TANGLE und WEAVE mit R — selbstgemacht

Peter Wolf

Datei: `webR.rev`
Ort: `/home/wiwi/pwolf/R/work/relax/webR`

20.Sep.2006

1 Verarbeitungsprozesse

1 ⟨*process* 1⟩≡

```
notangle -Rdefine-tangleR         webR.rev > tangleR.R
notangle -Rdefine-tangleR         webR.rev > ../install.dir/relax/R/tangleR.R
notangle -Rdefine-tangleR-help webR.rev > ../install.dir/relax/man/tangleR.Rd
notangle -Rdefine-weaveR         webR.rev > weaveR.R
notangle -Rdefine-weaveR         webR.rev > ../install.dir/relax/R/weaveR.R
notangle -Rdefine-weaveR-help webR.rev > ../install.dir/relax/man/weaveR.Rd
notangle -Rdefine-weaveRhtml  webR.rev > ../install.dir/relax/R/weaveRhtml.R
notangle -Rdefine-weaveRhtml-help webR.rev > ../install.dir/relax/man/weaveRhtml.Rd
```

2 TEIL I — TANGLE

2.1 Problemstellung

In diesem Papier wird ein betriebssystemunabhängiges R-Programm für den TANGLE-Verarbeitungsprozeß beschrieben. Dieses kann Demonstrationsbeispielen beigefügt werden, außerdem kann für die Definition alternativer Verarbeitungsvorstellungen Anregungen geben.
In dem vorliegenden Vorschlag werden die verwendeten Modulnamen eines Quelldokumentes in den Code als Kommentarzeilen aufgenommen, so daß sie später für die Navigation verwendet werden können. Weiterhin werden alle Wurzeln aus dem Papier expandiert, sofern nicht eine andere Option angegeben wird.

2.2 Die grobe Struktur der Lösung

Der TANGLE-Prozeß soll mittels einer einzigen Funktion gelöst werden. Sie bekommt den Namen `tangleR`. Als Input ist der Name der Quelldatei zu übergeben. Nach dem Einlesen und der Aufbereitung des Quellfiles werden die Code-Chunks und die Stellen ihrer Verwendungen festgestellt. Dann werden Chunks mit dem Namen `start` und alle weiteren Wurzeln expandiert. Über Optionen läßt sich die Menge der zu expandierender Wurzeln bestimmen. Die Funktion besitzt folgende Struktur:

2 ⟨*define-tangleR* 2⟩≡
```
tangleR<-
function(in.file,out.file,expand.roots=NULL,expand.root.start=TRUE){
  # german documentation of the code:
  # look for file webR.pdf, P. Wolf 050204
  ⟨bereite Inhalt der Input-Datei auf tangleR 3⟩
  ⟨initialisiere Variable für Output tangleR 11⟩
  ⟨ermittle Namen und Bereiche der Code-Chunks tangleR 9⟩
  if(expand.root.start){
    ⟨expandiere Start-Sektion tangleR 12⟩
  }
  ⟨ermittle Wurzeln tangleR 14⟩
  ⟨expandiere Wurzeln tangleR 15⟩
  ⟨korrigiere ursprünglich mit @ versehene Zeichengruppen tangleR 17⟩
  ⟨speichere code.out tangleR 18⟩
}
```

2.3 Umsetzung der Teilschritte

2.3.1 Aufbereitung des Datei-Inputs

Aus der eingelesenen Input-Datei werden Text-Chunks entfernt und Definitions- und Verwendungszeilen gekennzeichnet.

3 ⟨*bereite Inhalt der Input-Datei auf* `tangleR` 3⟩≡
```
⟨lese Datei ein tangleR 4⟩
⟨entferne Text-Chunks tangleR 6⟩
⟨substituiere mit @ versehene Zeichengruppen tangleR 5⟩
⟨stelle Typ der Zeilen fest tangleR 7⟩
```

Die Input-Datei muß gelesen werden. Dieses werden zeilenweise auf `code.ch` abgelegt. `code.n` zeigt die aktuelle Zeilenzahl von `code.ch` an.

4 ⟨*lese Datei ein* `tangleR` 4⟩≡
```
if(!file.exists(in.file)) in.file<-paste(in.file,"rev",sep=".")
if(!file.exists(in.file)){
  cat(paste("ERROR:",in.file,"not found!!??\n"))
  return("Error in tangle: file not found")
}
# code.ch<-scan(in.file,sep="\n",what=" ")
code.ch<-readLines(in.file) # 2.1.0
```

5 ⟨*substituiere mit @ versehene Zeichengruppen* `tangleR` 5⟩≡
```
code.ch<-gsub("@>>","DoSpCloseKl-esc",gsub("@<<","DoSpOpenKl-esc",code.ch))
```

Text-Chunks beginnen mit einem `@`, Code-Chunks enden mit der Zeichenfolge `>>=`. Es werden die Nummern ersten Zeilen der Code-Chunks auf `code.a` abgelegt. `code.z` zeigt den Beginn von Text-Chunks an, weiter unten wird diese Variable die letzten Zeilen eines Code-Chunks anzeigen. Aus der Kumulation des logischen Vektor `change`, der die diese Übergänge anzeigt, lassen sich schnell die Bereiche der Text-Chunks ermitteln.

6 ⟨*entferne Text-Chunks* `tangleR` 6⟩≡
```
code.ch<-c(code.ch,"@")
code.a<- grep("^<<(.*)>>=",code.ch)
if(0==length(code.a)){return("Warning: no code found!!!!")}
code.z<-grep("^@",code.ch)
code.z    <-unlist(sapply(code.a ,function(x,y)min(y[y>x]),code.z))
code.n    <-length(code.ch)
change    <-rep(0,code.n); change[c(code.a ,code.z)]<-1
code.ch   <-code.ch[1==(cumsum(change)%%2)]
code.n    <-length(code.ch)
```

In dieser Implementation dürfen vor der Verwendung von Verfeinerungen Anweisungsteile stehen, nicht aber dahinter. Deshalb werden die Zeilen, die `<<` enthalten aufgebrochen. Sodann werden die Orte der Code-Chunk-Definitionen und Verwendungen festgestellt. Auf der Variable `line.typ` wird die Qualität der Zeilen von `code.ch` anzeigt: `D` steht für Definition, `U` für Verwendungen und `C` für normalen Code-Zeilen. `code.n` hält die Zeilenanzahl,

7 ⟨*stelle Typ der Zeilen fest* `tangleR` 7⟩≡
⟨*knacke ggf. Zeilen mit mehrfachen Chunk-Uses* `tangleR` 8⟩
```
line.typ  <-rep("C",code.n)
code.a    <-grep("cOdEdEf",code.ch)
code.ch[code.a]<-substring(code.ch[code.a],8)
line.typ[code.a]<-"D"
code.use    <-grep("uSeChUnK",code.ch)
code.ch[code.use]<-substring(code.ch[code.use],9)
line.typ[code.use]<-"U"
```

8 ⟨*knacke ggf. Zeilen mit mehrfachen Chunk-Uses* `tangleR` 8⟩≡
```
code.ch<-gsub("(.*)<<(.*)>>=(.*)","cOdEdEf\\2",code.ch)
repeat{
  if(0==length(cand<-grep("<<(.*)>>",code.ch))) break
  code.ch<-unlist(strsplit(gsub("(.*)<<(.*)>>(.*)",
             "\\1bReAkuSeChUnK\\2bReAk\\3",code.ch),"bReAk"))
}
code.ch<-code.ch[code.ch!=""]
code.n<-length(code.ch)
if(exists("DEBUG")) print(code.ch)
```

2.3.2 Ermittlung der Code-Chunks

Zur Erleichterung für spätere Manipulationen werden in den Bezeichnern die Zeichenketten `<<` `>>` bzw. `>>=` entfernt. Die Zeilennummern der Code-Chunks-Anfänge bezüglich `code.ch` stehen auf `code.a`, die Enden auf `code.z`.

9 ⟨*ermittle Namen und Bereiche der Code-Chunks* `tangleR` 9⟩≡
```
def.names<-code.ch[code.a]
use.names<- code.ch[code.use]
code.z<-c(if(length(code.a)>1) code.a[-1]-1, code.n)
code.ch<-paste(line.typ,code.ch,sep="")
if(exists("DEBUG")) print(code.ch)
```

Randbemerkung Zur Erleichterung der Umsetzung wurden in dem ersten Entwurf von `tangleR` mit Hilfe eines `awk`-Programms alle Text-Chunks aus dem Quellfile entfernt, so daß diese in der R-Funktion nicht mehr zu berücksichtigen waren. Dieses `awk`-Programm mit dem Namen `pretangle.awk` sei hier eingefügt, vielleicht ist es im Zusammenhang mit einer S-PLUS-Implementation hilfreich.

10 ⟨*ein* `awk`*-Programm zur Entfernung von Text-Chunks aus einem Quellfile* 10⟩≡

```
#
# Problemstellung: Vorverarbeitung fuer eigenes TANGLE-Programm
# Dateiname:       pretangle.awk
# Verwendung:      gawk -f pretangle.awk test.rev > tmp.rev
# Version:         pw 15.5.2000
#
BEGIN {code=0};
/^@/{code=0};
/<</{DefUse=2}
/>>=/{code=1;DefUse=1};
{
  if(code==0){next};
  if(code==1){
      if(DefUse==1){$0="D"$0}
      else{
        if(DefUse==2){$0="U"$0}
        else{$0="C"$0}
      };
      DefUse=0; print $0;
  }
}
```

2.3.3 Initialisierung des Outputs

Auf `code.out` werden die fertiggestellten Code-Zeilen abgelegt. Diese Variable muß initialisiert werden.

11 ⟨*initialisiere Variable für Output* `tangleR` 11⟩≡

```
code.out<-NULL
```

2.3.4 Expansion der Startsektion

Im REVWEB-System hat der Teilbaum unter der Wurzel `start` eine besondere Relevanz. Diesen gilt es zunächst zu expandieren. Dazu werden alle Chunks mit dem Namen `start` gesucht und auf dem Zwischenspeicher `code.stack` abgelegt. Dann werden normale Code-Zeilen auf die Output-Variablen übertragen und Verfeinerungsverwendungen werden auf `code.stack` durch ihre Definitionen ersetzt.

12 ⟨*expandiere Start-Sektion* `tangleR` 12⟩≡

```
if(exists("DEBUG")) cat("bearbeite start\n")
code.out<-c(code.out,"#0:","##start:##")
if(any(ch.no <-def.names=="start")){
   ch.no     <-seq(along=def.names)[ch.no]; rows<-NULL
   for(i in ch.no)
      if((code.a[i]+1)<=code.z[i]) rows<-c(rows, (code.a[i]+1):code.z[i])
   code.stack<-code.ch[rows]
   repeat{
     ⟨transferiere Startzeilen oder ersetze Verfeinerungen bis Ende erreicht tangleR 13⟩
   }
}
code.out<-c(code.out,"##:start##","#:0")
```

Falls `code.stack` leer ist, ist nichts mehr zu tun. Andernfalls wird die Anzahl der aufeinanderfolgenden Codezeilen festgestellt und auf die Output-Variable übertragen. Falls die nächste keine Codezeile ist, muß es sich um die Verwendung einer Verfeinerung handeln. In einem solchen Fall wird die nächste Verfeinerung identifiziert und der Bezeichner der Verfeinerung wird durch seine Definition ersetzt. Nicht definierte, aber verwendete Chunks führten anfangs zu einer Endlosschleife. Dieser Fehler ist inzwischen behoben `051219`. Eine entsprechende Änderung wurde auch für nicht-start-chunks fällig.

13 ⟨*transferiere Startzeilen oder ersetze Verfeinerungen bis Ende erreicht* `tangleR` 13⟩≡

```
if(0==length(code.stack))break
if("C"==substring(code.stack[1],1,1)){
  n.lines<-sum(cumprod("C"==substring(code.stack,1,1)))
  code.out<-c(code.out, substring(code.stack[1:n.lines],2))
  code.stack<-code.stack[-(1:n.lines)]
}else{
  if(any(found<-def.names==substring(code.stack[1],2))){
    found<-seq(along=def.names)[found]; rows<-NULL
    for(no in found){
      row.no<-c((code.a[no]+1),code.z[no])
      if(row.no[1]<=row.no[2]) rows<-c(rows,row.no[1]:row.no[2])
    }
    code.stack<-c(code.ch[rows],code.stack[-1])
    cat(found,", ",sep="")
  } else code.stack <-code.stack[-1] # ignore not defined chunks!
  # 051219
}
```

2.3.5 Ermittlung aller Wurzeln

Nach den aktuellen Überlegungen sollen neben `start` auch alle weiteren Wurzeln gesucht und expandiert werden. Wurzeln sind alle Definitionsnamen, die nicht verwendet werden.

14 ⟨*ermittle Wurzeln* `tangleR` 14⟩≡

```
root.no<-is.na(match(def.names,use.names))&def.names!="start"
root.no<-seq(along=root.no)[root.no]
roots  <-def.names[root.no]
if(!is.null(expand.roots)){
  h<-!is.na(match(roots,expand.roots))
  roots<-roots[h]; root.no<-root.no[h]
}
```

2.3.6 Expansion der Wurzeln

Im Prinzip verläuft die Expansion der Wurzel wie die von `start`. Jedoch werden etwas umfangreichere Kommentare eingebaut.

15 ⟨*expandiere Wurzeln* `tangleR` 15⟩≡
```
if(exists("DEBUG")) cat("bearbeite Sektion-Nr./Name\n")
for(r in seq(along=roots)){
  if(exists("DEBUG")) cat(root.no[r],":",roots[r],", ",sep="")
  row.no<-c((code.a[root.no[r]]+1),code.z[root.no[r]])
  if(row.no[1]<=row.no[2]){
    code.stack<-code.ch[row.no[1]:row.no[2]]
    code.out<-c(code.out,paste("#",root.no[r],":",sep=""),
                         paste("##",roots[r],":##",sep=""))
    repeat{
      ⟨transferiere Codezeilen oder ersetze Verfeinerungen bis Ende erreicht tangleR 16⟩
    }
    code.out<-c(code.out,paste("##:",roots[r],"##",sep=""),
                         paste("#:",root.no[r],sep=""))
  }
}
```

Die Abhandlung normaler Code-Zeilen ist im Prinzip mit der zur Expansion von `start` identisch. Bei einer Expansion von Verfeinerungsschritten sind jedoch noch die erforderlichen Beginn-/Ende-Kommentare einzusetzen.

16 ⟨*transferiere Codezeilen oder ersetze Verfeinerungen bis Ende erreicht* `tangleR` 16⟩≡
```
if(0==length(code.stack))break
if("C"==substring(code.stack[1],1,1)){
  n.lines<-sum(cumprod("C"==substring(code.stack,1,1)))
  code.out<-c(code.out, substring(code.stack[1:n.lines],2))
  code.stack<-code.stack[-(1:n.lines)]
}else{
  def.line<-substring(code.stack[1],2)
  if(any(found<-def.names==def.line)){
    code.stack<-code.stack[-1]
    found<-rev(seq(along=def.names)[found])
    for(no in found)
      row.no<-c((code.a[no]+1),code.z[no])
      if(row.no[1]<=row.no[2]){
        code.stack<-c(paste("C#"  ,no,":"  ,sep=""      ),
                      paste("C##" ,def.line,":##",sep=""),
                      code.ch[row.no[1]:row.no[2]] ,
                      paste("C##:",def.line, "##",sep=""),
                      paste("C#:" ,no      ,sep=""      ),
                      code.stack)
      }
  } else code.stack <-code.stack[-1] # ignore not defined chunks!
  # 051219
}
```

17 ⟨*korrigiere ursprünglich mit @ versehene Zeichengruppen* `tangleR` 17⟩≡
```
code.out<-gsub("DoSpCloseKl-esc",">>",gsub("DoSpOpenKl-esc","<<",code.out))
```

18 ⟨*speichere* `code.out` `tangleR` 18⟩≡

```
if(missing(out.file)||in.file==out.file){
  out.file<-sub("\\.([A-Za-z])*$","",in.file)
}
if(0==length(grep("\\.R$",out.file)))
  out.file<-paste(out.file,".R",sep="")
get("cat","package:base")(code.out,sep="\n",file=out.file)
cat("tangle process finished\n")
```

2.4 Beispiel

19 ⟨*Beispiel* – `tangleR` 19⟩≡

```
tangleR("out")
```

2.5 Help-Page

20 ⟨*define-tangleR-help* 20⟩≡

```
\name{tangleR}
\alias{tangleR}
%- Also NEED an '\alias' for EACH other topic documented here.
\title{ function to tangle a file }
\description{
  \code{tangleR} reads a file that is written according to
  the rules of the \code{noweb} system and performs a specific kind
  of tangling. As a result a \code{.R}-file is generated.
}
\usage{
tangleR(in.file, out.file, expand.roots = NULL,
expand.root.start = TRUE)
}
%- maybe also 'usage' for other objects documented here.
\arguments{
  \item{in.file}{ name of input file }
  \item{out.file}{ name of output file; if missing
  the extension of the input file is turned to \code{.R} }
  \item{expand.roots}{ name(s) of root(s) to be expanded; if NULL
  all will be processed }
  \item{expand.root.start}{ if TRUE (default) root chunk
  "start" will be expanded }
}
\details{
  General remarks: A \code{noweb} file consists of a mixture of text
  and code chunks. An \code{@} character (in column 1 of a line)
  indicates the beginning of a text chunk. \code{<<name of code chunk>>=}
  (starting at column 1 of a line) is a header line of a code chunk with
  a name defined by the text between \code{<<} and \code{>>=}.
  A code chunk is finished by the beginning of the next text chunk.
  Within the code chunk you can use other code chunks by referencing
  them by name ( for example by: \code{<<name of code chunk>>} ).
  In this way you can separate a big job in smaller ones.

  Special remarks: \code{tangleR} expands code chunk \code{start}
   if flag \code{expand.root.start} is TRUE. Code chunks will be surrounded
   by comment lines showing the number of the code chunk the code is
   coming from.
   If you want to use \code{<<} or \code{>>} in your code
   it may be necessary to escape them by an \code{@}-sign. Then
   you have to type in: \code{@<<} or \code{@>>}.
}
\value{
  usually a file with R code is generated
}
\references{ \url{http://www.eecs.harvard.edu/~nr/noweb/intro.html} }
\author{Hans Peter Wolf}

\seealso{  \code{\link{weaveR}} }
\examples{
\dontrun{
## This example cannot be run by examples() but should be work in an interactive R sessi
  tangleR("testfile.rev")
}
"tangleR(\"testfile.rev\")"
## The function is currently defined as
function(in.file,out.file,expand.roots=NULL,expand.root.start=TRUE){
# german documentation of the code:
# look for file webR.pdf, P. Wolf 050204
...
```

```
  }
}
\keyword{file}
\keyword{programming}
```

2.6 Ein Abdruck aus Verärgerung

Bei Übertragungsversuchen von R nach S-PLUS schien die Funktion `strsplit` zu fehlen, so dass sie mal grad entworfen wurde. Jedoch hätte man statt dessen die Funktion `unpaste` (!!) verwenden können. Wer hätte das gedacht?

21 ⟨*Definition einer unnötigen Funktion* 21⟩≡

```
if(!exists("strsplit"))
  strsplit<-function(x, split){
  # S-Funktion zum Splitten von Strings
  # Syntax wie unter R
  # pw16.5.2000
    out<-NULL; split.n<-nchar(split)
    for(i in x){
      i.n<-nchar(i)
      hh <-split==(h<-substring(i,1:(i.n+1-split.n),split.n:i.n))
      if(!any(hh)){out<-c(out,list(i));next}
      pos<-c(1-split.n,seq(along=hh)[hh])
      new<-unlist(lapply(pos,
          function(x,charvec,s.n) substring(charvec,x+s.n),i,split.n))
      anz<-diff(c(pos,length(h)+split.n))-split.n
      new<-new[anz>0];anz<-anz[anz>0]
      new<-unlist(lapply(seq(along=anz),
          function(x,vec,anz)substring(vec[x],1,anz[x]),new,anz))
      out<-c(out,list(new))
    }
    return(out)
  }
```

3 TEIL II — WEAVE

3.1 weaveR — eine einfache WEAVE-Funktion

In diesem Teil wird eine einfache Funktionen zum WEAVEN von Dateien beschrieben. Als Nebenbedingungen der Realisation sind zu nennen:

- Code-Chunk-Header müssen ganz links beginnen.
- Code-Chunk-Verwendungen müssen separat in einer Zeile stehen.
- Für eckige Klammern zum Setzen von Code im Text gelten folgende Bedingungen. Kommt in einer Zeile nur ein Fall *Code im Text* vor, dürfte es keine Probleme geben. Weiter werden auch Fällen, in denen die Code-Stücke keine Leerzeichen enthalten, selbst aber von Leerzeichen eingeschlossen sind, funktionieren.
- Eckige Klammern in Verbatim-Umgebungen werden nicht ersetzt.

Die Funktion besitzt folgenden Aufbau:

22 ⟨*define-weaveR* 22⟩≡
```
weaveR<-function(in.file,out.file){
  # german documentation of the code:
  # look for file webR.pdf, P. Wolf 050204, 060517
  ⟨initialisiere weaveR 23⟩
  ⟨lese Datei ein weaveR 24⟩
  ⟨substituiere mit @ versehene Zeichengruppen weaveR 25⟩
  ⟨stelle Typ der Zeilen fest weaveR 30⟩
  ⟨erstelle Output weaveR 36⟩
  ⟨ersetze Umlaute weaveR 26⟩
  ⟨korrigiere ursprünglich mit @ versehene Zeichengruppen weaveR 27⟩
  ⟨schreibe die Makrodefinition für Randnummern vor die erste Zeile 28⟩
  ⟨schreibe Ergebnis in Datei weaveR 29⟩
}
```

Zunächst fixieren wir die Suchmuster für wichtige Dinge. Außerdem stellen wir fest, ob R auf UTF-8-Basis arbeitet.

23 ⟨*initialisiere* weaveR 23⟩≡
```
pat.use.chunk<-paste("<","<(.*)>",">",sep="")
pat.chunk.header<-paste("^<","<(.*)>",">=",sep="")
pat.verbatim.begin<-"\\\\begin\\{verbatim\\}"
pat.verbatim.end<-"\\\\end\\{verbatim\\}"
pat.leerzeile<-"^(\\ )*$"
lcctype<-grep("LC_CTYPE",strsplit(Sys.getlocale(),";")[[1]],value=T)
UTF<-(1==length(grep("UTF",lcctype)))
UTF<- UTF | nchar(deparse("\xc3")) > 3
if(UTF) cat("character set: UTF\n") else cat("character set: ascii\n")
```

Die zu bearbeitende Datei wird zeilenweise auf die Variable `input` eingelesen.

24 ⟨*lese Datei ein* weaveR 24⟩≡
```
if(!file.exists(in.file)) in.file<-paste(in.file,"rev",sep=".")
if(!file.exists(in.file)){
  cat(paste("ERROR:",in.file,"not found!!??\n"))
  return("Error in weave: file not found")
}
# input<-scan(in.file,what="",sep="\n",blank.lines.skip = FALSE)
input<-readLines(in.file) # 2.1.0
length.input<-length(input)
```

25 ⟨*substituiere mit @ versehene Zeichengruppen* weaveR 25⟩≡
```
input<-gsub("@>>","DoSpCloseKl-esc",gsub("@<<","DoSpOpenKl-esc",input))
input<-gsub("@\\]\\]","DoEckCloseKl-esc",gsub("@\\[\\[","DoEckOpenKl-esc",input))
```

Umlaute sind ein Dauerbrenner. Hinweis: im richtigen Code steht unten übrigens: *äöüÄÖÜ* sowie in der ersten Zeile ein *ß*.

26 ⟨*ersetze Umlaute* `weaveR` 26⟩≡
```
if(!UTF){
 # im Tcl/Tk-Textfenster eingegeben -> iso-8859-1 (man iso-8859-1 / Latin1 / unicode
  input<-gsub("\283","",input)
  input<-chartr("\244\266\274\204\226\234\237","\344\366\374\304\326\334\337",input)
  # Latin1 -> TeX-Umlaute
  input<-gsub("\337","{\\\\ss}",input)
  input<-gsub("(\344|\366|\374|\304|\326|\334)","\\\\\"\\1",input)
  input<-chartr("\344\366\374\304\326\334","aouAOU",input)
}else{
  input<-gsub("\283\237","{\\\\ss}",input)
  input<-gsub("(\283\244|\283\266|\283\274|\283\204|\283\226|\283\234)",
                            "\\\\\"\\1",input)
  input<-chartr("\283\244\283\266\283\274\283\204\283\226\283\234",
                              "aouAOU", input)
}
cat("german Umlaute replaced\n")
```

Vor dem Wegschreiben müssen die besonderen Zeichengruppen zurückübersetzt werden.

27 ⟨*korrigiere ursprünglich mit @ versehene Zeichengruppen* `weaveR` 27⟩≡
```
input<-gsub("DoSpCloseKl-esc",">>",gsub("DoSpOpenKl-esc","<<",input))
input<-gsub("DoEckCloseKl-esc","]]",gsub("DoEckOpenKl-esc","[[",input))
```

28 ⟨*schreibe die Makrodefinition für Randnummern vor die erste Zeile* 28⟩≡
```
input[1]<-paste(
     "\\newcounter{Rchunkno}",
     "\\newcommand{\\makemarginno}{\\stepcounter{Rchunkno}",
     "\\rule{0mm}{0mm}\\\\\\hspace*{-3em}\\makebox[0mm]{",
     "\\arabic{Rchunkno}",
     "}\\hspace*{3em}}",
     input[1],sep="")
```

Zum Schluss müssen wir die modifizierte Variable `input` wegschreiben.

29 ⟨*schreibe Ergebnis in Datei* `weaveR` 29⟩≡
```
if(missing(out.file)||in.file==out.file){
  out.file<-sub("\\.([A-Za-z])*$","",in.file)
}
if(0==length(grep("\\.tex$",out.file)))
  out.file<-paste(out.file,".tex",sep="")
get("cat","package:base")(input,sep="\n",file=out.file)
cat("weave process finished\n")
```

Zu jeder Zeile wird ihr Typ festgestellt und auf dem Vektor `line.typ` eine Kennung vermerkt. Außerdem merken wir zu jedem Typ auf einer Variablen alle Zeilennummer des Typs. Wir unterscheiden:

Typ	Kennung	Indexvariable
Leerzeile	`EMPTY`	`empty.index`
Text-Chunk-Start	`TEXT-START`	`text.start.index`
Code-Chunk-Start	`HEADER`	`code.start.index`
Code-Chunk-Verwendungen	`USE`	`use.index`
normale Code-Zeilen	`CODE`	`code.index`
normale Textzeilen	`TEXT`	
Verbatim-Zeilen	`VERBATIM`	`verb.index`

Leerzeilen, Text- und Code-Chunk-Anfänge sind leicht zu finden. Code-Verwendungen sind alle diejenigen Zeilen, die << und >> enthalten, jedoch keine Headerzeilen sind. Am schwierigsten sind normale Code-Zeilen zu identifizieren. Sie werden aus den Code-Chunk-Anfängen und den Text-Chunkanfängen ermittelt, wobei die `USE`-Zeilen wieder ausgeschlossen werden. Alle übrigen Zeilen werden als Textzeilen eingestuft.

30 ⟨*stelle Typ der Zeilen fest* `weaveR` 30⟩≡
  ⟨*checke Leer-, Textzeilen* `weaveR` 31⟩
  ⟨*checke verbatim-Zeilen* `weaveR` 32⟩
  ⟨*checke Header- und Use-Zeilen* `weaveR` 33⟩
  ⟨*checke normale Code-Zeilen* `weaveR` 34⟩
  ⟨*belege Typ-Vektor* `weaveR` 35⟩

31 ⟨*checke Leer-, Textzeilen* `weaveR` 31⟩≡
```
empty.index<-grep(pat.leerzeile,input)
text.start.index<-which("@"==substring(input,1,1))
```

32 ⟨*checke verbatim-Zeilen* `weaveR` 32⟩≡
```
a<-rep(0,length.input)
a[grep(pat.verbatim.begin,input)]<-1
a[grep(pat.verbatim.end,input)]<- -1
a<-cumsum(a)
verb.index<-which(a>0)
```

33 ⟨*checke Header- und Use-Zeilen* `weaveR` 33⟩≡
```
code.start.index<-grep(pat.chunk.header,input)
use.index<-grep(pat.use.chunk,input)
use.index<-use.index[is.na(match(use.index,code.start.index))]
```

34 ⟨*checke normale Code-Zeilen* `weaveR` 34⟩≡
```
a<-rep(0,length.input)
a[text.start.index]<- -1; a[code.start.index]<-2
a<-cbind(c(text.start.index,code.start.index),
  c(rep(-1,length(text.start.index)),rep(1,length(code.start.index))))
a<-a[order(a[,1]),,drop=F]
b<-a[a[,2]!=c(-1,a[-length(a[,1]),2]),,drop=F]
a<-rep(0,length.input); a[b[,1]]<-b[,2]
a<-cumsum(a); a[code.start.index]<-0; a[empty.index]<-0
code.index<-which(a>0)
code.index<-code.index[is.na(match(code.index,use.index))]
```

35 ⟨*belege Typ-Vektor* `weaveR` 35⟩≡
```
line.typ<-rep("TEXT" ,length.input)
line.typ[empty.index]<-"EMPTY"
line.typ[text.start.index]<-"TEXT-START"
line.typ[verb.index]<-"VERBATIM"
line.typ[use.index]<-"USE"
line.typ[code.start.index]<-"HEADER"
line.typ[code.index]<-"CODE"
```

36 ⟨*erstelle Output* `weaveR` 36⟩≡
```
⟨erledige Text-Chunk-Starts weaveR 37⟩
⟨extrahiere Header-, Code- und Verwendungszeilen weaveR 38⟩
⟨schreibe Header-Zeilen weaveR 39⟩
⟨schreibe Code-Verwendungszeilen weaveR 40⟩
⟨schreibe Code-Zeilen weaveR 42⟩
⟨setze Code in Text-, Header- und Verwendungszeilen weaveR 43⟩
```

Es müssen nur die Klammeraffen entfernt werden. Einfacher ist es den entsprechenden Zeilen etwas Leeres zuzuweisen.

37 ⟨*erledige Text-Chunk-Starts* `weaveR` 37⟩≡
```
input[text.start.index]<-""
```

38 ⟨*extrahiere Header-, Code- und Verwendungszeilen* `weaveR` 38⟩≡
```
code.chunk.names<-code.start.lines<-sub(pat.chunk.header,"\\1",input[code.start.index])
use.lines<-input[use.index]
code.lines<-input[code.index]
```

39 ⟨*schreibe Header-Zeilen* `weaveR` 39⟩≡
```
no<-1:length(code.start.index)
def.ref.no<-match(gsub("\\ ","",code.start.lines), gsub("\\ ","",code.start.lines))
code.start.lines<-paste(
      # "\\rule{0mm}{0mm}\\\\\\hspace*{-3em}", "\\makebox[0mm]{",no,"}\\hspace*{3em}", #
      "\\makemarginno ", # new: margin.no by counter
      "$\\langle${\\it ",code.start.lines,"}\\ $",def.ref.no,
      "\\rangle",ifelse(no!=def.ref.no,"+",""),"\\equiv$\\newline",sep="")
input[code.start.index]<-code.start.lines
```

Über `Rchunkno` lassen sich Verweise erstellen. Hilfreich könnte dazu zwei Makros sein, um die Vorgehensweise von `\label` und `\ref` abzubilden. Dieses könnte so aussehen:

```
@
\newcommand{\chunklabel}[1]{  \newcounter{#1}\setcounter{#1}{\value{Rchunkno}} }
\newcommand{\chunkref}[1]{\arabic{#1}}

<<norm>>=
rnorm(10)
@
\chunklabel{chunkA}

Dies war der Chunk Nummer \chunkref{chunkA}.
<<*>>=
rnorm(1)

@
<<zwei>>=
2+3
@
Dies war der Chunk Nummer \chunkref{chunkB}.

Zur Erzeugung von NV-Zufallszahlen siehe: \chunkref{chunkB} und
Chunk Nummer \chunkref{chunkA} zeigt die Erstellung einer Graphik.
```

40 ⟨*schreibe Code-Verwendungszeilen* `weaveR` 40⟩≡

```
use.lines<-input[use.index]
leerzeichen.vor.use<-paste("\\verb|",sub("[^ ](.*)$","",use.lines),"|",sep="")
use.lines<-substring(use.lines,nchar(leerzeichen.vor.use)-8)
for(i in seq(use.lines)){
  uli<-use.lines[i]
  repeat{
    if(0==length(cand<-grep("<<(.*)>>",uli))) break
    uli.h<-gsub("(.*)<<(.*)>>(.*)","\\1bReAkuSeChUnK\\2bReAk\\3",uli)
    uli<-unlist(strsplit(uli.h,"bReAk"))
  }
  cand<-grep("uSeChUnK",uli); uli<-sub("uSeChUnK","",uli)
  ref.no<-match(uli[cand],code.chunk.names)
  uli[cand]<-paste("$\\langle${\\it ",uli[cand],"} ",ref.no,"$\\rangle$",sep="")
  if(length(uli)!=length(cand)){
    if(!UTF){
      uli[-cand]<-paste("\\verb\267",uli[-cand],"\267",sep="") #050612
    }else{
      uli[-cand]<-paste("\\verb\140",uli[-cand],"\140",sep="") #060516
    }
  }
  use.lines[i]<-paste(uli,collapse="")
}
input[use.index]<-paste(leerzeichen.vor.use,use.lines,"\\newline")
```

41 ⟨*old: schreibe Code-Verwendungszeilen* `weaveR` 41⟩≡

```
leerzeichen.vor.use<-paste("\\verb|",sub("<.*$","",input[use.index]),"|",sep="")
ref.no<-match(gsub("\\ ","",use.lines), gsub("\\ ","",code.chunk.names))
use.lines<- paste(leerzeichen.vor.use,"$\\langle${\\it ",
                            use.lines,"}$\\ ",ref.no,"\\rangle$\\newline")
input[use.index]<-use.lines
```

Das Zeichen `\267` rief teilweise Probleme hervor, so dass statt dessen demnächst ein anderes Verwendung finden muss. Ein Weg besteht darin, aus dem Zeichenvorrat ein ungebrauchtes Zeichen auszuwählen, dessen `catcode` zu verändern und dann dieses zu verwenden. Nachteilig ist bei diesem Zeichen, dass verschiedene Editoren dieses nicht darstellen können. Darum ist es besser ein ungewöhnliches, aber darstellbares Zeichen zu verwenden. Zum Bespiel könnte man `\343` verwenden, so dass die Zeile unten lauten würde:
`input[code.index]<-paste("\\verb\343",code.lines,"\343\\newline")`
Um ganz sicher zu gehen, dass dieses Zeichen akzeptiert wird, könnte man den `catcode` so verändern: `\catcode‘\343=12"` – also in R:
`\\\catcode‘\\\343=12"` im oberen Bereich des Dokumentes einfügen.

42 ⟨*schreibe Code-Zeilen* `weaveR` 42⟩≡
```
if(!UTF){
  input[code.index]<-paste("\\verb\267",code.lines,"\267\\newline")
}else{
  input[code.index]<-paste("\\verb\140",code.lines,"\140\\newline") #060516
}
```

43 ⟨*setze Code in Text-, Header- und Verwendungszeilen* `weaveR` 43⟩≡
```
typ<-"TEXT"
⟨setze Code in Zeilen vom Typ typ weaveR 44⟩
typ<-"HEADER"
⟨setze Code in Zeilen vom Typ typ weaveR 44⟩
typ<-"USE"
⟨setze Code in Zeilen vom Typ typ weaveR 44⟩
```

Code im Text wird auf zwei Weisen umgesetzt:
a) Zerlegung von Zeilen in Wörter. Wörter der Form `x==(1:10)+1` werden untersucht und komische Zeichen werden ersetzt. b) In Zeilen, in denen immer noch doppelte Klammern gefunden werden, werden als ganzes behandelt; dabei wird versucht von vorn beginnend zu einander passende Klammern zu finden.

44 ⟨*setze Code in Zeilen vom Typ* `typ` `weaveR` 44⟩≡
```
⟨suche in Zeilen des Typs nach Code im Text code.im.text.index weaveR 45⟩
if(0<length(code.im.text.index)){
  lines.to.check<-input[code.im.text.index]
  ⟨ersetze zusammenhängende Wortstücke weaveR 46⟩
  ⟨checke und ersetze Code im Text mit Leerzeichen weaveR 47⟩
  input[code.im.text.index]<-lines.to.check
}
```

45 ⟨*suche in Zeilen des Typs nach Code im Text* `code.im.text.index` `weaveR` 45⟩≡
```
index<-which(line.typ==typ)
code.im.text.index<-index[grep("\\[\\[(.*)\\]\\]",input[index])]
```

Die Zeilen werden mit `strsplit` aufgebrochen und die Teile mit doppelten eckigen Klammern werden behandelt. Die Behandlung erfolgt, wie mit nächsten Text-Chunk beschrieben. Anschließend wird die Zeile mit tt-gesetzten Code-Stücken wieder zusammengebaut.

46 ⟨*ersetze zusammenhängende Wortstücke* `weaveR` 46⟩≡

```
lines.to.check<-strsplit(lines.to.check," ") # Zerlegung in Worte
lines.to.check<-unlist(lapply(lines.to.check,function(x){
    ind.cand<-grep("^\\[\\[(.*)\\]\\]$",x)
    if(0<length(ind.cand)){
      cand<-gsub("^\\[\\[(.*)\\]\\]$","\\1",x[ind.cand])
      cand<-gsub("\\[\\[","DoEckOpenKl-esc",cand)
      cand<-gsub("\\]\\]","DoEckCloseKl-esc",cand)
      cand<-gsub("\\\\","\\\\char'134 ",cand)
      cand<-gsub("([#$&_%{}])","\\\\\\1",cand) #2.1.0
      cand<-gsub("\\~","\\\\char'176 ",cand)
      cand<-gsub("\\^","\\\\char'136 ",cand)
      cand<-gsub("DoSpOpenKl-esc","\\\\verb|<<|",cand) # 050612
      cand<-gsub("DoSpCloseKl-esc","\\\\verb|>>|",cand) # 050612
      x[ind.cand]<-paste("{\\tt ",cand,"}",sep="")
    }
    x<-paste(x,collapse=" ")}
)) # end of unlist(apply(..))
```

Nicht zusammenhängende Anweisungen, eingeschlossen in doppelten eckigen Klammern sind auch erlaubt. Diese werden in `lines.to.check` gesucht: `ind.cand`. Es werden die gefundenen Klammeraffen entfernt. Die verbleibenden Kandidaten werden, wie folgt, abgehandelt: Ersetzung der doppelten eckigen Klammern durch eine unwahrscheinliche Kennung: `AbCxYz` und Zerlegung der Zeilen nach diesem Muster. Der mittlere Teil wird in eine Gruppe gesetzt und Sonderzeichen werden escaped bzw. durch den Charactercode ersetzt. Dann wird die Zeile wieder zusammengebaut und das Ergebnis zugewiesen.

47 ⟨*checke und ersetze Code im Text mit Leerzeichen* `weaveR` 47⟩≡

```
ind.cand<-grep("\\[\\[(.*)\\]\\]",lines.to.check)
if(0<length(ind.cand)) {
  # zerlege Zeile in token der Form [[,  ]] und sonstige
  zsplit<-lapply(strsplit(lines.to.check[ind.cand],"\\[\\["),function(x){
     zs<-strsplit(rbind("[[",paste(x[],"\333",sep=""))[-1],"\\]\\]")
     zs<-unlist(lapply(zs,function(y){ res<-rbind("]]",y[])[-1]; res }))
     gsub("\333","",zs)
  })
  # suche von vorn beginnend zusammenpassende [[-]]-Paare
  z<-unlist(lapply(zsplit,function(x){
    repeat{
      cand.sum<-cumsum((x=="[[")-(x=="]]"))
      if(is.na(br.open<-which(cand.sum==1)[1])) break
      br.close<-which(cand.sum==0)
      if(is.na(br.close<-br.close[br.open<br.close][1])) break
      if((br.open+1)<=(br.close-1)){
        h<-x[(br.open+1):(br.close-1)]; h<-gsub("\\\\","\\\\char'134 ",h)
        h<-gsub("([#$&_%{}])","\\\\\\1",h); h<-gsub("\\~","\\\\char'176 ",h) #2.1.0
        h<-gsub(" ","\\\\ ",h) # Leerzeichen nicht vergessen! 060116
        h<-gsub("DoSpOpenKl-esc","\\\\verb|<<|",h) # 050612
        h<-gsub("DoSpCloseKl-esc","\\\\verb|>>|",h) # 050612
      x[(br.open+1):(br.close-1)]<-gsub("\\^","\\\\char'136 ",h)
      }
      x[br.open]<-"{\\tt "; x[br.close]<-"}"
      x<-c(paste(x[1:br.close],collapse=""), x[-(1:br.close)])
    }
    paste(x,collapse="")
  }))
  lines.to.check[ind.cand]<-z
}
```

48 ⟨*checke und ersetze Code im Text mit Leerzeichen* `weaveR`*, old* 48⟩≡

```
ind.cand<-grep("\\[\\[(.*)\\]\\]",lines.to.check)
if(0<length(ind.cand)) {
      extra<-lines.to.check[ind.cand]
      extra<-gsub("(.*)\\[\\[(.*)\\]\\](.*)","\\1AbCxYz\\2AbCxYz\\3",extra)
      extra<-strsplit(extra,"AbCxYz")
      extra<-unlist(lapply(extra,function(x){
        cand<-gsub("\\\\","\\\\char'134 ",x[2])
        cand<-gsub("([#$&_%{}])","\\\\\\1",cand)
        cand<-gsub("\\~","\\\\char'176 ",cand)
        x[2]<-gsub("\\^","\\\\char'136 ",cand)
        x<-paste(x[1],"{\\tt ",x[2],"}",if(!is.na(x[3]))x[3],collapse="")}))
      lines.to.check[ind.cand]<-extra
}
```

Ein Test von `weaveR`.

49 ⟨*teste Funktion* `weaveR` 49⟩≡

⟨*definiere Funktion* `weaveR` (never defined)⟩

```
weaveR("out.rev"); system("cat q|latex out.tex")
```

3.2 Help-Page

50 ⟨*define-weaveR-help* 50⟩≡

```
\name{weaveR}
\alias{weaveR}
\title{ function to weave a file }
\description{
  \code{weaveR} reads a file that is written according to
  the rules of the \code{noweb} system and performs a simple kind
  of weaving. As a result a LaTeX file is generated.
}
\usage{
weaveR(in.file,out.file)
}
%- maybe also 'usage' for other objects documented here.
\arguments{
  \item{in.file}{ name of input file }
  \item{out.file}{ name of output file; if missing the extension of the
  input file is turned to \code{.tex} }
}
\details{
  General remarks: A \code{noweb} file consists of a mixture of text
  and code chunks. An \code{@} character (in column 1 of a line)
  indicates the beginning of a text chunk. \code{<<name of code chunk>>=}
  (starting at column 1 of a line) is a header line of a code chunk with
  a name defined by the text between \code{<<} and \code{>>=}.
  A code chunk is finished by the beginning of hte next text chunk.
  Within the code chunk you are allowed to use other code chunks by referencing
  them by name ( for example by: \code{<<name of code chunk>>} ).
  In this way you can separate a big job in smaller ones.

  Technical remarks:
  To format small pieces of code in text chunks you have to put them in
  \code{[[...]]}-brackets: \code{text text [[code]] text text}.
  One occurence of such a code in a text line is assumed to work always.
  If an error emerges caused by formatting code in a text chunk
  simplify the line by splitting it.
  Sometimes you want to use
  \code{[[}- or even \code{<<}-characters in your text. Then it
  may be necessary to escape them by an \code{@}-sign and
  you have to type in: \code{@<<}, \code{@[[} and so on.

  \code{weaveR} expands the input by adding some latex macros
  to typeset code by a typewriter font.
  Furthermore chunk numbers are appended to code chunk headers.
  The number of the last code chunk is stored in LaTeX-counter \code{Rchunkno}.
  After defining
  \code{\newcommand\{\chunklabel\}[1]\{\newcounter\{#1\}\setcounter\{#1\}\{\value\{Rchun
  and \code{\newcommand\{\chunkref\}[1]\{\arabic\{#1\} \} } you can label a code chunk
  by \code{\chunklabel\{xyzname\} } and reference it by \code{\chunkref\{xyzname\} }.
}
\value{
  a latex file is generated
}
\references{ \url{http://www.eecs.harvard.edu/~nr/noweb/intro.html} }
\author{Hans Peter Wolf}
\seealso{  \code{\link{tangleR}} }
\examples{
\dontrun{
## This example cannot be run by examples() but should be work in an interactive R sessi
  weaveR("testfile.rev","testfile.tex")
  weaveR("testfile.rev")
}
```

```
## The function is currently defined as
weaveR<-function(in.file,out.file){
  # german documentation of the code:
  # look for file webR.pdf, P. Wolf 050204
  ...
}
}
\keyword{file}
\keyword{documentation}
\keyword{programming}
```

4 TEIL III — WEAVEtoHTML

4.1 `weaveRhtml` — eine WEAVE-Funktion zur Erzeugung einfacher `html`-Pendants

Aufbauend auf der `weaveR`-Funktion wird in diesem Teil eine einfache Funktionen zur Erzeugung einfacher `html`-Dateien beschrieben. Die Nebenbedingungen der Realisation entsprechen denen von `weaveR`. Auch die grobe Struktur und besonders der Anfang der Lösung wurde im wesentlichen kopiert. Die Funktion besitzt folgenden Aufbau:

51 ⟨*define-weaveRhtml* 51⟩≡

```
weaveRhtml<-function(in.file,out.file){
  # german documentation of the code:
  # look for file webR.pdf, P. Wolf 060920
  ⟨initialisiere weaveRhtml 52⟩
  ⟨lese Datei ein weaveRhtml 53⟩
  ⟨entferne Kommentarzeichen weaveRhtml 55⟩
  ⟨substituiere mit @ versehene Zeichengruppen weaveRhtml 54⟩
  ⟨stelle Typ der Zeilen fest weaveRhtml 66⟩
  ⟨erstelle Output weaveRhtml 72⟩
  ⟨ersetze Umlaute weaveRhtml 56⟩
  ⟨korrigiere ursprünglich mit @ versehene Zeichengruppen weaveRhtml 57⟩
  ⟨formatiere Überschriften weaveRhtml 59⟩
  ⟨definiere einfachen head weaveRhtml 61⟩
  ⟨setze Schriften um weaveRhtml 64⟩
  ⟨entferne unbrauchbare Makros weaveRhtml 62⟩
  ⟨schreibe Ergebnis in Datei weaveRhtml 65⟩
  "ok"
}
```

Zunächst fixieren wir die Suchmuster für wichtige Dinge. Außerdem stellen wir fest, ob R auf UTF-8-Basis arbeitet.

52 ⟨*initialisiere* `weaveRhtml` 52⟩≡

```
verbose<-FALSE
pat.use.chunk<-paste("<","<(.*)>",">",sep="")
pat.chunk.header<-paste("^<","<(.*)>",">=",sep="")
pat.verbatim.begin<-"\\\\begin\\{verbatim\\}"
pat.verbatim.end<-"\\\\end\\{verbatim\\}"
pat.leerzeile<-"^(\\ )*$"
lcctype<-grep("LC_CTYPE",strsplit(Sys.getlocale(),";")[[1]],value=T)
UTF<-(1==length(grep("UTF",lcctype)))
UTF<- UTF | nchar(deparse("\xc3")) > 3
if(verbose)  {if(UTF) cat("character set: UTF\n") else cat("character set: ascii\n")}
```

Die zu bearbeitende Datei wird zeilenweise auf die Variable `input` eingelesen.

53 ⟨*lese Datei ein* `weaveRhtml` 53⟩≡
```
if(!file.exists(in.file)) in.file<-paste(in.file,"rev",sep=".")
if(!file.exists(in.file)){
  cat(paste("ERROR:",in.file,"not found!!??\n"))
  return("Error in weaveRhtml: file not found")
}
input<-readLines(in.file)
input<-gsub("\t","      ",input)

length.input<-length(input)
```

54 ⟨*substituiere mit @ versehene Zeichengruppen* `weaveRhtml` 54⟩≡
```
input<-gsub("@>>","DoSpCloseKl-ESC",gsub("@<<","DoSpOpenKl-ESC",input))
input<-gsub("@\\]\\]","DoEckCloseKl-ESC",gsub("@\\[\\[","DoEckOpenKl-ESC",input))
```

55 ⟨*entferne Kommentarzeichen* `weaveRhtml` 55⟩≡
```
h<-grep("^[ ]*%",input)
if(0<length(h)) input<-input[-h]
```

Umlaute sind ein Dauerbrenner. Hinweis: im richtigen Code steht unten übrigens: *äöüÄÖÜ* sowie in der ersten Zeile ein *ß*.

56 ⟨*ersetze Umlaute* `weaveRhtml` 56⟩≡
```
if(!UTF){
  # im Tcl/Tk-Textfenster eingegeben -> iso-8859-1 (man iso-8859-1 / Latin1 / unicode
  input<-gsub("\283","",input)
  input<-chartr("\244\266\274\204\226\234\237","\344\366\374\304\326\334\337",input)
  # Latin1 -> TeX-Umlaute
  input<-gsub("\337","&szlig;",input) # SZ
  input<-gsub("(\344|\366|\374|\304|\326|\334)","&\\1uml;",input)
  input<-chartr("\344\366\374\304\326\334","aouAOU",input)
}else{
  input<-gsub("\283\237","&szlig;",input)
  input<-gsub("(\283\244|\283\266|\283\274|\283\204|\283\226|\283\234)",
                            "&\\1uml;",input)
  input<-chartr("\283\244\283\266\283\274\283\204\283\226\283\234",
                              "aouAOU", input)
}
if(verbose) cat("german Umlaute replaced\n")
```

Vor dem Wegschreiben müssen die besonderen Zeichengruppen zurückübersetzt werden.

57 ⟨*korrigiere ursprünglich mit @ versehene Zeichengruppen* `weaveRhtml` 57⟩≡
```
#input<-gsub("DoSpCloseKl-esc",">>",gsub("DoSpOpenKl-esc","<<",input))
input<-gsub("DoSpCloseKl-ESC","&gt;&gt;",gsub("DoSpOpenKl-ESC","&lt;&lt;",input))
input<-gsub("DoEckCloseKl-ESC","]]",gsub("DoEckOpenKl-ESC","[[",input))
```

Die Funktion `get.argument` holt die Argumente aller Vorkommnisse eines LaTeX-Kommandos, dieses wird verwendet für Graphik-Einträge mittels `includegraphics`. `get.head.argument` ermittelt für den Dokumentenkopf wichtige Elemente, dieses wird zur Ermittlung von Autor, Titel und Datum verwendet. `transform.command` ersetzt im Text `txt` LaTeX-Kommandos mit einem Argument, zur Zeit nicht benutzt. `transform.command.line` transformiert LaTeX-Kommandos mit einem Argument, die in einer Zeile zu finden sind, dieses wird gebraucht für kurzzeitige Schriftenwechsel. `transform.structure.command`

58 ⟨*initialisiere* `weaveRhtml` 52⟩+≡

```
get.argument<-function(command,txt,default="",kla="{",kle="}",dist=TRUE){
## print("get.argument")
  command<-paste("\\\\",command,sep="")
  if(0==length(grep(command,txt))) return(default)
  txt<-unlist(strsplit(paste(txt,collapse="\n"),command))[-1]
  arg<-lapply(txt,function(x){
       n<-nchar(x); if(n<3) return(x)
       x<-substring(x,1:n,1:n)
       h<-which(x==kla)[1]; if(is.na(h)) h<-1
       if(dist)x<-x[h:length(x)]
       k<-which(cumsum((x==kla)-(x==kle))==0)[1]
       paste(x[2:(k-1)],collapse="")
  })
  arg
}
get.head.argument<-function(command,txt,default="",kla="{",kle="}",dist=TRUE){
## print("get.head.argument")
  command<-paste("\\\\",command,sep="")
  txt<-unlist(strsplit(paste(txt,collapse="\n"),command))[-1]
  arg<-lapply(txt,function(x){
       n<-nchar(x); x<-substring(x,1:n,1:n)
       if(dist)x<-x[which(x==kla)[1]:length(x)]
       k<-which(cumsum((x==kla)-(x==kle))==0)[1]
       paste(x[2:(k-1)],collapse="")
  })
  unlist(arg)
}
transform.command<-function(command,txt,atag="<i>",etag="</i>",
                      kla="{",kle="}"){
## print("transform.command")
  command<-paste("\\\\",command,sep="")
  ##  if(0==length(grep(command,txt))){print("hallo"); return(txt)}
  txt<-unlist(strsplit(paste(txt,collapse="\n"),command))
  tx<-unlist(lapply(txt[-1],function(x){
       n<-nchar(x); if(n<4) return(x)
       x<-substring(x,1:n,1:n)
       an<-which(x==kla)[1]
       en<-which(cumsum((x==kla)-(x==kle))==0)[1]
      if(!is.na(an))
       paste(atag,paste(x[(an+1):(en-1)],collapse=""),etag,
            paste(x[-(1:en)],collapse="")) else x
  }))
  unlist(strsplit(c(txt[1],tx),"\n"))
}
transform.command.line<-function(command,txt,atag="<i>",etag="</i>",
                      kla="{",kle="}"){
  command<-paste("\\\\",command,sep="")
  if(0==length(ind<-grep(command,txt))){return(txt)}
  txt.lines<-txt[ind]
  txt.lines<-strsplit(txt.lines,command)
  txt.lines<-lapply(txt.lines,function(xxx){
    for(i in 2:length(xxx)){
```

```
      m<-nchar(xxx[i])
      if(is.na(m)) break
      x.ch<-substring(xxx[i],1:m,1:m); x.info<-rep(0,m)
      x.info<-cumsum((x.ch=="{") - (x.ch=="}"))
      h<-which(x.info==0)[1]
      if(!is.na(h)) {x.ch[1]<-atag; x.ch[h]<- etag }
      xxx[i]<-paste(x.ch,collapse="")
    }
    paste(xxx,collapse="")
  })
  txt[ind]<-unlist(txt.lines)
  txt
}
transform.structure.command<-function(command,txt,atag="<i>",etag="</i>",
                       kla="{",kle="}"){
## print("transform.structure.command")
  command<-paste("\\\\",command,sep="")
  ##  if(0==length(grep(command,txt))){print("hallo"); return(txt)}
  txt<-unlist(strsplit(paste(txt,collapse="\n"),command))
  tx<-unlist(lapply(txt[-1],function(x){
       n<-nchar(x); if(n<4) return(x)
       x<-substring(x,1:n,1:n)
       an<-which(x==kla)[1]
       en<-which(cumsum((x==kla)-(x==kle))==0)[1]
      if(!is.na(an))
       paste(atag,paste(x[(an+1):(en-1)],collapse=""),etag,
            paste(x[-(1:en)],collapse=""))
              else x
  }))
  unlist(strsplit(c(txt[1],tx),"\n"))
}
```

59 ⟨*formatiere Überschriften* `weaveRhtml` 59⟩≡

```
atag<-"<h2>"; etag<-"</h2>"; command<-"section"
⟨formatiere Strukturkommandos weaveRhtml 60⟩
sec.links<-command.links
sec.no<-com.lines
atag<-"<h3>"; etag<-"</h3>"; command<-"subsection"
⟨formatiere Strukturkommandos weaveRhtml 60⟩
atag<-"<h4>"; etag<-"</h4>"; command<-"subsubsection"
⟨formatiere Strukturkommandos weaveRhtml 60⟩
atag<-"<br><b>"; etag<-"</b>"; command<-"paragraph"
⟨formatiere Strukturkommandos weaveRhtml 60⟩
subsec.links<-command.links
subsec.no<-com.lines
contents<-c(paste(seq(sec.links),sec.links),paste("  ",subsec.links))[order(c(
## print(contents)
```

60 ⟨*formatiere Strukturkommandos* `weaveRhtml` 60⟩≡

```
command.n<-nchar(command)+2; command.links<-NULL
kla<-"{"; kle<-"}"
## print("STRUKTUR")
if(0<length(com.lines<-grep(paste("^\\\\",command,sep=""),input))){
  sec<-NULL
  for(i in seq(com.lines)){
    txt<-input[com.lines[i]+0:2]
    txt<-paste(txt,collapse="\n"); n<-nchar(txt)
    x<-substring(txt,command.n:n,command.n:n)
    en<-which(cumsum((x==kla)-(x==kle))==0)[1]
    x[1]<-paste("<a name=\"",command,i,"\">",atag,sep="")
    x[en]<-etag;  txt<-paste(x,collapse="")
    sec<-c(sec,paste(x[2:(en-1)],collapse=""))
    input[com.lines[i]+0:2]<-unlist(strsplit(txt,"\n"))
  }
  command.links<-paste("<a href=\"#",command,seq(com.lines),"\">",sec,"</a>",sep="")
}
```

61 ⟨*definiere einfachen head* `weaveRhtml` 61⟩≡

```
## if(verbose) print("head")
head<-grep("^\\\\title|^\\\\author|^\\\\date",input)
if(0<length(head)){
  h<-min(max(head)+5,length(input))
  head<-input[1:h]
  titel<-get.head.argument("title",head)[1]
  titel<-sub("Report: \\\\rule\\{(.*)\\}","Report: ..............",titel)
  autor<-get.head.argument("author",head)[1]
  autor<-sub("File: \\\\jobname.rev",paste("File:",sub(".*/","",in.file)),autor)
  datum<-get.head.argument("date",head)[1]
  if(is.null(datum)) datum<-date()
  ## print(datum)
} else {
  head<-""; titel<-paste("File:",in.file); autor<-"processed by weaveRhtml"; datum<-date
}
if(0<length(h<-grep("\\\\begin\\{document\\}",input)))
  input<-input[-(1:h[1])]
input[1]<-paste(collapse="\n",
  "<!--  generated by weaveRhtml --><html><head>",
  "<meta content=\"text/html; charset=ISO-8859-1\">",
  "<title>",titel,"</title></head>",
  "<body bgcolor=\"#FFFFFF\">",
  "<h1>",if(!is.null(titel))titel,"</h1>",
  "<h2>",if(!is.null(autor))autor,"</h2>",
  "<h3>",if(!is.null(datum))datum,"</h3>",
  "<h4>",paste(contents,collapse="<br>"),"</h4>"
)
```

62 ⟨*entferne unbrauchbare Makros* `weaveRhtml` 62⟩≡

```
input<-gsub("\\\\newpage","",input)
input<-gsub("\\\\tableofcontents","",input)
input<-gsub("\\\\raggedright","",input)
input<-gsub("\\\\\\\\","<br>",input)
h<-grep("\\\\maketitle|\\\\author|\\\\date|\\\\title|\\\\end\\{document\\}",input)
if(0<length(h)) input<-input[-h]
```

63 ⟨*zentriere und quote* `weaveRhtml` 63⟩≡
```
input<-sub("\\\\begin\\{center}","<center>",input)
input<-sub("\\\\end\\{center}","</center>",input)
input<-sub("\\\\begin\\{quote}","<ul>",input)
input<-sub("\\\\end\\{quote}","</ul>",input)
input<-sub("\\\\begin\\{itemize}","<ul>",input)
input<-sub("\\\\end\\{itemize}","</ul>",input)
input<-sub("\\\\item","</li><li>",input)
```

64 ⟨*setze Schriften um* `weaveRhtml` 64⟩≡
```
if(0<length(h<-grep("\\\\myemph",input))){
  input<-transform.command.line("myemph",input,"<i>","</i>")
}
if(0<length(h<-grep("\\\\texttt",input))){
  input<-transform.command.line("texttt",input,"<code>","</code>")
}
```

Zum Schluss müssen wir die modifizierte Variable `input` wegschreiben.

65 ⟨*schreibe Ergebnis in Datei* `weaveRhtml` 65⟩≡
```
if(missing(out.file)||in.file==out.file){
  out.file<-sub("\\.([A-Za-z])*$","",in.file)
}
if(0==length(grep("\\.html$",out.file)))
  out.file<-paste(out.file,".html",sep="")
## out.file<-"/home/wiwi/pwolf/tmp/out.html"
get("cat","package:base")(input,sep="\n",file=out.file)
cat("weaveRhtml process finished\n")
```

Zu jeder Zeile wird ihr Typ festgestellt und auf dem Vektor `line.typ` eine Kennung vermerkt. Außerdem merken wir zu jedem Typ auf einer Variablen alle Zeilennummern des Typs. Wir unterscheiden:

Typ	Kennung	Indexvariable
Leerzeile	`EMPTY`	`empty.index`
Text-Chunk-Start	`TEXT-START`	`text.start.index`
Code-Chunk-Start	`HEADER`	`code.start.index`
Code-Chunk-Verwendungen	`USE`	`use.index`
normale Code-Zeilen	`CODE`	`code.index`
normale Textzeilen	`TEXT`	
Verbatim-Zeilen	`VERBATIM`	`verb.index`

Leerzeilen, Text- und Code-Chunk-Anfänge sind leicht zu finden.
Code-Verwendungen sind alle diejenigen Zeilen, die << und >> enthalten, jedoch keine Headerzeilen sind. Am schwierigsten sind normale Code-Zeilen zu identifizieren. Sie werden aus den Code-Chunk-Anfängen und den Text-Chunkanfängen ermittelt, wobei die `USE`-Zeilen wieder ausgeschlossen werden. Alle übrigen Zeilen werden als Textzeilen eingestuft.

66 ⟨*stelle Typ der Zeilen fest* `weaveRhtml` 66⟩≡
```
⟨checke Leer-, Textzeilen weaveRhtml 67⟩
⟨behandle verbatim-Zeilen weaveRhtml 68⟩
⟨checke Header- und Use-Zeilen weaveRhtml 69⟩
⟨checke normale Code-Zeilen weaveRhtml 70⟩
⟨belege Typ-Vektor weaveRhtml 71⟩
```

67 ⟨*checke Leer-, Textzeilen* `weaveRhtml` 67⟩≡
```
empty.index<-grep(pat.leerzeile,input)
text.start.index<-which("@"==substring(input,1,1))
```

68 ⟨*behandle verbatim-Zeilen* `weaveRhtml` 68⟩≡
```
a<-rep(0,length(input))
an<-grep(pat.verbatim.begin,input)
if(0<length(an)) {
  a[an]<- 1
  en<-grep(pat.verbatim.end,input); a[en]<- -1
  input[a==1]<-"<code><FONT COLOR=\"#0000FF\">"
  input[a==-1]<-"</font></code><br>"
  a<-cumsum(a)
}
verb.index<-which(a>0)
input[verb.index]<-paste(input[verb.index],"<br>")
```

69 ⟨*checke Header- und Use-Zeilen* `weaveRhtml` 69⟩≡
```
code.start.index<-grep(pat.chunk.header,input)
use.index<-grep(pat.use.chunk,input)
use.index<-use.index[is.na(match(use.index,code.start.index))]
```

70 ⟨*checke normale Code-Zeilen* `weaveRhtml` 70⟩≡
```
a<-rep(0,length.input)
a[text.start.index]<- -1; a[code.start.index]<-2
a<-cbind(c(text.start.index,code.start.index),
  c(rep(-1,length(text.start.index)),rep(1,length(code.start.index))))
a<-a[order(a[,1]),,drop=F]
b<-a[a[,2]!=c(-1,a[-length(a[,1]),2]),,drop=F]
a<-rep(0,length.input); a[b[,1]]<-b[,2]
a<-cumsum(a); a[code.start.index]<-0; a[empty.index]<-0
code.index<-which(a>0)
code.index<-code.index[is.na(match(code.index,use.index))]
```

71 ⟨*belege Typ-Vektor* `weaveRhtml` 71⟩≡
```
line.typ<-rep("TEXT" ,length.input)
line.typ[empty.index]<-"EMPTY"
line.typ[text.start.index]<-"TEXT-START"
line.typ[verb.index]<-"VERBATIM"
line.typ[use.index]<-"USE"
line.typ[code.start.index]<-"HEADER"
line.typ[code.index]<-"CODE"
```

72 ⟨*erstelle Output* `weaveRhtml` 72⟩≡
⟨*zentriere und quote* `weaveRhtml` 63⟩
⟨*erledige Text-Chunk-Starts* `weaveRhtml` 73⟩
⟨*ersetze Befehl zur Bildeinbindung* 74⟩
⟨*extrahiere Header-, Code- und Verwendungszeilen* `weaveRhtml` 75⟩
⟨*schreibe Header-Zeilen* `weaveRhtml` 76⟩
⟨*schreibe Code-Verwendungszeilen* `weaveRhtml` 77⟩
⟨*schreibe Code-Zeilen* `weaveRhtml` 79⟩
⟨*setze Code in Text-, Header- und Verwendungszeilen* `weaveRhtml` 80⟩

Es müssen nur die Klammeraffen entfernt werden. Zur Kennzeichnung der Absätze erzeugen wir einen neuen Paragraphen durch `<p>`.

73 ⟨*erledige Text-Chunk-Starts* `weaveRhtml` 73⟩≡
```
input[text.start.index]<-"<p>"    # vorher: @
lz<-grep("^[ ]*$",input)
if(0<length(lz)) input[lz]<-"<br>"
```

74 ⟨*ersetze Befehl zur Bildeinbindung* 74⟩≡
```
plz.ind<-grep("\\\\includegraphics",input)
if(0<length(plz.ind)){
  plz<-input[plz.ind]
  h<-unlist(get.argument("includegraphics",plz))
  h<-paste("<img SRC=\"",sub(".ps$",".jpg",h),"\">",sep="")
  input[plz.ind]<-h
}
```

75 ⟨*extrahiere Header-, Code- und Verwendungszeilen* `weaveRhtml` 75⟩≡
```
code.chunk.names<-code.start.lines<-sub(pat.chunk.header,"\\1",input[code.start.index])
use.lines<-input[use.index]
code.lines<-input[code.index]
## print(input[code.start.index])
```

76 ⟨*schreibe Header-Zeilen* `weaveRhtml` 76⟩≡
```
no<-1:length(code.start.index)
def.ref.no<-match(gsub("\\ ","",code.start.lines), gsub("\\ ","",code.start.lines))
code.start.lines<-paste(
      "<a name=\"codechunk",no,"\"></a>",
      "<a href=\"\#codechunk",1+(no%%max(no)),"\">",
      "<br>Chunk:",no," <i>&lt;",code.start.lines,def.ref.no,
      "&gt;",ifelse(no!=def.ref.no,"+",""),"=</i></a><br>",sep="")
input[code.start.index]<-code.start.lines
```

77 ⟨*schreibe Code-Verwendungszeilen* `weaveRhtml` 77⟩≡
```
use.lines<-input[use.index]
leerzeichen.vor.use<-sub("[^ ](.*)$","",use.lines)
use.lines<-substring(use.lines,nchar(leerzeichen.vor.use))
leerzeichen.vor.use<-gsub("\\ "," ",leerzeichen.vor.use)
for(i in seq(use.lines)){
  uli<-use.lines[i]
  such<-paste("(.*)<","<(.*)>",">(.*)",sep="")
  repeat{
    if(0==length(cand<-grep("<<(.*)>>",uli))) break
    uli.h<-gsub(such,"\\1BrEaKuSeCHUNK\\2BrEaK\\3",uli)
    uli<-unlist(strsplit(uli.h,"BrEaK"))
  }
  cand<-grep("uSeCHUNK",uli); uli<-sub("uSeCHUNK","",uli)
  ref.no<-match(uli[cand],code.chunk.names)
  uli[cand]<-paste("<code>&lt;",uli[cand]," ",ref.no,"&gt;</code>",sep="")
  if(length(uli)!=length(cand)){
    if(!UTF){
      uli[-cand]<-paste("",uli[-cand],"",sep="") #050612
    }else{
      uli[-cand]<-paste("",uli[-cand],"",sep="") #060516
    }
  }
  use.lines[i]<-paste(uli,collapse="")
}
input[use.index]<-paste(leerzeichen.vor.use,use.lines,"<br>")
```

78 ⟨*ddd* 78⟩≡
```
uli<-paste("xxxt<","<hallo>",">asdf",sep="")
such<-paste("(.*)<","<(.*)>",">(.*)",sep="")
uli.h<-gsub(such,"\\1bReAkuSeChUnK\\2bReAk\\3",uli)
rm(uli,uli.h)
```

Das Zeichen `\267` rief teilweise Probleme hervor, so dass statt dessen demnächst ein anderes Verwendung finden muss. Ein Weg besteht darin, aus dem Zeichenvorrat ein ungebrauchtes Zeichen auszuwählen, dessen `catcode` zu verändern und dann dieses zu verwenden. Nachteilig ist bei diesem Zeichen, dass verschiedene Editoren dieses nicht darstellen können. Darum ist es besser ein ungewöhnliches, aber darstellbares Zeichen zu verwenden. Zum Bespiel könnte man `\343` verwenden, so dass die Zeile unten lauten würde:
`input[code.index]<-paste("\\verb\343",code.lines,"\343\\newline")`
Um ganz sicher zu gehen, dass dieses Zeichen akzeptiert wird, könnte man den `catcode` so verändern: `\catcode‘\343=12"` – also in R:
`\\\catcode‘\\\343=12"` im oberen Bereich des Dokumentes einfügen.

79 ⟨*schreibe Code-Zeilen* `weaveRhtml` 79⟩≡
```
leerzeichen.vor.c<-gsub("\t","      ",code.lines)
leerzeichen.vor.c<-sub("[^ ](.*)$","",leerzeichen.vor.c)
leerzeichen.vor.c<-gsub("\\ "," ",leerzeichen.vor.c)
if(!UTF){
  input[code.index]<-paste(leerzeichen.vor.c,"<code>",code.lines,"</code><br>")
}else{
  input[code.index]<-paste(leerzeichen.vor.c,"<code>",code.lines,"</code><br>")
}
```

80 ⟨*setze Code in Text-, Header- und Verwendungszeilen* `weaveRhtml` 80⟩≡
```
typ<-"TEXT"
⟨setze Code in Zeilen vom Typ typ weaveRhtml 81⟩
typ<-"HEADER"
⟨setze Code in Zeilen vom Typ typ weaveRhtml 81⟩
typ<-"USE"
⟨setze Code in Zeilen vom Typ typ weaveRhtml 81⟩
```

Code im Text wird auf zwei Weisen umgesetzt:
a) Zerlegung von Zeilen in Wörter. Wörter der Form `x==(1:10)+1` werden untersucht und komische Zeichen werden ersetzt. b) In Zeilen, in denen immer noch doppelte Klammern gefunden werden, werden als ganzes behandelt; dabei wird versucht von vorn beginnend zu einander passende Klammern zu finden.

81 ⟨*setze Code in Zeilen vom Typ* `typ` `weaveRhtml` 81⟩≡
```
⟨suche in Zeilen des Typs nach Code im Text code.im.text.index weaveRhtml 82⟩
if(0<length(code.im.text.index)){
  lines.to.check<-input[code.im.text.index]
  ⟨ersetze zusammenhängende Wortstücke weaveRhtml 83⟩
  ⟨checke und ersetze Code im Text mit Leerzeichen weaveRhtml 85⟩
  input[code.im.text.index]<-lines.to.check
}
```

82 ⟨*suche in Zeilen des Typs nach Code im Text* `code.im.text.index` `weaveRhtml` 82⟩≡
```
index<-which(line.typ==typ)
code.im.text.index<-index[grep("\\[\\[(.*)\\]\\]",input[index])]
```

Die Zeilen werden mit `strsplit` aufgebrochen und die Teile mit doppelten eckigen Klammern werden behandelt. Die Behandlung erfolgt, wie mit nächsten Text-Chunk beschrieben. Anschließend wird die Zeile mit tt-gesetzten Code-Stücken wieder zusammengebaut.

83 ⟨*ersetze zusammenhängende Wortstücke* `weaveRhtml` 83⟩≡

```
lines.to.check<-strsplit(lines.to.check," ") # Zerlegung in Worte
lines.to.check<-unlist(lapply(lines.to.check,function(x){
    ind.cand<-grep("^\\[\\[(.*)\\]\\]$",x)
    if(0<length(ind.cand)){
      cand<-gsub("^\\[\\[(.*)\\]\\]$","\\1",x[ind.cand])
      cand<-gsub("\\[\\[","DoEckOpenKl-ESC",cand)
      cand<-gsub("\\]\\]","DoEckCloseKl-ESC",cand)
      cand<-gsub("DoSpOpenKl-ESC","<<",cand) # 050612
      cand<-gsub("DoSpCloseKl-ESC",">>",cand) # 050612
      x[ind.cand]<-paste("<code>",cand,"</code>",sep="")
    }
    x<-paste(x,collapse=" ")}
)) # end of unlist(apply(..))
```

84 ⟨*old* 84⟩≡

```
lines.to.check<-strsplit(lines.to.check," ") # Zerlegung in Worte
lines.to.check<-unlist(lapply(lines.to.check,function(x){
    ind.cand<-grep("^\\[\\[(.*)\\]\\]$",x)
    if(0<length(ind.cand)){
      cand<-gsub("^\\[\\[(.*)\\]\\]$","\\1",x[ind.cand])
      cand<-gsub("\\[\\[","DoEckOpenKl-ESC",cand)
      cand<-gsub("\\]\\]","DoEckCloseKl-ESC",cand)
      cand<-gsub("\\\\","\\\\char'134 ",cand)
      cand<-gsub("([#$&_%{}])","\\\\\\1",cand) #2.1.0
      cand<-gsub("\\~","\\\\char'176 ",cand)
      cand<-gsub("\\^","\\\\char'136 ",cand)
      cand<-gsub("DoSpOpenKl-ESC","\\\\verb|<<|",cand) # 050612
      cand<-gsub("DoSpCloseKl-ESC","\\\\verb|>>|",cand) # 050612
      x[ind.cand]<-paste("{\\tt ",cand,"}",sep="")
    }
    x<-paste(x,collapse=" ")}
)) # end of unlist(apply(..))
```

Nicht zusammenhängende Anweisungen, eingeschlossen in doppelten eckigen Klammern sind auch erlaubt. Diese werden in `lines.to.check` gesucht: `ind.cand`. Es werden die gefundenen Klammeraffen entfernt. Die verbleibenden Kandidaten werden, wie folgt, abgehandelt: Ersetzung der doppelten eckigen Klammern durch eine unwahrscheinliche Kennung: `AbCxYz` und Zerlegung der Zeilen nach diesem Muster. Der mittlere Teil wird in eine Gruppe gesetzt und Sonderzeichen werden escaped bzw. durch den Charactercode ersetzt. Dann wird die Zeile wieder zusammengebaut und das Ergebnis zugewiesen.

85 ⟨*checke und ersetze Code im Text mit Leerzeichen* `weaveRhtml` 85⟩≡

```
ind.cand<-grep("\\[\\[(.*)\\]\\]",lines.to.check)
if(0<length(ind.cand)) {
  # zerlege Zeile in token der Form [[,  ]] und sonstige
  zsplit<-lapply(strsplit(lines.to.check[ind.cand],"\\[\\["),function(x){
     zs<-strsplit(rbind("[[",paste(x[],"\333",sep=""))[-1],"\\]\\]")
     zs<-unlist(lapply(zs,function(y){ res<-rbind("]]",y[])[-1]; res }))
     gsub("\333","",zs)
  })
  # suche von vorn beginnend zusammenpassende [[-]]-Paare
  z<-unlist(lapply(zsplit,function(x){
    repeat{
      cand.sum<-cumsum((x=="[[")-(x=="]]"))
      if(is.na(br.open<-which(cand.sum==1)[1])) break
      br.close<-which(cand.sum==0)
      if(is.na(br.close<-br.close[br.open<br.close][1])) break
      if((br.open+1)<=(br.close-1)){
        h<-x[(br.open+1):(br.close-1)]
        h<-gsub(" "," ",h) # Leerzeichen nicht vergessen! 060116
        h<-gsub("DoSpOpenKl-ESC","<<",h)
        h<-gsub("DoSpCloseKl-ESC",">>",h)
        x[(br.open+1):(br.close-1)]<-h
      }
      x[br.open]<-"<code> "; x[br.close]<-"</code>"
      x<-c(paste(x[1:br.close],collapse=""), x[-(1:br.close)])
    }
    paste(x,collapse="")
  }))
  lines.to.check[ind.cand]<-z
}
```

Konstruktion eines geeigneten Shellscript.

86 ⟨*lege bin-Datei* `weaveRhtml` *an* 86⟩≡

```
tangleR("weaveRhtml",expand.roots="")
file.copy("weaveRhtml.R","/home/wiwi/pwolf/bin/revweaveRhtml.R",TRUE)
h<-'echo "source(\\"/home/wiwi/pwolf/bin/revweaveRhtml.R\\"); weaveRhtml(\'"$1"\')" | R
cat(h,"\n",file="/home/wiwi/pwolf/bin/revweaveRhtml")
system("chmod +x /home/wiwi/pwolf/bin/revweaveRhtml")
```

Ein Test von `weaveRhtml`.

87 ⟨*teste Funktion* `weaveRhtml` 87⟩≡

```
⟨definiere-weaveRhtml (never defined)⟩
#weaveRhtml("/home/wiwi/pwolf/tmp/vskm16.rev")
weaveRhtml("/home/wiwi/pwolf/tmp/doof")
#weaveRhtml("/home/wiwi/pwolf/tmp/aufgabenblatt2.rev")
```

88 ⟨*t* 88⟩≡

```
⟨teste Funktion weaveRhtml 87⟩
```

4.2 Help-Page

89 ⟨*define-weaveRhtml-help* 89⟩≡

```
\name{weaveRhtml}
\alias{weaveRhtml}
\title{ function to weave a rev-file to a html-file}
\description{
  \code{weaveRhtml} reads a file that is written according to
  the rules of the \code{noweb} system and performs a simple kind
  of weaving. As a result a html-file is generated.
}
\usage{
weaveRhtml(in.file,out.file)
}
%- maybe also 'usage' for other objects documented here.
\arguments{
  \item{in.file}{ name of input file }
  \item{out.file}{ name of output file; if this argument is missing the extension of the
  input file is turned to \code{.html} }
}
\details{
  General remarks: A \code{noweb} file consists of a mixture of text
  and code chunks. An \code{@} character (in column 1 of a line)
  indicates the beginning of a text chunk. \code{<<name of code chunk>>=}
  (starting at column 1 of a line) is a header line of a code chunk with
  a name defined by the text between \code{<<} and \code{>>=}.
  A code chunk is finished by the beginning of hte next text chunk.
  Within the code chunk you are allowed to use other code chunks by referencing
  them by name ( for example by: \code{<<name of code chunk>>} ).
  In this way you can separate a big job in smaller ones.

  Technical remarks:
  To format small pieces of code in text chunks you have to put them in
  \code{[[...]]}-brackets: \code{text text [[code]] text text}.
  One occurence of such a code in a text line is assumed to work always.
  If an error emerges caused by formatting code in a text chunk
  simplify the line by splitting it.
  Sometimes you want to use
  \code{[[}- or even \code{<<}-characters in your text. Then it
  may be necessary to escape them by an \code{@}-sign and
  you have to type in: \code{@<<}, \code{@[[} and so on.

  \code{weaveRhtml} expands the input by adding a simple html-header
  as well as some links for navigation.
  Chunk numbers are written in front of the code chunk headers.

  Further details:
  Some LaTeX macros are transformed to improve the html document.
  1. \code{weaveRhtml} looks for the LaTeX macros \code{\author},
  \code{\title} and \code{\date} at the beginning of the input text.
  If these macros are found their arguments are used to construct a simple
  html-head.
  2. \code{\section\{...\}, \subsection\{...\}, \paragraph\{...\}} macros will be extrac
  to include some section titles, subsection titles, paragraph titles in bold face fonts
  Additionally a simple table of contents is generated.
  3. Text lines between \code{\begin\{center\}} and \code{\end\{center\}}
  are centered.
  4. Text lines between \code{\begin\{quote\}} and \code{\end\{quote\}}
  are shifted a little bit to the right.
  5. Text lines between \code{\begin\{itemize\}} and \code{\end\{itemize\}}
  define a listing. The items of such a list have to begin with \code{\item}.
  6. \code{\emh\{xyz\}} is transformed to \code{<i>xyz</i>} -- \code{xyz} will appear it
  7. \code{\texttt\{xyz\}} is transformed to \code{<code>xyz</code>} -- this is formated
```

```
}
\value{
  a html file is generated
}
\references{ \url{http://www.eecs.harvard.edu/~nr/noweb/intro.html} }
\author{Hans Peter Wolf}
\seealso{  \code{\link{weaveR}}, \code{\link{tangleR}} }
\examples{
\dontrun{
## This example cannot be run by examples() but should be work in an interactive R sessi
  weaveRhtml("testfile.rev","testfile.tex")
  weaveR("testfile.rev")
}
## The function is currently defined as
weaveRhtml<-function(in.file,out.file){
  # german documentation of the code:
  # look for file webR.pdf, P. Wolf 060910
  ...
}
}
\keyword{file}
\keyword{documentation}
\keyword{programming}
```