TANGLE und WEAVE mit R — selbstgemacht

Peter Wolf

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

20.09.2006, 10.11.2006

1 Verarbeitungsprozesse

⟨*process*⟩≡
```
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 betriebssystemunabhngiges R-Programm fr den TANGLE-Verarbeitungsproze beschrieben. Dieses kann Demonstrationsbeispielen beigefgt werden, auerdem kann fr die Definition alternativer Verarbeitungsvorstellungen Anregungen geben.
In dem vorliegenden Vorschlag werden die verwendeten Modulnamen eines Quelldokumentes in den Code als Kommentarzeilen aufgenommen, so da sie spter fr die Navigation verwendet werden knnen. Weiterhin werden alle Wurzeln aus dem Papier expandiert, sofern nicht eine andere Option angegeben wird.

2.2 Die grobe Struktur der Lsung

Der TANGLE-Proze soll mittels einer einzigen Funktion gelst werden. Sie bekommt den Namen `tangleR`. Als Input ist der Name der Quelldatei zu bergeben. 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. ber Optionen lt sich die Menge der zu expandierender Wurzeln bestimmen. Die Funktion besitzt folgende Struktur:

⟨*define-tangleR*⟩≡
```
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⟩
  ⟨initialisiere Variable fr Output tangleR⟩
  ⟨ermittle Namen und Bereiche der Code-Chunks tangleR⟩
  if(expand.root.start){
    ⟨expandiere Start-Sektion tangleR⟩
  }
  ⟨ermittle Wurzeln tangleR⟩
  ⟨expandiere Wurzeln tangleR⟩
  ⟨korrigiere ursprnglich mit @ versehene Zeichengruppen tangleR⟩
  ⟨speichere code.out tangleR⟩
}
```

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.

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

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

⟨*lese Datei ein* `tangleR`⟩≡
```
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
```

⟨*substituiere mit @ versehene Zeichengruppen* `tangleR`⟩≡
```
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 bergnge anzeigt, lassen sich schnell die Bereiche der Text-Chunks ermitteln.

⟨*entferne Text-Chunks* `tangleR`⟩≡
```
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 drfen 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 Qualitt der Zeilen von `code.ch` anzeigt: `D` steht fr Definition, `U` fr Verwendungen und `C` fr normalen Code-Zeilen. `code.n` hlt die Zeilenanzahl,

⟨*stelle Typ der Zeilen fest* `tangleR`⟩≡
⟨*knacke ggf. Zeilen mit mehrfachen Chunk-Uses* `tangleR`⟩
```
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"
```

⟨*knacke ggf. Zeilen mit mehrfachen Chunk-Uses* `tangleR`⟩≡
```
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 fr sptere Manipulationen werden in den Bezeichnern die Zeichenketten `<<` `>>` bzw. `>>=` entfernt. Die Zeilennummern der Code-Chunks-Anfnge bezglich `code.ch` stehen auf `code.a`, die Enden auf `code.z`.

⟨*ermittle Namen und Bereiche der Code-Chunks* `tangleR`⟩≡
```
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 da diese in der R-Funktion nicht mehr zu bercksichtigen waren. Dieses `awk`-Programm mit dem Namen `pretangle.awk` sei hier eingefgt, vielleicht ist es im Zusammenhang mit einer S-PLUS-Implementation hilfreich.

⟨*ein* `awk`*-Programm zur Entfernung von Text-Chunks aus einem Quellfile*⟩≡
```
#
# 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 mu initialisiert werden.

⟨*initialisiere Variable fr Output* `tangleR`⟩≡
```
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 zunchst 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 bertragen und Verfeinerungsverwendungen werden auf `code.stack` durch ihre Definitionen ersetzt.

⟨*expandiere Start-Sektion* `tangleR`⟩≡
```
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⟩
   }
}
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 bertragen. Falls die nchste keine Codezeile ist, mu es sich um die Verwendung einer Verfeinerung handeln. In einem solchen Fall wird die nchste Verfeinerung identifiziert und der Bezeichner der Verfeinerung wird durch seine Definition ersetzt. Nicht definierte, aber verwendete Chunks fhrten anfangs zu einer Endlosschleife. Dieser Fehler ist inzwischen behoben `051219`. Eine entsprechende nderung wurde auch fr nicht-start-chunks fllig.

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

```
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 berlegungen sollen neben `start` auch alle weiteren Wurzeln gesucht und expandiert werden. Wurzeln sind alle Definitionsnamen, die nicht verwendet werden.

⟨*ermittle Wurzeln* `tangleR`⟩≡

```
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 verluft die Expansion der Wurzel wie die von `start`. Jedoch werden etwas umfangreichere Kommentare eingebaut.

⟨*expandiere Wurzeln* `tangleR`⟩≡
```
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⟩
    }
    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.

⟨*transferiere Codezeilen oder ersetze Verfeinerungen bis Ende erreicht* `tangleR`⟩≡
```
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
}
```

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

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

⟨*Beispiel* – `tangleR`⟩≡
```
  tangleR("out")
```

2.5 Help-Page

⟨*define-tangleR-help*⟩≡

```
\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 Verrgerung

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

⟨*Definition einer unntigen Funktion*⟩≡

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

Die Funktion besitzt folgenden Aufbau:

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

Das Encoding wird mittels tcltk festgestellt.

⟨*initialisiere* weaveR⟩≡
```
require(tcltk)
pat.use.chunk<-paste("<","<(.*)>",">",sep="")
pat.chunk.header<-paste("^<","<(.*)>",">=",sep="")
pat.verbatim.begin<-"\\\\begin\\{verbatim\\}"
pat.verbatim.end<-"\\\\end\\{verbatim\\}"
pat.leerzeile<-"^(\\ )*$"
.Tcl("set xyz [encoding system]"); UTF<-tclvalue("xyz")
UTF<-0<grep("utf",UTF)
if(UTF) cat("character set: UTF\n") else cat("character set: not utf\n")
```

ALT: Zunchst fixieren wir die Suchmuster fr wichtige Dinge. Auerdem stellen wir fest, ob R auf UTF-8-Basis arbeitet. Versuche zum UTF-Problem:

```
is.utf<-"\"<c3>\""==deparse("\xc3")
is.utf<-substring(intToUtf8(as.integer(999)),2,2)==""
```

⟨*old*⟩≡
```
lcctype<-grep("LC_CTYPE",strsplit(Sys.getlocale(),";")[[1]],value=T)
UTF<-(1==length(grep("UTF",lcctype)))
is.utf<-substring(intToUtf8(as.integer(999)),2,2)==""
UTF<- UTF | is.utf
```

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

⟨*lese Datei ein* `weaveR`⟩≡
```
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)
```

⟨*substituiere mit @ versehene Zeichengruppen* `weaveR`⟩≡
```
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 brigens:
sowie in der ersten Zeile ein .

⟨*ersetze Umlaute* `weaveR`⟩≡
```
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 mssen die besonderen Zeichengruppen zurckbersetzt
werden.

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

⟨*schreibe die Makrodefinition fr Randnummern vor die erste Zeile*⟩≡
```
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 mssen wir die modifizierte Variable `input` wegschreiben.

⟨*schreibe Ergebnis in Datei* `weaveR`⟩≡
```
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. Auerdem 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-Anfnge 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-Anfngen und den Text-Chunkanfngen ermittelt, wobei die `USE`-Zeilen wieder ausgeschlossen werden. Alle brigen Zeilen werden als Textzeilen eingestuft.

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

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

⟨*checke verbatim-Zeilen* `weaveR`⟩≡
```
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)
```

⟨*checke Header- und Use-Zeilen* `weaveR`⟩≡
```
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))]
```

⟨*checke normale Code-Zeilen* `weaveR`⟩≡
```
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))]
```

⟨*belege Typ-Vektor* `weaveR`⟩≡
```
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"
```

⟨*erstelle Output* `weaveR`⟩≡
```
⟨erledige Text-Chunk-Starts weaveR⟩
⟨extrahiere Header-, Code- und Verwendungszeilen weaveR⟩
⟨schreibe Header-Zeilen weaveR⟩
⟨schreibe Code-Verwendungszeilen weaveR⟩
⟨schreibe Code-Zeilen weaveR⟩
⟨setze Code in Text-, Header- und Verwendungszeilen weaveR⟩
```

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

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

⟨*extrahiere Header-, Code- und Verwendungszeilen* `weaveR`⟩≡
```
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]
```

⟨*schreibe Header-Zeilen* `weaveR`⟩≡
```
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
```

ber `Rchunkno` lassen sich Verweise erstellen. Hilfreich knnte dazu zwei Makros sein, um die Vorgehensweise von `\label` und `\ref` abzubilden. Dieses knnte 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.
```

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

```
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")
```

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

```
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 demnchst ein anderes Verwendung finden muss. Ein Weg besteht darin, aus dem Zeichenvorrat ein ungebrauchtes Zeichen auszuwhlen, dessen `catcode` zu verndern und dann dieses zu verwenden. Nachteilig ist bei diesem Zeichen, dass verschiedene Editoren dieses nicht darstellen knnen. Darum ist es besser ein ungewhnliches, aber darstellbares Zeichen zu verwenden. Zum Bespiel knnte man `\343` verwenden, so dass die Zeile unten lauten wrde:
`input[code.index]<-paste("\\verb\343",code.lines,"\343\\newline")`
Um ganz sicher zu gehen, dass dieses Zeichen akzeptiert wird, knnte man den `catcode` so verndern: `\catcode‘\343=12"` – also in R:
`\\\catcode‘\\\343=12"` im oberen Bereich des Dokumentes einfgen.

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

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

Code im Text wird auf zwei Weisen umgesetzt:
a) Zerlegung von Zeilen in Wrter. Wrter 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.

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

⟨*suche in Zeilen des Typs nach Code im Text* `code.im.text.index` `weaveR`⟩≡
```
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 nchsten Text-Chunk beschrieben. Anschlieend wird die Zeile mit tt-gesetzten Code-Stcken wieder zusammengebaut.

⟨*ersetze zusammenhngende Wortstcke* `weaveR`⟩≡

```
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 zusammenhngende 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.

⟨*checke und ersetze Code im Text mit Leerzeichen* `weaveR`⟩≡
```
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[],"aAzsplitAa",sep=""))[-1],"\\]\\]")
     zs<-unlist(lapply(zs,function(y){ res<-rbind("]]",y[])[-1]; res }))
     gsub("aAzsplitAa","",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
}
```

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

⟨*teste Funktion* `weaveR`⟩≡
```
  ⟨definiere Funktion weaveR⟩
weaveR("out.rev"); system("cat q|latex out.tex")
```

3.2 Help-Page

⟨*define-weaveR-help*⟩≡

```
\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 Lsung wurde im wesentlichen kopiert. Die Funktion besitzt folgenden Aufbau:

⟨*define-weaveRhtml*⟩≡
```
weaveRhtml<-function(in.file,out.file){
  # german documentation of the code:
  # look for file webR.pdf, P. Wolf 060920
  ⟨initialisiere weaveRhtml⟩
  ⟨lese Datei ein weaveRhtml⟩
  ⟨entferne Kommentarzeichen weaveRhtml⟩
  ⟨substituiere mit @ versehene Zeichengruppen weaveRhtml⟩
  ⟨stelle Typ der Zeilen fest weaveRhtml⟩
  ⟨erstelle Output weaveRhtml⟩
  ⟨ersetze Umlaute weaveRhtml⟩
  ⟨korrigiere ursprnglich mit @ versehene Zeichengruppen weaveRhtml⟩
  ⟨formatiere berschriften weaveRhtml⟩
  ⟨definiere einfachen head weaveRhtml⟩
  ⟨setze Schriften um weaveRhtml⟩
  ⟨entferne unbrauchbare Makros weaveRhtml⟩
  ⟨schreibe Ergebnis in Datei weaveRhtml⟩
  "ok"
}
```

Zunchst fixieren wir die Suchmuster fr wichtige Dinge. Auerdem stellen wir fest, ob R auf UTF-8-Basis arbeitet.

⟨*initialisiere* `weaveRhtml`⟩≡
```
require(tcltk)
verbose<-FALSE
pat.use.chunk<-paste("<","<(.*)>",">",sep="")
pat.chunk.header<-paste("^<","<(.*)>",">=",sep="")
pat.verbatim.begin<-"\\\\begin\\{verbatim\\}"
pat.verbatim.end<-"\\\\end\\{verbatim\\}"
pat.leerzeile<-"^(\\ )*$"
.Tcl("set xyz [encoding system]"); UTF<-tclvalue("xyz")
UTF<-0<grep("utf",UTF)
if(verbose){ if(UTF) cat("character set: UTF\n") else cat("character set: not utf\n") }
```

⟨*old*⟩+≡
```
  lcctype<-grep("LC_CTYPE",strsplit(Sys.getlocale(),";")[[1]],value=T)
  UTF<-(1==length(grep("UTF",lcctype)))
  is.utf<-substring(intToUtf8(as.integer(999)),2,2)==""
  UTF<- UTF | is.utf
  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.

⟨*lese Datei ein* `weaveRhtml`⟩≡
```
  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)
```

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

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

Umlaute sind ein Dauerbrenner. Hinweis: im richtigen Code steht unten brigens:
sowie in der ersten Zeile ein .

⟨*ersetze Umlaute* `weaveRhtml`⟩≡
```
  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 mssen die besonderen Zeichengruppen zurckbersetzt
werden.

⟨*korrigiere urspnglich mit @ versehene Zeichengruppen* `weaveRhtml`⟩≡
```
  #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 fr Graphik-Eintrge mittels `includegraphics`. `get.head.argument` ermittelt fr 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 fr kurzzeitige Schriftenwechsel. `transform.structure.command`

⟨*initialisiere* `weaveRhtml`⟩+≡

```
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"))
}
```

⟨*formatiere berschriften* `weaveRhtml`⟩≡

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

⟨*formatiere Strukturkommandos* `weaveRhtml`⟩≡
```
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="")
}
```

⟨*definiere einfachen head* `weaveRhtml`⟩≡
```
## 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>"
)
```

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

⟨*zentriere und quote* `weaveRhtml`⟩≡
```
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)
```

⟨*setze Schriften um* `weaveRhtml`⟩≡
```
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 mssen wir die modifizierte Variable `input` wegschreiben.

⟨*schreibe Ergebnis in Datei* `weaveRhtml`⟩≡
```
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. Auerdem 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-Anfnge 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-Anfngen und den Text-Chunkanfngen ermittelt, wobei die `USE`-Zeilen wieder ausgeschlossen werden. Alle brigen Zeilen werden als Textzeilen eingestuft.

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

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

⟨*behandle verbatim-Zeilen* `weaveRhtml`⟩≡
```
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>")
```

⟨*checke Header- und Use-Zeilen* `weaveRhtml`⟩≡
```
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))]
```

⟨*checke normale Code-Zeilen* `weaveRhtml`⟩≡
```
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))]
```

⟨*belege Typ-Vektor* `weaveRhtml`⟩≡
```
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"
```

⟨*erstelle Output* `weaveRhtml`⟩≡
⟨*zentriere und quote* `weaveRhtml`⟩
⟨*erledige Text-Chunk-Starts* `weaveRhtml`⟩
⟨*ersetze Befehl zur Bildeinbindung*⟩
⟨*extrahiere Header-, Code- und Verwendungszeilen* `weaveRhtml`⟩
⟨*schreibe Header-Zeilen* `weaveRhtml`⟩
⟨*schreibe Code-Verwendungszeilen* `weaveRhtml`⟩
⟨*schreibe Code-Zeilen* `weaveRhtml`⟩
⟨*setze Code in Text-, Header- und Verwendungszeilen* `weaveRhtml`⟩

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

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

⟨*ersetze Befehl zur Bildeinbindung*⟩≡
```
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
}
```

⟨*extrahiere Header-, Code- und Verwendungszeilen* `weaveRhtml`⟩≡
```
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])
```

⟨*schreibe Header-Zeilen* `weaveRhtml`⟩≡
```
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
```

⟨*schreibe Code-Verwendungszeilen* `weaveRhtml`⟩≡
```
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>")
```

⟨*ddd*⟩≡
```
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 demnchst ein anderes Verwendung finden muss. Ein Weg besteht darin, aus dem Zeichenvorrat ein ungebrauchtes Zeichen auszuwhlen, dessen `catcode` zu verndern und dann dieses zu verwenden. Nachteilig ist bei diesem Zeichen, dass verschiedene Editoren dieses nicht darstellen knnen. Darum ist es besser ein ungewhnliches, aber darstellbares Zeichen zu verwenden. Zum Bespiel knnte man `\343` verwenden, so dass die Zeile unten lauten wrde:
`input[code.index]<-paste("\\verb\343",code.lines,"\343\\newline")`
Um ganz sicher zu gehen, dass dieses Zeichen akzeptiert wird, knnte man den `catcode` so verndern: `\catcode‘\343=12"` – also in R:
`\\\catcode‘\\\343=12"` im oberen Bereich des Dokumentes einfgen.

⟨*schreibe Code-Zeilen* `weaveRhtml`⟩≡
```
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>")
}
```

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

Code im Text wird auf zwei Weisen umgesetzt:
a) Zerlegung von Zeilen in Wrter. Wrter 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.

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

⟨*suche in Zeilen des Typs nach Code im Text* `code.im.text.index` `weaveRhtml`⟩≡
```
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 nchsten Text-Chunk beschrieben. Anschlieend wird die Zeile mit tt-gesetzten Code-Stcken wieder zusammengebaut.

⟨*ersetze zusammenhngende Wortstcke* `weaveRhtml`⟩≡

```
  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(..))
```

⟨*old*⟩+≡

```
  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 zusammenhngende 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.

⟨*checke und ersetze Code im Text mit Leerzeichen* `weaveRhtml`⟩≡
```
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[],"aAzsplitAa",sep=""))[-1],"\\]\\]")
     zs<-unlist(lapply(zs,function(y){ res<-rbind("]]",y[])[-1]; res }))
     gsub("aAzsplitAa","",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.

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

⟨*teste Funktion* `weaveRhtml`⟩≡
```
⟨definiere-weaveRhtml⟩
#weaveRhtml("/home/wiwi/pwolf/tmp/vskm16.rev")
weaveRhtml("/home/wiwi/pwolf/tmp/doof")
#weaveRhtml("/home/wiwi/pwolf/tmp/aufgabenblatt2.rev")
```

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

4.2 Help-Page

⟨*define-weaveRhtml-help*⟩≡

```
\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}
```