1 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`

1.1 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 1.1.

⟨ *** ⟩ ≡

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

1.2 Puzzling code blocks together

1.2.1 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))
        }
}
```

1.2.2 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*⟩ + ≡

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

1.3 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*⟩ ≡

```
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 ⇒ ()
                                }
                        }
                }
        }
    }
}
```