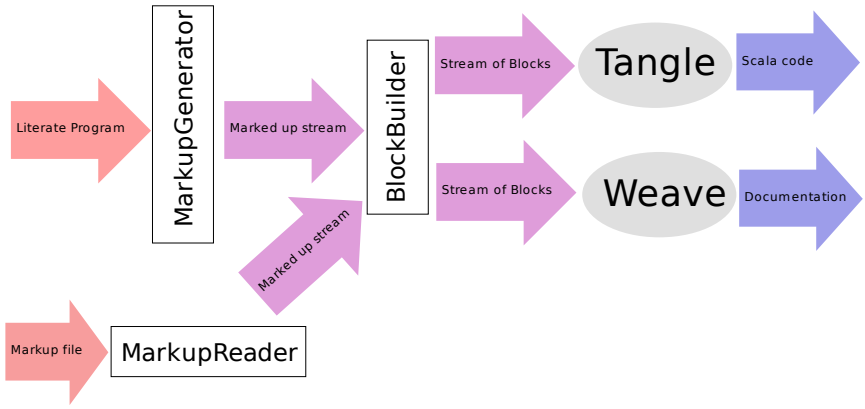
# 1 The noweb markup format

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
}
```

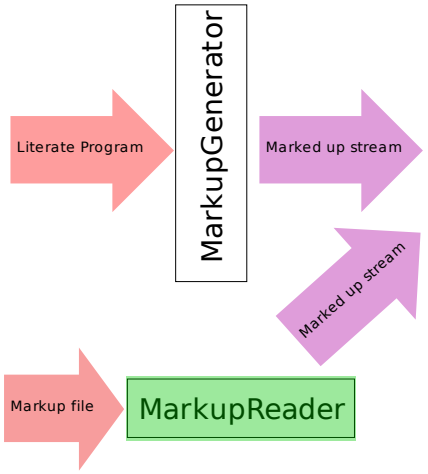
**Special indicators** We will have two special indicators: One telling us which file we are reading and one to indicate the end of our input.

⟨*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*⟩ ≡

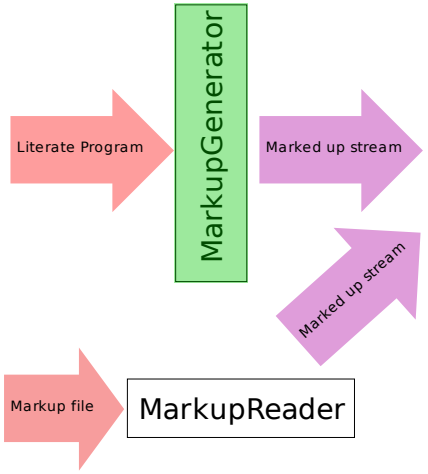
```
object MarkupReader {
  import java.io.{FileReader,BufferedReader,Reader}
  import java.io.InputStreamReader

  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 $\langle\langle$. 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
    }
  }
}
```