A Simple Noweb-Like Tool Implemented in R

Ross Ihaka

September 3, 2013

1 Introduction

In a 1984 paper, Donald Knuth introduced the concept of *literate programming.*

> Knuth, Donald E. (1984). "Literate Programming." *The Computer Journal* **27**, 97–111.

He argued that there needed to be a change in the way that programmers view the way they work.

> "Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do."

Over the years, there have been a number of literate programming systems developed. One of the more popular ones is the `noweb` system developed by Norman Ramsey.

> Ramsey, Norman (1994). "Literate Programming Simplified." *IEEE Software*, **11**, 97–105.

The `noweb` system has the advantage of being very lightweight and of supporting a variety of programming languages.

One unfortunate aspect of `noweb` is that it depends on a tool chain that is not readily available on all platforms (and specifically not on Microsoft Windows). The `Rnoweb` program described by this document is an attempt aleviate this situation by providing a "`noweb`-like" tool, suitable for maintaining R code, that depends only on the R environment. It will thus be available to R programmers no matter what platform they work on. This tool is not intended to replace `noweb`, it is simply a stopgap measure for anyone who is unable able to run the real thing.

The R `noweb` function aims to produce output that is close enough to pin-compatible with `noweb` that the same LaTeX macros can be used to typeset the resulting document. There are some differences, however. These are described in a different document.

This code is made available under the terms of the Free Software Foundation GPL-3 License, with the option that licensees may distribute the code under any later version of the GPL license, should they choose to.

2 Lexical Structure and the Main Function

The `noweb` function provides the user-callable interface to this software. It is written in *closure* form to conceal the collection of helper functions used to help carry out the computations. `noweb` is the only user-visible component of this software.

2a ⟨*Rnoweb.R* 2a⟩≡
```
  ⟨comments and copyright 25b⟩
  noweb = local({
      ⟨constants 2c⟩
      ⟨chunk information assembly 11⟩
      ⟨document weaving 21b⟩
      ⟨code tangling 25a⟩
      ⟨main noweb function 2b⟩
  })
```
This code is written to file `Rnoweb.R`.

The `noweb` function manages the entire *weaving* and *tangling* process. It has a single argument, called `files`, that contains a vector of filenames to be processed. For each filename in this vector, `noweb` opens the file, reads its contents into memory and extracts the basic information in the chunks that make up the file. It then carries out the weaving and tangling process.

2b ⟨*main noweb function* 2b⟩≡
```
  function(files, weave = TRUE, tangle = TRUE,
           fullxref = FALSE) {
      for(fileno in seq(along = files)) {
          filename = files[fileno]
          lines = readLines(filename)
          info = extractChunkInfo(lines, filename, fileno)
          hash = buildHash(info)
          if (weave)
              weaveFile(lines, info, hash,
                        filename, fileno, fullxref)
          if (tangle)
              tangleFile(lines, info, hash,
                         filename, fileno)
      }
  }
```
Uses `extractChunkInfo` 4, `tangleFile` 22, and `weaveFile` 12a.

The following regular expressions are placed here and used throughout the code that follows. This ensures consistent usage and aids maintainability.

2c ⟨*constants* 2c⟩≡
```
  CHUNKSTART = "(^<<[-. 0-9A-Za-z]*>>=)|(^\\@ )|(^\\@$)"
  CHUNKREF   = "<<[-. 0-9A-Za-z]*>>"
```
Defines:
  `CHUNKREF`, used in chunks 6c, 9, 18, and 24.
  `CHUNKSTART`, used in chunk 5a.

3 Assembling Chunk Information

The `extractChunkInfo` function assembles all the information about code and documentation chunks to be used in the weaving and tangling steps that take place later in the noweb process. The information is returned in a list that has one component for each (code or documentation) chunk in the file. The elements of the list are themselves lists whose named components give information the chunk. The names of the components are as follows:

`number`:	The sequence number of the chunk.
`name`:	The (canonicalised) name of the chunk.
`label`:	The chunk label. There is a 1-1 correspondence between names and labels.
`sublabel`:	The chunk sublabel. Each chunk has a unique sublabel.
`start`:	The first line of the chunk (includes the header).
`end`:	The last line of the chunk.
`uses`:	A vector containing the names of the chunks referenced in this chunk.
`useslabels`:	A vector containing the sublabels of the chunks referenced in this chunk.
`usedin`:	A vector containing the sublabels of chunks that this chunk is used in.
`defines`:	A vector containing the identifiers defined in this chunk.
`usesdefs`:	A vector containing the identifiers defined in other chunks that are used in this chunk.

The function proceeds by first locating the start lines of every chunk. It also determines the last lines of each chunk. The values are stored in the *vectors* called `start` and `end`. The `start` values are used to extract the chunk headers which are then stored in a *vector* called `headers`.

The chunk names are then extracted from `headers` and stored in a *vector* called `name`. If a chunk is named it is a code chunk, otherwise it is a documentation chunk. This is used to create a *vector* called `type` that holds the type information as a character string.

Next the chunk labels and sublabels are constructed. First, a *vector* called `prefix` is constructed that contains strings that are unique to each chunk name (i.e. some prefixes may be the same). This is used to create the chunk labels, which correspond directly to the chunk names, and chunk sublabels which are unique to each chunk. The labels and sublabels are stored in *vectors* called `label` and `sublabel`.

Next, information on chunk usage is compiled. The vectors of names of chunks referenced within each chunk are determined and stored in the *list* called `uses`. The corresponding vectors of sublabels are stored in the *list* called `useslabels`. Vectors of the sublabels of chunks where each chunk is referenced are also compiled and stored in the *list* called `usein`.

Finally, the vectors of identifiers defined in each chunk and the vectors of identifiers used in chunk are determined. They are assembled into the *lists* called `defines` and `usesdefines`.

All this information is then assembled into a *list* of named component *lists*; one named component list for each chunk. It is this list of lists that is returned by `extractChunkInfo`.

4 ⟨*chunk information extraction* 4⟩≡
```
extractChunkInfo =
    function(lines, filename, fileno) {
        start = chunkStarts(lines)
        end = c(start[-1] - 1, length(lines))
        header = lines[start]
        name = chunkName(header)
        type = ifelse(name == "", "doc", "code")
        prefix = chunkLabelPrefix(name, filename, fileno)
        label = chunkLabel(prefix)
        sublabel = chunkSublabel(prefix)
        uses = chunkUsesChunks(lines, start, end, type)
        useslabels = chunkUsesLabels(uses, name, sublabel)
        usedin = chunkIsUsedInChunks(name, uses, sublabel)
        defines = chunkDefines(header, type)
        usesdefs = chunkUsesDefines(lines, start, end,
            type, defines)
        info = vector("list", length = length(start))
        for(i in 1:length(start))
            info[[i]] =
                list(number = i,
                     type = type[i],
                     name = name[i],
                     label = label[i],
                     sublabel = sublabel[i],
                     start = start[i],
                     end = end[i],
                     uses = uses[[i]],
                     useslabels = useslabels[[i]],
                     usedin = usedin[[i]],
                     defines = defines[[i]],
                     usesdefs = usesdefs[[i]])
        info
    }
```
Defines:
`extractChunkInfo`, used in chunk 2b.

Uses `chunkDefines` 8b, `chunkIsUsedInChunks` 8a, `chunkLabel` 6a, `chunkLabelPrefix` 5c, `chunkName` 5b, `chunkStarts` 5a, `chunkSublabel` 6b, `chunkUsesChunks` 7a, `chunkUsesDefines` 9, and `chunkUsesLabels` 7b.

The `extractChunkInfo` function calls a variety of helper functions to compute information about each of the chunks. These functions are described in the following paragraphs.

The `chunkStarts` function examines a file and detects which lines contain chunk starts. The regular expression it contains detects both code and documentation chunks.

5a ⟨*chunk starts* 5a⟩≡

```
chunkStarts =
    function(lines)
    grep(CHUNKSTART, lines)
```

Defines:
`chunkStarts`, used in chunk 4.
Uses `CHUNKSTART` 2c.

Given a vector of strings containing the header lines of code and documentation chunks, `chunkName` extracts the chunk names. All documentation chunks are given the name `""`.

5b ⟨*chunk name* 5b⟩≡

```
chunkName =
    function(chunkstart)
    ifelse(grepl("(^\\@\\s)|(^\\@$)", chunkstart), "",
           gsub("^\\s+", "",
                gsub("\\s*>>.*$", "",
                     gsub("^.*<<\\s*", "",
                          chunkstart))))
```

Defines:
`chunkName`, used in chunks 4, 18a, and 24.

The `chunkLabelPrefix` function takes a vector of chunk names and produces prefixes for the LaTeX labels used in cross-referencing. The label prefix consists of "`RNW`" with the first three letters of the file name and file number appended, followed by "`-`," followed by the first three letters of the chunk name followed, by the chunk number. This ensures that each chunk's label prefix is unique.

5c ⟨*chunk label prefix* 5c⟩≡

```
chunkLabelPrefix =
    function(name, filename, fileno) {
        number = as.numeric(factor(name, unique(name)))
        ifelse(name == "", "",
               paste("RNW",
                     substring(gsub(" ", "", filename), 1, 3),
                     fileno, "-",
                     substring(gsub(" ", "", name), 1, 3),
                     number,
                     sep = ""))
    }
```

Defines:
`chunkLabelPrefix`, used in chunk 4.

The `chunkLabel` function takes the chunk label prefixes and constructs LaTeX labels that are unique to a code chunk name. All blocks in a file with the same name share the same `label` value.

6a ⟨*chunk chunklabel* 6a⟩≡
```
chunkLabel =
    function(prefix)
    ifelse(prefix == "", "", paste(prefix, 1, sep = "-"))
```
Defines:
`chunkLabel`, used in chunk 4.

The `chunkSublabel` function takes the chunk label prefixes and constructs LaTeX labels that are unique to each code chunk. These are used to generate the margin tags in the LaTeX output.

6b ⟨*chunk sublabel* 6b⟩≡
```
chunkSublabel =
    function(prefix) {
        index = seq(along = prefix)
        fprefix = factor(prefix, unique(prefix))
        index[order(fprefix)] =
            sequence(table(fprefix))
        ifelse(prefix == "", "",
               paste(prefix, index, sep = "-"))
}
```
Defines:
`chunkSublabel`, used in chunk 4.

The next two functions handle chunk inclusion. The first is a helper function that extracts the names of all chunks included in the code chunk passed in as its argument and returns them in a vector.

The function works as follows. First, the lines containing chunk references are extracted. Next, the material between inclusions (including `>>` and `<<` are turned into newlines. Finally the starting material (incuding `<<`) and terminal material (including `>>`) are removed. The remaining lines are split on newlines and the results turned into a character vector.

6c ⟨*chunk uses chunks* 6c⟩≡
```
includedChunks =
    function(chunk)
    unlist(strsplit(sub(">>.*?$", "",
                        sub(".*?<<", "",
                            gsub(">>.*?<<", "\n",
                                 grep(CHUNKREF,
                                      chunk,
                                      value = TRUE)
                                 ))), "\n"))
```
Defines:
`includedChunks`, used in chunk 7a.
Uses `CHUNKREF` 2c.

The second function uses the first to build a list of character vectors that contain the names of all the inclusions in each code chunk. In the case of documentation chunks, the vector of included names is `character(0)`.

7a ⟨*chunk uses chunks* 6c⟩+≡

```
chunkUsesChunks =
    function(lines, start, end, type) {
        uses = vector("list", length(start))
        for(i in 1:length(start))
            if (type[i] == "doc" || start[[i]] >= end[[i]])
                uses[[i]] = character(0)
            else
                uses[[i]] =
                    includedChunks(lines[(start[i] + 1):end[i]])
        uses
    }
```

Defines:
`chunkUsesChunks`, used in chunk 4.
Uses `includedChunks` 6c.

The `chunkUsesLabels` function maps the names, obtained by calling the previous function, into their corresponding sublabels. This is where the detection of undefined code chunk references takes place.

7b ⟨*chunk uses labels* 7b⟩≡

```
chunkUsesLabels =
    function(uses, names, sublabels)
    lapply(uses,
           function(u) {
               labels = sublabels[match(u, names)]
               if (any(is.na(labels)))
                   stop(paste("undefined chunk: <<",
                              u[is.na(labels)][1], ">>", sep = ""),
                        call. = FALSE)
               labels
           })
```

Defines:
`chunkUsesLabels`, used in chunk 4.

Given the vector of chunk names and the list of uses computed by `chunkUsesChunks`, this function computes the set of sublabels of the chunks where each chunk name is referenced.

8a ⟨*chunk is used in chunks* 8a⟩≡
```
chunkIsUsedInChunks =
    function(names, uses, sublabel) {
        usedin = vector("list", length(names))
        for(i in 1:length(names)) {
            usedin[[i]] = sublabel[sapply(uses,
                          function(u) any(u == names[i]))]
        }
        usedin
    }
```
Defines:
`chunkIsUsedInChunks`, used in chunk 4.

This function creates a list containing vectors of character strings giving the identifiers listed in the "`@ %def`" statment following each code block. The variable "`header`" contains all the chunk header lines and "`type`" contains their types. The identofiers are sorted alphabetically within each chunk.

8b ⟨*chunk defines* 8b⟩≡
```
chunkDefines =
    function(header, type) {
        defs = c(header[-1], "")
        defs = ifelse(grepl("^@\\s+%def\\s+", defs), defs, "")
        defs = ifelse(type == "code" &
                      c(type[-1], "code") == "doc",
            sub("\\s*$", "",
                sub("^@\\s+%def +", "", defs)), "")
        lapply(strsplit(defs, "\\s+"), sort)
    }
```
Defines:
`chunkDefines`, used in chunk 4.

For each chunk, this function determines which defines from other chunks are used in this chunk. The function returns a list of character vectors. The uses in documentation blocks are set to `character(0)`.

The header line of the chunk has been removed in the caller and chunk references are removed by the first `grep` statement. This means that spurious identifier matches in chunk names should be eliminated.

There is also work that needs to be done to avoid spurious identifier in strings and comments. The is done by first emptying string contents and then stripping comments stripped before matching the identofiers. (The order in which these operations is carried out is important.)

The reason that strings must be emptied first is that there may be a comment character “`#`” within a string. Emptying the strings first means that any comment character that remains must start a comment. There remains an issue with strings of the left-hand side of assignments, however. Even parsing the chunks with R and walking the resulting parse tree extracting symbol names would not work in this case.

One issue remains. Weird identifiers can be quoted with backticks `‘‘`. If such identifiers contain “`#`” then they will be mangled by the comment removal mechanism above and will not appear in the cross-listing of variable uses. Of course, anyone including “`#`” within an identifier deserves everything that fate throws at them.

9 ⟨*chunk uses defines* 9⟩≡
```
chunkUsesDefines =
    function(lines, start, end, type, define) {
        usesdefs = vector("list", length(start))
        for(i in 1:length(start)) {
            if (type[i] == "code") {
                chunk = grep(CHUNKREF,
                    chunkContent(lines, start[i], end[i]),
                    value = TRUE, invert = TRUE)
                chunk = gsub("#.*", "",
                    gsub("’.*?’", "’’",
                         gsub(’".*?"’, ’""’,
                              gsub(’\\\\"|\\\\\’’, "",
                                   chunk))))
                defs = unlist(define[-i])
                pats = paste("(^|[^a-zA-Z0-9._])",
                    gsub("\\.", "\\\\.", defs),
                    "($|[^a-zA-Z0-9._])", sep = "")
                usesdefs[[i]] = sort(defs[sapply(pats,
                               function(p)
                               any(grepl(p, chunk)))])
            }
            else usesdefs[[i]] = character()
        }
        usesdefs
    }
```
Defines:
`chunkUsesDefines`, used in chunk 4.
Uses `chunkContent` 10a and `CHUNKREF` 2c.

Given a chunk starting at line “`start`” and ending at line “`end`” and including a chunk header line, this function extracts the chunk contents (i.e. all but the first line of the chunk).

10a ⟨*chunk content* 10a⟩≡

```
chunkContent =
    function(lines, start, end)
    if (start < end) lines[(start + 1):end] else character(0)
codeChunkContent =
    function(lines, start, end)
    if (start < end) lines[(start + 1):end] else character(0)
docChunkContent =
    function(lines, start, end)
    c(sub("^@ ", "", lines[start]),
      if (start < end) lines[(start + 1):end] else character(0))
```

Defines:
`chunkContent`, used in chunks 9 and 15.
`codeChunkContent`, never used.
`docChunkContent`, used in chunk 14a.

The following function builds a hash table that provides a fast way of looking up chunk labels and sublabels given the chunk name. This is used when processing code lines containing chunk references. It could probably be used more widely.

10b ⟨*build chunk hash table* 10b⟩≡

```
buildHash =
    function(info) {
        hash = new.env(parent = emptyenv())
        for(chunk in info)
            with(chunk,
                 if (name != "")
                     assign(name,
                            list(label = label,
                                 sublabel = sublabel),
                            envir = hash))
        hash
    }
```

The functions in this section are assembled together (by concatenation) as follows.

11 ⟨*chunk information assembly* 11⟩≡
  ⟨*chunk starts* 5a⟩
  ⟨*chunk name* 5b⟩
  ⟨*chunk label prefix* 5c⟩
  ⟨*chunk chunklabel* 6a⟩
  ⟨*chunk sublabel* 6b⟩
  ⟨*chunk uses chunks* 7a⟩
  ⟨*chunk uses labels* 7b⟩
  ⟨*chunk is used in chunks* 8a⟩
  ⟨*chunk defines* 8b⟩
  ⟨*chunk uses defines* 9⟩
  ⟨*chunk content* 10a⟩
  ⟨*chunk information extraction* 4⟩
  ⟨*build chunk hash table* 10b⟩

3.1 Document Weaving

Once the information in a noweb file has been analysed, the file is *woven* to produce produce a LaTeX document describing the software components in the file. The weaving process is carried out by the sequence of functions listed in this section.

The following function processes a single source file. It opens the file that will contain the woven output. Next it processes any initial non-chunk (as documentation) by calling `weaveInitial` and then processes the documentation and code chunks as they appear in the input file. The actual work of processing the chunks is carried out in the specialised `weaveDoc` and `weaveCode` functions.

Immediately after the last code chunk has been processed all indexing information is processed and written to the output file. The indexing is done separately for chunks and identifiers.

12a ⟨*file weaving* 12a⟩≡
```
weaveFile =
    function(lines, info, hash, filename, fileno, fullxref) {
        lastcode = max(which(sapply(info,
            function(i) i$type) == "code"))
        file = openWeaveFile(filename)
        start = info[[1]]$start
        if (start > 1)
            weaveInitial(lines[1:(start - 1)], file)
        for (i in seq(along = info)) {
            if (i == 1) weaveFilename(filename, file)
            if (info[[i]]$type == "doc")
                weaveDoc(lines, info[[i]], file)
            else
                weaveCode(lines, info[[i]], hash, file)
            if (i == lastcode) {
                weaveChunkIndex(info, file, fullxref)
                weaveIdentifierIndex(info, file)
            }
        }
        weaveNewline(file = file)
        close(file)
    }
```
Defines:
`weaveFile`, used in chunk 2b.
Uses `openWeaveFile` 12b, `weaveCode` 15, `weaveDoc` 14a, `weaveFilename` 13a, `weaveInitial` 13c, and `weaveNewline` 17d.

12b ⟨*file weaving support* 12b⟩≡
```
openWeaveFile =
    function(filename) {
        texfilename = sub("^.*/", "",
            sub("\\.[^.]*$", ".tex", filename))
        file(texfilename, "w")
    }
```
Defines:
`openWeaveFile`, used in chunk 12a.

The `weaveFilename` function sets the name of the file being processed when the first code or document chunk is encountered.

13a ⟨*file weaving support* 12b⟩+≡
```
weaveFilename =
    function(filename, file)
    cat("\\nwfilename{", filename, "}", sep = "",
        file = file)
```
Defines:
`weaveFilename`, used in chunk 12a.

The `weaveDocLine` prints lines in both `weaveInitial` and `weaveDoc`. The function handles code fragments quoted with double brackets. This is just a crude approximation to *noweb*'s behaviour. It uses a non-greedy regular expression to match the quotation delimiters. This means that such code fragments may not contain either of the code quotation delimiters. It is also not permissible to have newlines within quoted code. These restrictions could be eliminated by properly parsing the documentation lines. I'm just not sure that it is worth the effort.

The function surrounds the quoted fragment with `\verb?...?`. This means the world will end if a user puts a question mark inside quoted text.

13b ⟨*documentation line cleaning* 13b⟩≡
```
weaveDocLine =
    function(line)
    gsub("\\[\\[(.*?)\\]\\]", "\\\\verb?\\1?", line)
```
Defines:
`weaveDocLine`, used in chunks 13c and 14a.

Some noweb files start with an initial undeclared chunk. This is taken to be a documentation chunk but is processed differently; no LaTeX start- and end-of-chunk directives are emitted. The `weaveInitial` function handles any initial non chunk.

13c ⟨*weave an initial nonchunk* 13c⟩≡
```
weaveInitial =
    function(lines,  file) {
        for(i in 1:length(lines))
            cat(weaveDocLine(lines[i]), "\n",
                sep = "", file = file)
    }
```
Defines:
`weaveInitial`, used in chunk 12a.
Uses `weaveDocLine` 13b.

Processing of documentation chunks is very simple. LaTeX start- and end-of-chunk delimiters are emitted and any quoted code within the chunk is turned into verbatim text. (This could be replaced by the *noweb* quoted code mechanism.)

14a ⟨*weave a documentation chunk* 14a⟩≡
```
weaveDoc =
    function(lines, info, file) {
        with(info, {
            chunk = docChunkContent(lines, start, end)
            weaveBeginDoc(number, file, chunk[1] == "")
            for(i in seq(along = chunk))
                cat(weaveDocLine(chunk[i]), "\n",
                    sep = "", file = file)
            weaveEndDoc(file)
        })
    }
```
Defines:
  `weaveDoc`, used in chunk 12a.
Uses `docChunkContent` 10a, `weaveBeginDoc` 14b, `weaveDocLine` 13b, and `weaveEndDoc` 14c.

The `weaveBeginDoc` and `weaveEndDoc` functions are invoked at the start and end of each documentation chunk. The first function specifies the chunk number and sets up a mode suitable for typesetting documentation; the second provides a bracketing close for documentation. If a documentation chunk starts with a blank line then a paragraph break is output at the start of the chunk.

14b ⟨*output support for documentation chunk weaving* 14b⟩≡
```
weaveBeginDoc =
    function(n, file, parbreak = FALSE) {
        cat("\\nwbegindocs{", n, "}", sep = "", file = file)
        if(parbreak)
            cat("\\nwdocspar ", sep = "", file = file)
    }
```
Defines:
  `weaveBeginDoc`, used in chunk 14a.

14c ⟨*output support for documentation chunk weaving* 14b⟩+≡
```
weaveEndDoc =
    function(file)
    cat("\\nwenddocs{}", file = file)
```
Defines:
  `weaveEndDoc`, used in chunk 14a.

The following function weaves a code chunk. It is complicated because of all the cross-referencing of code chunks that takes place.

15 ⟨*weave a code chunk* 15⟩≡
```
weaveCode =
    function(lines, info, hash, file) {
        with(info, {
            chunk = chunkContent(lines, start, end)
            unused = length(usedin) == 0
            weaveBeginCode(number, name, label, sublabel, file)
            if (length(usedin) == 0)
                weaveNotUsedHeader(file)
            weaveNewline(file)
            for(i in seq(along = chunk))
                if (containsInsert(chunk[i]))
                    weaveInsert(chunk[i], hash, file)
                else
                    weaveCodeLine(chunk[i], file)
            if (number == 1)
                cat("\\nosublabel{", sublabel, "-u4}",
                    sep = "", file = file)
            weaveDefines(defines, sublabel, file)
            weaveDefineUses(usesdefs, sublabel, file)
            if (length(usedin) == 0)
                weaveNotUsedChunk(name, file)
            ##  if (notused) notusedfile(name, file)
            weaveEndCode(file)
            })
    }
```
Defines:
`weaveCode`, used in chunk 12a.

Uses `chunkContent` 10a, `containsInsert` 18b, `weaveBeginCode` 16a, `weaveCodeLine` 19, `weaveDefines` 16c, `weaveDefineUses` 17a, `weaveEndCode` 17c, `weaveInsert` 18a, `weaveNewline` 17d, `weaveNotUsedChunk` 17b, and `weaveNotUsedHeader` 16b.

The next series of function provides support for typesetting code chunks. This is much more complex than the code for documentation chunks because the cross-referencing of chunk names and identifiers defined in the chunk.

If the `sublabel` for a chunk is not the same as the `label` then this is not the first chunk with this name. In such a case, the terminating directive is `\plusendmoddef` rather than `\endmoddef`.

16a ⟨*output support for code chunk weaving* 16a⟩≡

```
weaveBeginCode =
    function(n, name, label, sublabel, file)
    cat(paste("\\nwbegincode{", n, "}",
              "\\sublabel{", sublabel, "}",
              "\\nwmargintag{{\\nwtagstyle{}\\subpageref{",
              sublabel, "}}}", "\\moddef{", name,
              "~{\\nwtagstyle{}\\subpageref{", label, "}}}\\",
              if(sublabel != label) "plus" else "",
              "endmoddef", sep = ""),
        file = file)
```

Defines:
`weaveBeginCode`, used in chunk 15.

16b ⟨*output support for code chunk weaving* 16a⟩+≡

```
weaveNotUsedHeader =
    function(file)
    cat("\\let\\nwnotused=\\nwoutput{}", file = file)
```

Defines:
`weaveNotUsedHeader`, used in chunk 15.

16c ⟨*output support for code chunk weaving* 16a⟩+≡

```
weaveDefines =
    function(defines, sublabel, file)
    if (length(defines) > 0) {
        cat(paste("\\nwindexdefn{", defines, "}{",
                  defines, "}{", sublabel, "}", sep = ""),
            "\\eatline\n", sep = "", file = file)
        cat("\\nwidentdefs{",paste("\\\\{{", defines, "}{",
                                   defines,"}}", sep = ""),
            "}", sep = "", file = file)
    }
```

Defines:
`weaveDefines`, used in chunk 15.

17a ⟨*output support for code chunk weaving* 16a⟩+≡
```
weaveDefineUses =
    function(usesdefs, sublabel, file) {
        if (length(usesdefs) > 0) {
            cat("\\nwidentuses{",
                paste("\\\\{{", usesdefs, "}{",
                      usesdefs, "}}", sep = ""),
                "}", sep = "", file = file)
            cat(paste("\\nwindexuse{",
                      usesdefs, "}{", usesdefs,
                      "}{", sublabel, "}", sep = ""),
                sep = "", file = file)
            }
        }
```
Defines:
`weaveDefineUses`, used in chunk 15.

17b ⟨*output support for code chunk weaving* 16a⟩+≡
```
weaveNotUsedChunk =
    function(name, file)
    cat(paste("\\nwnotused{", name, "}", sep = ""),
        file = file)
```
Defines:
`weaveNotUsedChunk`, used in chunk 15.

17c ⟨*output support for code chunk weaving* 16a⟩+≡
```
weaveEndCode =
    function(file)
    cat("\\nwendcode{}", file = file)
```
Defines:
`weaveEndCode`, used in chunk 15.

17d ⟨*output support for code chunk weaving* 16a⟩+≡
```
weaveNewline =
    function(file)
    cat("\n", file = file)
```
Defines:
`weaveNewline`, used in chunks 12a and 15.

The `weaveInsert` function weaves lines that contain chunk references. It works as follows:

```
While there are still references to process in the line.
    Split off any text that preceeds the first reference and print it.
    Remove the text from the start of the line.
    Split off the reference that now starts the line and and process it.
    Remove the reference from the start of the line.
Print any text that is left in the line.
```

18a ⟨*output support for code chunk weaving* 16a⟩+≡

```
weaveInsert =
    function(line, hash, file) {
        while(grepl(CHUNKREF, line)) {
            text = sub("(^|[^@])<<.*$", "\\1", line)
            cat(text, file = file)
            line = substring(line, nchar(text) + 1, nchar(line))
            text = sub("<<([-. 0-9A-Za-z]*)>>.*$", "\\1", line)
            name = chunkName(text)
            sublabel = get(name, envir = hash)$sublabel
            cat("\\LA{}", name,
                "~{\\nwtagstyle{}\\subpageref{",
                sublabel, "}}\\RA{}", sep = "", file = file)
            line = substring(line, nchar(text) + 5, nchar(line))
        }
        cat(line, "\n", sep = "", file = file)
    }
```

Defines:
  `weaveInsert`, used in chunk 15.
Uses `chunkName` 5b and `CHUNKREF` 2c.

18b ⟨*contains insert* 18b⟩≡

```
containsInsert =
    function(line)
    grepl(CHUNKREF, line)
```

Defines:
  `containsInsert`, used in chunk 15.
Uses `CHUNKREF` 2c.

The `weaveCodeLine` function processes lines of code and writes them to the output file. The function maps "`{`" and "`}`" to "`\{`" and "`\}`" and "`\`" to "`\\`." In addition it strips the protecting "`@`" from "`@@<<`."

19 ⟨*output support for code chunk weaving* 16a⟩+≡
```
weaveCodeLine =
    function(line, file)
    cat(gsub("\\{", "\\\\{",
             gsub("\\}", "\\\\}",
                  gsub("\\\\", "\\\\\\\\",
                       gsub("@<<", "<<",
                            line)))), "\n",
        sep = "", file = file)
```
Defines:
`weaveCodeLine`, used in chunk 15.

The next section of code deals with indexing. (This is where the work carried out in `extractChunkInfo` pays off.) There are two kinds of indexing information. The chunk index contains information about where chunks are defined and where they are used. The identifier index contains information about where indentifiers are defined and where they are used.

The chunk index is generated from lines that have the form:

`\nwixlogsorted{c}{{`*name*`}{`*sublabel*`}{ ...}`

where *name* is the chunk name, *sublabel* is the chunk sublabel and ... is a list of elements of the form

`\nwixd{`*sublabel*`}`

indicating where the chunk was defined, or of the form

`\nwixu{`*sublabel*`}`

indicating locations where the chunk was used.

The function that produces these lines is shown below. It is invoked immediately after the last code chunk is processed. (It could be output anywhere during the weaving process, but this is where `noweb` does it.)

20 ⟨*weave chunk index* 20⟩≡

```
weaveChunkIndex =
    function(info, file, fullxref) {
        code = sapply(info, function(i) i$type) == "code"
        name = sapply(info, function(i) i$name)[code]
        sublabel = sapply(info, function(i) i$sublabel)[code]
        usedin = lapply(info, function(i) i$usedin)[code]
        o = order(name)
        name = name[o]
        sublabel = sublabel[o]
        usedin = sapply(usedin[o],
            function(u) {
                if (length(u) == 0) "" else
                paste("\\nwixu{", u, "}",
                      sep = "", collapse = "")
                })
        cat("\n", file = file)
        if (fullxref)
            o = TRUE
        else
            o = !duplicated(name)
        cat(paste("\\nwixlogsorted{c}{{", name[o],
                  "}{", sublabel[o], "}{\\nwixd{",
                  sublabel[o], "}",
                  usedin[o], "}}%\n", sep = ""),
            sep = "", file = file)
    }
```

The indentifier index is much simpler because information about indentifier definition and use is produced as the chunks are processed. The identifier index is produced from lines that have the form

`\nwixlogsorted{i}{{`*identifier*`}{`*identifier*`}}%`

The code that produces the lines is shown below.

21a ⟨*weave identifier index* 21a⟩≡

```
weaveIdentifierIndex =
    function(info, file) {
        code= sapply(info, function(i) i$type) == "code"
        defines = lapply(info, function(i) i$defines)[code]
        if (length(defines) > 0) {
            defines = unique(sort(unlist(defines)))
            cat(paste("\\nwixlogsorted{i}{{", defines,
                      "}{", defines, "}}%\n", sep = ""),
                sep = "", file = file)
        }
    }
```

The components of the weaving process are assembled together by concatenation as follows.

21b ⟨*document weaving* 21b⟩≡

⟨*file weaving* 12a⟩
⟨*file weaving support* 13a⟩
⟨*documentation line cleaning* 13b⟩
⟨*weave an initial nonchunk* 13c⟩
⟨*weave a documentation chunk* 14a⟩
⟨*contains insert* 18b⟩
⟨*weave a code chunk* 15⟩
⟨*output support for documentation chunk weaving* 14c⟩
⟨*output support for code chunk weaving* 19⟩
⟨*weave chunk index* 20⟩
⟨*weave identifier index* 21a⟩

3.2 Code Tangling

The function `tangleFile` builds a "chunk table," called `chunktable`, which is an environment providing a map from canonical chunk name to the contents of that (concatenated) chunk. It then writes out, to an approppriate file, any chunks which are not referenced by any other chunk, with references to other chunks expanded recursively. Note that files are only created if the chunk name contains nothing other than letters digits and ".". In particular, if there are spaces in the chunk name then no file is written and a warning is issued.

22 ⟨*tangle file* 22⟩≡

```
tangleFile =
    function(lines, info, hash, filename, fileno) {
        chunktable = new.env(parent = emptyenv())
        code = which(sapply(info,
                            function(i) i$type) == "code")
        for(i in code)
            with(info[[i]],
                storeChunk(name,
                           if (start == end) character(0)
                           else lines[(start+1):end],
                           chunktable))
        unusedChunks = code[sapply(info[code],
            function(i) length(i$used) == 0)]
        for(i in unusedChunks)
            with(info[[i]], {
                if (grepl("^[A-Za-z0-9.]+$", name)) {
                    filecon = file(name, "w")
                    expandChunk(name, "", chunktable, filecon)
                    close(filecon)
                }
                else
                    warning(paste("unreferenced chunk <<",
                                  name, ">> not output",
                                  sep = ""),
                            call. = FALSE)
            })
    }
```

Defines:
  `tangleFile`, used in chunk 2b.
Uses `expandChunk` 24 and `storeChunk` 23a.

The `storeChunk` function stores a code chunk in the chunk table. If there is already a chunk with this name in the table then the chunk is concatenated onto the end of it.

23a ⟨*store chunk* 23a⟩≡

```
storeChunk =
    function(name, lines, chunktable) {
        if (exists(name, chunktable, inherits = FALSE))
            lines = c(fetchChunk(name, chunktable), lines)
        assign(name, lines, envir = chunktable)
    }
```

Defines:
`storeChunk`, used in chunk 22.
Uses `fetchChunk` 23b.

The `fetchChunk` function fetches a chunk with the given name from the chunk table. This needs bulletproofing. (E.g. what happens if we ask for a chunk that is not in the chunk table.)

23b ⟨*fetch chunk* 23b⟩≡

```
fetchChunk =
    function(name, chunktable)
    get(name, envir = chunktable)
```

Defines:
`fetchChunk`, used in chunks 23a and 24.

The `expandChunk` function outputs a chunk with the given name and specified indent (a string of spaces). References to other chunks are expanded recursively. The method of splitting lines into program and chunk references is identical to that used in `weaveInsert`.

The treatment of leading spaces and trailing newlines is complicated by the fact there can be multiple chunks on a single line of the source file. The `expandChunk` function does not indent the first line of a chunk and does not print a trailing newline. Instead, this is left to the code that called `expandChunk`. Operating in this way makes it possible to handle the multiple chunks per line case.

24 ⟨*expand and output chunk* 24⟩≡

```
expandChunk =
    function(name, indent, chunktable, file) {
        chunk = fetchChunk(name, chunktable)
        nline = length(chunk)
        for(i in seq(along = chunk)) {
            line = chunk[i]
            space = sub("^(\\s*).*", "\\1", line)
            line = sub("^\\s*", "", line)
            if (i > 1) cat(indent, file = file)
            cat(space, file = file)
            if (grepl(CHUNKREF, line)) {
                while(grepl(CHUNKREF, line)) {
                    text = sub("(^|[^@])<<.*$", "\\1", line)
                    line = substring(line, nchar(text) + 1,
                        nchar(line))
                    cat(sub("@<<", "<<", text),
                        sep = "", file = file)
                    text = sub("<<([-. 0-9A-Za-z]*)>>.*$",
                        "\\1", line)
                    line = substring(line, nchar(text) + 5,
                        nchar(line))
                    name = chunkName(text)
                    expandChunk(name,
                                paste(indent, space, sep = ""),
                                chunktable, file)
                }
            }
            cat(gsub("@<<", "<<", line),
                sep = "", file = file)
            if (i < nline) cat("\n", file = file)
        }
    }
```

Defines:
`expandChunk`, used in chunk 22.
Uses `chunkName` 5b, `CHUNKREF` 2c, and `fetchChunk` 23b.

25a ⟨*code tangling* 25a⟩≡
  ⟨*tangle file* 22⟩
  ⟨*store chunk* 23a⟩
  ⟨*fetch chunk* 23b⟩
  ⟨*expand and output chunk* 24⟩

25b ⟨*comments and copyright* 25b⟩≡

```
### Copyright Ross Ihaka, 2011
###
### Distributed under the terms of GPL3, but may also be
### redistributed under any later version of the GPL.
###
### To be clear: If this code is included as part of an R
### distribution, even if that distribution is broken into
### component parts, all of distribution's parts must be
### made available under the terms of GPL3.
###
### (Suck on that you Revolution Analytics swine!)
###
### Literate Programming with and for R
###
### This function provides an R implementation of (a subset of)
### Norman Ramsey's noweb system.  It makes noweb available to
### anyone who has R (and LaTeX) installed on their system.
```

4 Chunk Index

5 Identifier Index