.collection.mutable.HashMap*[*Int*,*Int*]()

Finding a line amounts to iterating over the stream of code blocks until we find the one containing the line. This might not be an optimal solution - now that we are eating the first newline after the chunk definition, it might actually happen that we point to the wrong line, for example with code like

```
val s = <Read s>
```

and

⟨⟨`Read s`⟩⟩=

```
1/0
```

Here, the error is in the second part of the code, but the line corresponding to the first part will be returned. One way to fix this would be to consider offset information.

⟨*find a line*⟩ ≡

```
import scalit.markup.StringRefs._
lazy val codeblocks: Stream[RealString] =
  chunks.expandedStream("*")

def findOrigLine(ol: Int): Int =
  if( lines2orig contains ol ) lines2orig(ol)
  else {
    def find0(offset: Int,
        search: Stream[RealString]): Int = search match {
      case Stream.empty => error("Could not find line for " + ol)
      case Stream.cons(first,rest) =>
        first match {
          case RealString(cont,from,to) => {
            val diff = to - from
            if( ol >= offset && ol <= offset + diff ) {
              val res = from + (ol - offset)
              lines2orig += (ol -> res)
              res
            } else
              find0(offset + diff,rest)
          }
        }
    }

    find0(0,codeblocks)
  }
```

Most of the work was already done in the tangling phase, so we can just check whether we are inside a string from the source file. For error reporting, we will want to point to the original source file, but there is, of course a problem: The source might come from a markup file, or standard input and not just a literate program. But in any case except reading a literate program from standard input (which is of dubious utility anyway), we know the original source file because of the `@file` directive, which is then given to us as an argument. This is very suboptimal: We slurp the whole file for random access:

⟨*the original source file*⟩ ≡

```
import scala.tools.nsc.util.{SourceFile,CharArrayReader}
lazy val origSourceFile = {
  val f = new java.io.File(chunks.filename)
  val inf = new java.io.BufferedReader(
    new java.io.FileReader(f))
  val arr = new Array[Char](f.length().asInstanceOf[Int])
  inf.read(arr,0,f.length().asInstanceOf[Int])
  new BatchSourceFile(chunks.filename,arr)
}
```

With all this information, we can finally override the method that tells us the position in the original source file. To access the original source file,

⟨*position in ultimate source file*⟩ ≡

```
override def positionInUltimateSource(position: Position) = {
  val line = position.line match {
    case None ⇒ 0
    case Some(l) ⇒ l
  }
  val col = position.column match {
    case None ⇒ 0
    case Some(c) ⇒ c
  }
  val literateLine = findOrigLine(line)
  LineColPosition(origSourceFile,literateLine,col)
}
```

The position class that we return is also defined especially for this use - we do not want to count the offset into the literate file:

⟨*A new position type*⟩ ≡

```
import scala.tools.nsc.util.SourceFile
case class LineColPosition(source0: SourceFile, line0: Int,
                           column0: Int) extends Position {
  override def offset = None
  override def column: Option[Int] = Some(column0)
  override def line: Option[Int]   = Some(line0)
  override def source = Some(source0)
}
```

4.4 The compilation process

With the source file format in place, calling the compiler becomes quite simple: We create a new `Run` which will compile the files:

⟨*Compile a literate program*⟩ ≡

```
def compile: Global#Run = {
  val r = new compiler.Run

  r.compileSources(sourceFiles)
  if( compiler.globalPhase.name != "terminal" ) {
    System.err.println("Compilation failed")
    System.exit(2)
  }

  r
}
```

At the moment, we are not very specific about error reporting: If compilation does not work, we'll just exit.

4.5 The command line application

For this command line application, we just take the list of chunks from the literate settings. Then fo every such chunk we'll create a source file. All of these source files are compiled together and stored in the path given by the `-d` command line flag.

⟨*LitComp - the command line application*⟩ ≡

```
object LitComp {
  def main(args: Array[String]): Unit = {
    import scalit.util.LiterateSettings

    val ls = new LiterateSettings(args)

    val sourceFiles = ls.chunkCollections map {
      cc ⇒ new LiterateProgramSourceFile(cc)
    }

    val destinationDir: Option[String] =
      ls.settings get "−d" match {
        case Some(x :: xs) ⇒ Some(x)
        case _ ⇒ None
      }

    val cotangle = new CoTangle(sourceFiles,
                                destinationDir)

    cotangle.compile
  }
}
```

4.6 Support for scalac

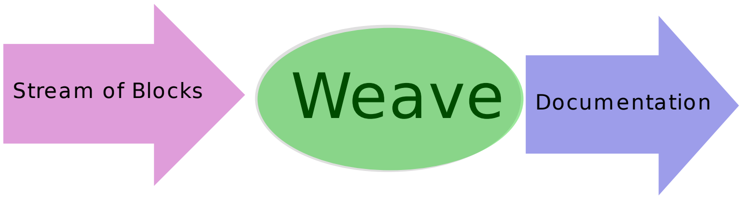
When the `scalac` compiler is invoked, it works directly on `SourceFiles`. Before these are parsed, we have to map the literate file to one of concrete code. Fortunately, we already have a class `LiterateProgramSourceFile` which serves that purpose. We therefore only provide one function, taking the filename of the literate source, building the chunks and then returning a `LiterateProgramSourceFile`.

⟨*LiterateCompilerSupport - object for scalac*⟩ ≡

```
object LiterateCompilerSupport {
  def getLiterateSourceFile(filename: String): BatchSourceFile = {
    import scalit.util.conversions
    val cbs = conversions.codeblocks(conversions.blocksFromLiterateFile(filename))
    val chunks = emptyChunkCollection(filename) addBlocks cbs
    new LiterateProgramSourceFile(chunks)
  }
}
```

5 Weave - Creating documentation out of a literate program

The program described here allows us to extract a human readable file out of the `noweb` input. The content will be output in the order that it was written in the noweb file, but code sections will be annotated and we will gather information that allows us to indicate information like which class was defined where and so on.



It is important to note that there exists a tool to do syntactic highlighting for scala called `verbfilter`[3]. The method `process` takes a character buffer and looks for `\begin{verbatim}`, there it will begin to transform the input. Our aim is therefore to convert the code sections to a character buffer so that it can be fed to verbfilter.

⟨*⟩ +≡

```
package scalit.weave
import markup._
<The LaTeX weaver>
<Source file information>
<The command line application>
```

5.1 The LaTeX weaver

LatexWeaver, the class that takes care of producing LaTeX output from a list of streams of blocks (as defined in `weave/blocks.nw`) has two methods: One for printing one block, `printBlock`, and one to print everything surrounding to produce a valid LaTeX document.

⟨*The LaTeX weaver*⟩ ≡

[3] To be found under misc/scala-tool-support/latex

```
sealed abstract class Weaver(blocks: List[(Stream[Block],String)])
case class LatexWeaver(blocks: List[(Stream[Block],String)],
                       tangled: List[tangle.ChunkCollection],
                       useVerbfilter: Boolean,
                       useIndex: Boolean,
                       classpath: Option[List[String]],
                       filename: String,
                       useHeader: Boolean)
     extends Weaver(blocks) {
  import java.io.PrintStream
  <escape in quoted sections>
  <escape code>
  <print one block>
  <print the document>
  <compiler help>
  <format index>
}
```

The useVerbfilter flag tells the weaver whether it should fire up the compiler to retreive information on the source code. This is made optional because it is pretty expensive. If `useHeader` is set to false, then we will not print a header (ideal for inclusion in other LaTeX files).

5.2 Printing one block

There are two types of blocks to consider, code and documentation blocks. In documentation blocks, we will need to know how to escape content: The backslash character especially has to be escaped.

⟨*escape in quoted sections*⟩ ≡

```
def escape(orig: String): String = {
  orig.replace("\\","$\\backslash$")
      .replace("{","$\\{$")
      .replace("}","$\\}$")
      .replace("#","$\\#$")
      .replace("<","$\\langle$")
      .replace(">","$\\rangle$")
      .replace("_","\\_")
}
```

This function will undoubtedly be extended by more escape sequences. Now on how to actually print these blocks. One speciality that we want to indicate is whether we have already begun the definition of a given chunk. As this would be too cumbersome to carry around, we'll keep track of it here:

$\langle$*print one block*$\rangle \equiv$

```
val chunksSeen =
  new scala.collection.mutable.HashMap[String,CodeBlock]
```

On to the block printing:

$\langle$*print one block*$\rangle + \equiv$

```
import StringRefs._
def printBlock(out: PrintStream,
        chunks: tangle.ChunkCollection)
        (b: Block): Unit = b match {
  case cb @ CodeBlock(bn,ln,content,blockname) => {
    out.print("$\\left<\\mbox{\\emph{" +
          blockname +
          "}}\\right>")
    if( chunksSeen contains blockname ) out.print("+")
    else chunksSeen += (blockname -> cb )
    out.println("\\equiv$")
```

Chunks that get defined for the first time start with $\langle name \rangle \equiv$, a continued chunk will be of the form $\langle name \rangle + \equiv$. If we use other code chunks, this will be noted by ⟨*name*⟩. In the following section we will slurp the whole content of the code block into a string:

⟨*print one block*⟩ + ≡

```
var begin = -1
var end   = -1
val content = "\\begin{verbatim}" +
((cb.stringRefForm(chunks.cm) map {
    case RealString(cont,from,to) => {
        if( begin == -1 ) begin = from

        if( to > end ) end = to

        cont
    }
    case BlockRef(b) => {
        "<" +
        b.blockname +
        ">"
    }
    case other => error("Unexpected: " + other)
} foldLeft "") {
    (acc: String, next: String) => acc + next
}) + "\\" + "end{verbatim}"
```

If we want to use verbfilter, then we'll pass it to the script. As we will have some unescaping to do (especially the $ character is troublesome) the output is not directly sent to out but stored in a byte array, on which we can then apply the unescaping. Also, if we want to create an index, we'll have to tell it here.

⟨*print one block*⟩ + ≡

```
    if( useVerbfilter ) {
      val vfOutput = new java.io.ByteArrayOutputStream
      toolsupport.verbfilterScala
        .process(codeEscape(content).getBytes,vfOutput)
      if( useIndex )
        out.println(
          indexed(
          codeUnescape(
          vfOutput.toString),
          begin,end,chunks.filename))
      else
        out.println(codeUnescape(vfOutput.toString))
    } else {
      if( useIndex )
        out.println(indexed(content,
          begin,end,chunks.filename))
      else
        out.println(content)
    }
  }
```

Here we just avoided a quine-like problem: `\end{verbatim}` is the sequence to terminate a code block, so if it occurs inside a code block, then we could run into a problem.

Documentation blocks will contain escaped sections (quoted), but otherwise they will be copied verbatim.

⟨*print one block*⟩ + ≡

```
case d @ DocuBlock(bn,ln,content) ⇒ {
  d.stringRefForm(Map()) foreach {
    x ⇒ x match {
      case RealString(cont,_,_) ⇒ out.print(cont)
      case QuotedString(cont) ⇒ {
        out.print("\\texttt{")
        out.print(escape(cont))
        out.print("}")
      }
      case BlockRef(_) ⇒
        error("Did not expect code reference" +
        " in documentation chunk")
    }
  }
}
}
```

5.2.1 Escaping code

As we will pass code to the `verbfilter` program afterwards, we have to be very careful with some code content that could also be interpreted as LaTeX escape sequences: We have to strip them out:

⟨*escape code*⟩ ≡

```
def codeEscape(code: String): String = {
  code.replace("$","SPEC" + "DOLLAR")
}
```

The problem then is, of course, that we will need to put them back in afterwards.

⟨*escape code*⟩ + ≡

```
def codeUnescape(code: String): String = {
  code.replace("SPEC" + "DOLLAR","\\Dollar")
}
```

5.2.2 Wrapping the document

With the knowledge on how to print blocks, we can go on printing the whole document. If the useHeader flag is set (which it is by default), we generate a standard LaTeX document, the only thing special to note is that we add `scaladefs` which contains macros to format scala output and `scalit` which enables definition indexing.

⟨*print the document*⟩ ≡

```
def writeDoc(out: PrintStream): Unit = {
  if( useHeader ) {
    out.println("\\documentclass[a4paper,12pt]{article}")
    out.println("\\usepackage{amsmath,amssymb}")
    out.println("\\usepackage{graphicx}")
    out.println("\\usepackage{scaladefs}")
    out.println("\\usepackage{scalit}")
    out.println("\\usepackage{fancyhdr}")
    out.println("\\pagestyle{fancy}")
    out.println("\\lhead{\\today}")
    out.println("\\rhead{" + escape(filename) + "}")
    out.println("\\begin{document}")
  }

  blocks zip tangled foreach {
    case ((bs,_),tang) => bs foreach printBlock(out,tang)
  }
  if( useHeader ) {
    out.println("\\end{document}\n\n")
  }
}
```

5.3 Information on the source file

During tangling, we directly interact with the compiler to compile from literate programs. But the compiler can be of much more help - We can for example find out where classes are defined, etc. For this, we reuse the source file class defined for literate compilation[4].

[4] tangle/compilesupport.nw

Another optional parameter is where to find the classes containing the other definitions: This will be used by the compiler to typecheck the code, thus generating the symbols we need.

⟨*Source file information*⟩ ≡

```
import scala.tools.nsc.ast.Trees
class SourceInformation(
    literateFile: tangle.LiterateProgramSourceFile,
    infoClassPath: Option[List[String]]) {
    <Instantiate a compiler>

    <collect information>
    <range of definitions>
}

<Definition info storage>
```

as with the compiler support class, we'll have to instantiate a compiler. However, we will not need to do all the phases, so we overwrite the phases we need:

⟨*Instantiate a compiler*⟩ ≡

```
    import scala.tools.nsc.{Global,Settings,SubComponent}
    import scala.tools.nsc.reporters.ConsoleReporter

    val settings = new Settings()
    infoClassPath match {
        case None ⇒ ()
        case Some(cp) ⇒ settings.classpath.value = cp.head
    }

    val reporter =
        new ConsoleReporter(settings,null,
                                    new java.io.PrintWriter(System.err))
    object compiler extends Global(settings, reporter) {
        override protected def builtInPhaseDescriptors:
        List[SubComponent] = List(
            analyzer.namerFactory: SubComponent,
            analyzer.typerFactory: SubComponent
        )
    }
```

with the compiler in place, we can now define how to collect the information: Execute the compiler just up to typing and collect them from the syntax tree.

⟨*collect information*⟩ ≡

```
lazy val info = {
  val r = new compiler.Run
  r.compileSources(literateFile :: Nil)
  val typedUnit = r.units.next
  collectDefinitions(typedUnit.body,Map())
}
```

The definition collector needs to have access to the tree case classes. They are part of the compiler. We are very forgiving if, for example we do not find a valid position.

⟨*collect information*⟩ + ≡

```
import compiler.{Tree,ClassDef,ModuleDef,PackageDef,
                 DefDef,ValDef,Template}
import DefinitionInfo._
```

We will want to have access to the information on a per-line-basis. However, there will be multiple definitions on one line, so we will need something like a multiset:

⟨*collect information*⟩ + ≡

```
type DefMap = Map[Int,Set[Definition]]
```

Another useful state to store is in which class we currently are, so that we can link methods (which might be defined in multiple classes) to a specific class. We overload this method to stay succinct.

⟨*collect information*⟩ + ≡

```
def collectDefinitions(t: Tree, acc: DefMap): DefMap =
  collectDefinitions(t,acc,None)

def collectDefinitions(t: Tree,
                       acc: DefMap,
                       container: Option[String]): DefMap = {
```

After these overloaded definitions, let'l begin by getting the source file position:

⟨*collect information*⟩ +≡

```
    val pos = literateFile.positionInUltimateSource(t.pos)
    val line = pos.line match {
      case None ⇒ −1
      case Some(l) ⇒ l
    }
    val before = acc.getOrElse(line,Set())
    t match {
<Handle the class case>
<Handle the object case>
<Handle the package case>
<Handle the method case>
<Handle the value case>
      case other ⇒ acc
    }
  }
```

The accumulator style makes this function rather heavy (note all the folds), but this way we can append the definitions in a predictable style. So, on to the starting point in our tree: The package definition

⟨*Handle the package case*⟩ ≡

```
      case PackageDef(name,stats) ⇒
        (stats foldLeft acc) {
          (defs: DefMap, t: Tree) ⇒ collectDefinitions(t,defs)
        }
```

`defs` holds the current state of the map. Nodes of this package definition will be classes and objects. We will first collect everything in the first class, then pass the definition results to the second class, etc. Here is what we do with classes:

⟨*Handle the class case*⟩ ≡

```
case ClassDef(_,name,tparams,impl) ⇒
  val nameString = name.toChars mkString ""
  val newAcc =
    acc + (line → (before +
                   ClassDefinition(nameString,line,false)))
  impl match {
    case Template(_,_,body) ⇒
      (body foldLeft newAcc) {
        (defs: DefMap, t: Tree) ⇒
          collectDefinitions(t,defs,Some(nameString))
      }
  }
```

Classes have a body which we need to scan, but before we will have to add the class itself to the map. If only that were so easy! Note that there are three basic types of top-level objects on the class level:

- "Normal" classes
- Objects
- Case classes / objects

Note that this is not an either/or decision. A normal way to emulate static members in Scala is the following:

```
class A {
  def aMethod = A.staticOne()
}
object A {
  def staticOne() = ...
}
```

So we will be allowed to have object definitions and class definitions on different lines. The compiler, however, seems to generate both `ClassDef` and `ModuleDef` nodes when we have a case class. Also, the definition map is in an incomplete state, so we can only bet on not seeing another `ClassDef` afterwards by assuming what the compiler generates first the module definition element. This first test tells us to only add contents if we are sure this actually is an object. If we are dealing with a case class, we'll have to indicate this, for this we will use the variable classDefinition.

⟨*Do not add if object is there*⟩ ≡

```
var classDefinition: Option[ClassDefinition] = None
val isRealObject = acc get line match {
  case Some(s) => !(s exists {
    case cd @ ClassDefinition(n,_,_) => {
      if( n == nameString ) {
        classDefinition = Some(cd)
        true
      } else false
    }
    case other => false
  })
  case None =>        true
}
```

If we have already content on this line, then we will not traverse it again, but we will update the fact that we are dealing with a case class.

⟨*Handle the object case*⟩ ≡

```
case ModuleDef(mods,name,impl) ⇒
  val nameString = name.toChars mkString ""
  <Do not add if object is there>
  if(isRealObject) {
    val newAcc = acc + (line →
        (before + ObjectDefinition(nameString,line)))
    impl match {
      case Template(_,_,body) ⇒
        (body foldLeft newAcc) {
          (defs: DefMap, t: Tree) ⇒
            collectDefinitions(t,defs,Some(nameString))
        }
    }
  } else {
    val Some(cd) = classDefinition
    val cc = ClassDefinition(cd.name,cd.l,true)
    acc + (line → (before − cd + cc))
  }
```

For the definitions, We will have to use the same trick as for objects: Some methods are added automatically (like toString, ¡init¿) and should therefore not be included in the index. We solve this by looking up whether a class was defined on the same line. Also, it might be that there exists already a value definition with the same name, in which case we won't add it either:

⟨*Handle the method case*⟩ ≡

```
case DefDef(_,name,tparams,vparams,tpt,_) ⇒
  val isGeneratedMethod = acc get line match {
    case None ⇒ false
    case Some(s) ⇒ s exists {
      case c : ClassDefinition ⇒ true
      case o : ObjectDefinition ⇒ true
      case vd : ValueDefinition ⇒ true
      case _ ⇒ false
    }
  }
  if( isGeneratedMethod ) acc
  else acc + (line →
    (before + MethodDefinition(name.toString,line,container)))
```

We also record value definitions

⟨*Handle the value case*⟩ ≡

```
case ValDef(_,name,_,_) ⇒ {
  val nameString = name.toChars mkString ""
  acc + (line → (before + ValueDefinition(nameString,line,container)))
}
```

One useful command would be to get all the definitions in a range of lines:

⟨*range of definitions*⟩ ≡

```
def getRange(from: Int, to: Int): List[Definition] =
  List.range(from,to + 1) flatMap {
    i ⇒ info.getOrElse(i,Nil)
  }
```

5.3.1 Connection to the weaver

Especially in LaTeXWeaver, we will want to auto-generate some annotations, therefore we'll have to have a value for that:

⟨*compiler help*⟩ ≡

```
import tangle.LiterateProgramSourceFile
val sourcefiles: Option[List[(String,LiterateProgramSourceFile)]] =
  if( useIndex ) {
    Some(tangled map {
      chunks ⇒
        (chunks.filename,
         new tangle.LiterateProgramSourceFile(chunks))
    })
  } else None

val sourceInformation: Map[String,SourceInformation] =
sourcefiles match {
  case None ⇒ Map()
  case Some(sfs) ⇒ Map() ++ (sfs map {
    case (name,sf) ⇒ (name → new SourceInformation(sf,classpath))
  })
}
```

5.3.2 Storing the gathered information

While traversing the tree, we need to store information on what is actually defined. This is particularly:

- Class and trait definitions
- Object definitions
- Method definitions

To each element, we'll want to store the line number so that it can be easily retrieved.

⟨*Definition info storage*⟩ ≡

```
object DefinitionInfo {
  sealed abstract class Definition(line: Int)

  case class ClassDefinition(name: String, l: Int,
                             isCase: Boolean)
    extends Definition(l) {
      override def toString = "" + l + ": Class " + name
    }

  case class ObjectDefinition(name: String,
                              l: Int)
    extends Definition(l) {
      override def toString = "" + l + ": Object " + name
    }

  case class MethodDefinition(name: String,
                              l: Int,
                              container: Option[String])
    extends Definition(l) {
      override def toString = "" + l + ": Method " + name
    }

  case class ValueDefinition(name: String,
                             l: Int,
                             container: Option[String])
    extends Definition(l) {
      override def toString = "" + l + ": Value " + name
    }
}
```

5.3.3 Testing the information gathering

The following command line tool tests the information gathering:

⟨*Source file information*⟩ + ≡

```
object InfoTester {
  def main(args: Array[String]) = {
    import util.LiterateSettings
    import tangle.LiterateProgramSourceFile


    val ls = new LiterateSettings(args)

    val sourceFiles =
      ls.chunkCollections map (
        new LiterateProgramSourceFile(_))

    val infoclasspath = ls.settings get "-classpath"

    val infolist =
      sourceFiles map (new SourceInformation(_,infoclasspath))

    infolist foreach { x => println(x.getRange(0,50)) }
  }
}
```

5.3.4 Adding index information

With the compiler support that we defined before, we will be able to print which functions were defined in a piece of code. The following function adds these bits:

⟨*format index*⟩ ≡

```
  def indexed(content: String, from: Int,
              to: Int, filename: String): String = {
    import DefinitionInfo._
    val ret = new StringBuffer
    ret append content

    sourceInformation get filename match {
      case None => ()
      case Some(si) => {
        val definitions = si.getRange(from,to)
```

The first definitions that we are interested in are which classes are defined:

⟨*format index*⟩ + ≡

```
val classes = definitions filter {
  case cd: ClassDefinition => true
  case _ => false
}
```

Then methods. Here, we want to filter out the generated methods

⟨*format index*⟩ + ≡

```
val methods = definitions filter {
  case MethodDefinition(name,_,_) => true
  case _ => false
}
```

Values and objects are filtered in a same way

⟨*format index*⟩ + ≡

```
val values = definitions filter {
  case v : ValueDefinition => true
  case _ => false
}
val objects = definitions filter {
  case o: ObjectDefinition => true
  case _ => false
}
```

Now for the LaTeX output generated: We use the commands defined in `scalit.sty` so that we can change presentation later on. Note that no real output needs to be generated.

⟨*format index*⟩ + ≡

```
            if (!classes.isEmpty || !methods.isEmpty ||
                  !objects.isEmpty) {
<output class definition info>
<output object definition info>
<output method definition info>
<output value definition info>
            }
            ret append "\n\n"
          }
        }

        ret.toString
      }
```

First the classes. The command `classdefinition` takes care of them. At the moment, we are not indicating case classes specially.

⟨*output class definition info*⟩ ≡

```
classes foreach {
  case ClassDefinition(name,_,caseClass) => {
    ret append "\\classdefinition{" + name + "}\n"
  }
  case _ => ()
}
```

Second come the object definitions.

⟨*output object definition info*⟩ ≡

```
objects foreach {
  case ObjectDefinition(name,_) => {
    ret append "\\objectdefinition{" + name + "}\n"
  }
  case _ => ()
}
```

Methods are prefixed by the class in which they are defined. Note the flaw in this system: This way we are only recording one level of classes, thus generating a flat index. Also, the body of the methods is not further traversed, so inner functions are not detected.

⟨*output method definition info*⟩ ≡

```
methods foreach {
  case MethodDefinition(name,_,cont) => {
    ret append "\\methoddefinition{"
    cont match {
      case None => ()
        case Some(n) => ret append escape(n)
    }
    ret append "}{"
    ret append escape(name) + "}\n"
  }
  case _ => ()
}
```

Value definitions are also noted. This might be a bit overkill, so in a further stage we could filter for only publicly accessible values.

⟨*output value definition info*⟩ ≡

```
values foreach {
  case ValueDefinition(name,_,cont) => {
    ret append "\\valuedefinition{"
    cont match {
      case None => ()
        case Some(n) => ret append escape(n)
    }
    ret append "}{"
    ret append name + "}\n"
  }
  case _ => ()
}
```

5.4 The command line application

The command line application gets quite a bit more complicated than before: Not only do we scan the code blocks but we also tangle the source! This way we can assure that we do not reference code blocks in the text that were not defined. Also, in a later stage we could use compiler information to see what is defined where etc. `LiterateSettings`[5] lets us read in multiple files to form one LaTeX document.

Specifically, it takes the following options:

-vf t/f if t, then apply verbfilter on source. Default true

-idx t/f if t, then generate definition index. Default false

-classpath path Classpath to be used for reference to other classes if index is built.

⟨*The command line application*⟩ ≡

```
object Weave {
  def main(args: Array[String]) = {
    import util.LiterateSettings

    val ls = new LiterateSettings(args)

    val blocks : List[(Stream[markup.Block],String)] = ls.blocks
```

This is the same as with tangle: We want to extract code blocks from the line format. We will actually partially execute the tangle phase: The part where we put chunks together.

But first, let us deal with the title of the file. If one is specified on the command line, we use this. Otherwise, we use the name of the first file given as input.

⟨*The command line application*⟩ + ≡

```
    val filename = ls.settings get "-title" match {
      case Some(x::xs) ⇒ x
      case _ ⇒ blocks match {
        case (_,name) :: xs ⇒ name
        case Nil ⇒ ""
      }
    }
```

[5] See util/commandline.nw

Some settings are taken out of the settings object directly, where to send output is handled specially.

⟨*The command line application*⟩ +≡

```
        val classpath = ls.settings get "-classpath"

        val verbfilter = ls.settings get "-vf" match {
          case Some(x :: xs) => x(0) == 't'
          case _ => true
        }

        val index = ls.settings get "-idx" match {
          case Some(x :: xs) => x(0) == 't'
          case _ => false
        }
```

This follows a very general pattern: If the option does not exist, it gets a default value, otherwise it depends on the option whether we interpret it as truth value or otherwise. The following option is on whether to include a header in the generated `tex` file. By default, we print one:

⟨*The command line application*⟩ +≡

```
        val header = ls.settings get "-header" match {
          case Some(x :: xs) => x(0) == 't'
          case _ => true
        }

        val weaved = LatexWeaver(blocks,ls.chunkCollections,
                                 verbfilter,index,classpath,filename,
                                 header)

        weaved.writeDoc(ls.output)
      }
    }
```

Definitions

6 Conversions

With all the literate programming utilities in place, we will want to access different stages without always setting up a `MarkupGenerator`, reading files etc. The following conversion functions will prove useful:

⟨*⟩ + ≡

```
package scalit.util

object conversions {
  <to line format>

  <to block format>
}
```

6.1 Conversions to line format

The line format will usally be the first step. It is usually either generated from a file or from standard input:

⟨*to line format*⟩ ≡

```
import java.io.{BufferedReader,FileReader,InputStreamReader}
import scala.util.parsing.input.StreamReader

import markup.{Line,MarkupGenerator}

def linesFromLiterateFile(filename: String): Stream[Line] = {
  val input = StreamReader(
                          new BufferedReader(
                          new FileReader(filename)))
  (new MarkupGenerator(input,filename)).lines
}

def linesFromLiterateInput(in: java.io.InputStream): Stream[Line] = {
  val input = StreamReader(new InputStreamReader(in))
  (new MarkupGenerator(input,"")).lines
}
```

We could, of course also get input in markup format. This is treated in the class `MarkupReader`:

⟨*to line format*⟩ + ≡

```
import markup.MarkupParser
def linesFromMarkupFile(filename: String): Stream[Line] = {
  val input = StreamReader(
                new BufferedReader(
                new FileReader(filename)))

  (new MarkupParser(input)).lines
}

def linesFromMarkupInput(in: java.io.InputStream): Stream[Line] = {
  val input = StreamReader(new InputStreamReader(in))
  (new MarkupParser(input)).lines
}
```

6.2 Conversions to block format

The block format takes a stream of lines as input, so we will have four similiar functions that just call the corresponding line generating functions.

⟨*to block format*⟩ ≡

```
import markup.{BlockBuilder,Block}
def blocksFromLiterateFile(filename: String): Stream[Block] =
  BlockBuilder(linesFromLiterateFile(filename)).blocks

def blocksFromLiterateInput(in: java.io.InputStream): Stream[Block] =
  BlockBuilder(linesFromLiterateInput(in)).blocks

def blocksFromMarkupFile(filename: String): Stream[Block] =
  BlockBuilder(linesFromMarkupFile(filename)).blocks

def blocksFromMarkupInput(in: java.io.InputStream): Stream[Block] =
  BlockBuilder(linesFromMarkupInput(in)).blocks
```

Another demand will be to just get the code blocks (for tangle, for example). We'll also have to make a (safe) downcast, unfortunately.

⟨*to block format*⟩ + ≡

```
import markup.{CodeBlock,DocuBlock}
def codeblocks(blocks: Stream[Block]): Stream[CodeBlock] =
  (blocks filter {
    case c: CodeBlock ⇒ true
    case d: DocuBlock ⇒ false
  }).asInstanceOf[Stream[CodeBlock]]
```

7 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.

7.1 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))
}
```

7.1.1 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)
  }
```

7.2 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
          }
      }
    }
```

7.3 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)
}
```

8 A filter mechanism for Scalit

The literate programming mechanism proposed until now is a very static thing, you only have a few choices (whether you want syntax highlighting, an index etc.). Things that you possibly would want to do is to convert a file from its LaTeX formatting into HTML, or add other syntax highlighting.

This file demonstrates a simple filter mechanism for Scalit, which is based on intervention in two stages: On the markup level and on the block level. Such filter modules can then be loaded by using reflection.

⟨*⟩ + ≡

```
package scalit.util
<Filtering on Markup level>
<Sample markup filters>
<Filtering on Block level>
<Sample block filters>
```

8.1 Filtering on the Markup level

Applying a filter on the markup level is quite limited in its capabilities as we only have access to information encoded in these lines. Nevertheless, for example a LaTeX-to-HTML converter could be implemented on this level.

These filters are basically functions from the input stream of lines to an altered stream of lines, therefore we inherit from this function:

⟨*Filtering on Markup level*⟩ ≡

```
import markup.Line
abstract class MarkupFilter extends
  (Stream[Line] ⇒ Stream[Line]) {
  <Feed external utility with lines>
}
```

Note that such filters can also wrap standard Unix tools by feeding them the lines on standard input and parsing the filtered output with a MarkupParser.

⟨*Feed external utility with lines*⟩ ≡

```
  def externalFilter(command: String,
                     lines: Stream[Line]): Stream[Line] = {
```

We take it we are dealing with utilities that take the markup format stream as standard input. If that is not the case, then you will have to write a custom function.

The following code is very Java-intensive and will therefore have to be changed as soon as Scala gets its own form of process control.

First, we'll have to start the process:

⟨*Feed external utility with lines*⟩ +≡

```
val p = Runtime.getRuntime().exec(command)
```

Ok, now we'll have to pass this process a print-out of the lines. We'll do this using a print writer:

⟨*Feed external utility with lines*⟩ +≡

```
val procWriter = new java.io.PrintWriter(p.getOutputStream())
```

Now let us write to this process:

⟨*Feed external utility with lines*⟩ +≡

```
lines foreach procWriter.println
```

The input will now be obtained from the input stream, parsed again in line format. For this, we use the conversion utility described in `uti/conversions.nw`, `linesFromLiterateInput`. We then return this result

⟨*Feed external utility with lines*⟩ +≡

```
    procWriter.close()
    p.waitFor()
    util.conversions.linesFromMarkupInput(p.getInputStream())
}
```

8.1.1 Tee: A sample line filter

To exemplify how such an external tool could be used, the following filter calls the unix utility `tee`, which just replicates its input on the output while also writing to a file.

⟨*Sample markup filters*⟩ ≡

```
class tee extends MarkupFilter {
  def apply(lines: Stream[Line]): Stream[Line] = {
    externalFilter("tee out",lines)
  }
}
```

8.1.2 SimpleSubst: Another example

Filters can also be written in pure Scala, avoiding the overhead of calling an external process and reparsing the markup lines. The following example replaces the sequence "LaTeX" with its nice form: LaTeX

⟨*Sample markup filters*⟩ + ≡

```
class simplesubst extends MarkupFilter {
  def apply(lines: Stream[Line]): Stream[Line] = {
    import markup.TextLine
    lines map {
      case TextLine(cont) ⇒
        TextLine(cont.replace(" " + "LaTeX ","  \\LaTeX "))
      case unchanged ⇒ unchanged
    }
  }
}
```

Now that the filter is defined, using it is as simple as invoking

```
sweave -lfilter util.simplesubst lp.nw -o lp.tex
```

8.2 Filtering on the Block level

Filters on the chunk level are already far more powerful. We already have a high-level view of the document and can do the same level of analysis that we use for the index generation (which is not in filter form).

What we get here as input is the stream of blocks and we return an altered stream of blocks. At the moment, no further help is given to the programmer in the form of utility functions, so we basically can define the interface in one line:

⟨*Filtering on Block level*⟩ ≡

```
import markup.Block
abstract class BlockFilter extends
  (Stream[Block] ⇒ Stream[Block]) {
}
```

8.2.1 A block-level example: Stats

While we do not have much help for filters, writing them still is not too hard, as this simple example (that collects statistics on how many lines of code vs how many lines of documentation were provided).

We could also access the tangled output, but in the interest of simplicity, this example will only deal with the unstructured blocks:

⟨*Sample block filters*⟩ ≡

```
class stats extends BlockFilter {
  def apply(blocks: Stream[Block]): Stream[Block] = {
    val (doclines,codelines) = collectStats(blocks)
```

With these values, we can build a new documentation block holding them:

⟨*Sample block filters*⟩ + ≡

```
    import markup.{TextLine,NewLine}
    val content =
      Stream.cons(NewLine,
      Stream.cons(TextLine("Documentation lines: " +
                           doclines +
                           ", Code lines: " +
                           codelines),
      Stream.cons(NewLine,Stream.empty)))
    Stream.concat(blocks,Stream.cons(
      markup.DocuBlock(−1,−1,content),Stream.empty))
  }
```

Now for the main collection function: It just traverses the content of the blocks in unprocessed form:

⟨*Sample block filters*⟩ + ≡

```
def collectStats(bs: Stream[Block]): (Int,Int) = {
  def collectStats0(str: Stream[Block],
                    doclines: Int,
                    codelines: Int): (Int,Int) = str match {
    case Stream.empty ⇒ (doclines,codelines)
    case Stream.cons(first,rest) ⇒ first match {
```

If a code block is encountered, then we will just increment the number of lines of code:

⟨*Sample block filters*⟩ + ≡

```
      case markup.CodeBlock(_,_,lines,_) ⇒
        val codels = (lines foldLeft 0) {
          (acc: Int, l: markup.Line) ⇒ l match {
            case markup.NewLine ⇒ acc + 1
            case _ ⇒ acc
          }
        }
        collectStats0(rest,doclines,codelines + codels)
```

The documentation block code is similiar, again with an accumulator for the lines:

⟨*Sample block filters*⟩ + ≡

```
            case markup.DocuBlock(_,_,lines) ⇒
              val doculs = (lines foldLeft 0) {
                (acc: Int, l: markup.Line) ⇒ l match {
                  case markup.NewLine ⇒ acc + 1
                  case _ ⇒ acc
                }
              }
              collectStats0(rest,doclines + doculs, codelines)
          }
      }
      collectStats0(bs,0,0)
    }
  }
```

Invoking this filter is as easy as calling

```
sweave -bfilter util.stats somefile.nw -o somefile.tex
```