1 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>
```

1.1 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())
}
```

1.1.1 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)
  }
}
```

1.1.2 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
```

1.2 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]) {
}
```

1.2.1 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
```