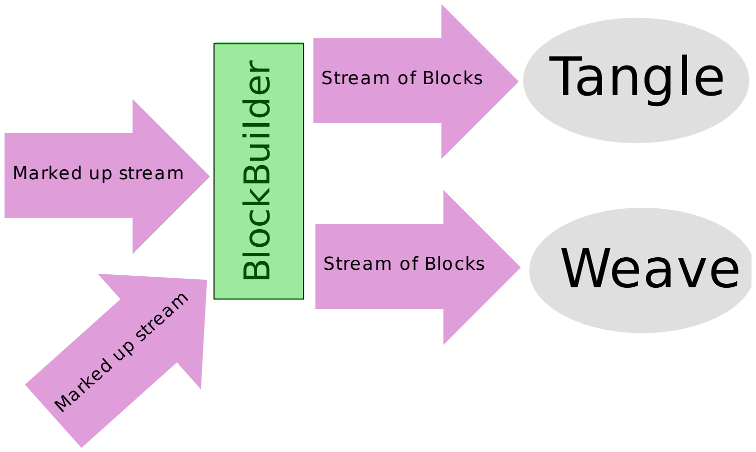
1 Extracting blocks from marked up files

While the markup intermediary format[1] 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.



⟨*⟩ ≡

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

1.1 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

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

- 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]
```

1.1.1 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)
    }
  }
```

1.1.2 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)
        }
      }
```

1.2 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))
        }
    }
```

1.3 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)
        }
    }
}
```