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, 30.8.2007, 12.11.2007, 04.09.2008, 31.10.2008

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
# noweave -delay -index webR.rev > webR.tex; latex webR; dvipdf webR
```

1.1 Changes

01/2009

- weaveR: adding use-of-chunk-infos in TeX document

10/2008

- weaveRhtml: now pre tags are used instead of code tags for verbatim outputs
- weaveR: par and vspace are included at the beginning of chunks
- tangleR: now expansion of not root chunks is allowed
- tangleR: the name of saved files after tangling must not have extension R
- tangleR: tangling without preserving comment lines with link information Quellenhinweisen

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,
         insert.comments=TRUE,add.extension=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 16⟩
  ⟨korrigiere ursprünglich mit @ versehene Zeichengruppen tangleR 18⟩
  ⟨speichere code.out tangleR 19⟩
}
print("OK")
```

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")
if(insert.comments) 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⟩
   }
}
if(insert.comments) 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)){ # verwende nur wirkliche Wurzeln
#  h<-!is.na(match(roots,expand.roots))
#  roots<-roots[h]; root.no<-root.no[h]
#}
if(!is.null(expand.roots)){ # verwende gegebene(n) Knoten
  h<-!is.na(match(def.names,expand.roots))
  root.no<-seq(def.names)
  roots<-def.names[h]; root.no<-root.no[h]
}
if(0==length(roots)) cat("Sorry, no chunk for expanding found!\n")
if(exists("DEBUG")) print(roots)
```

2.3.6 Expansion der Wurzeln

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

15 ⟨*OLD – expandiere Wurzeln* `tangleR` *– OLD* 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 17⟩
    }
    code.out<-c(code.out,paste("##:",roots[r],"##",sep=""),
                         paste("#:",root.no[r],sep=""))
  }
}
```

16 ⟨*expandiere Wurzeln* tangleR 16⟩≡

```
if(exists("DEBUG")) cat("bearbeite Sektion-Nr./Name\n")
roots<-unique(roots)
for(r in seq(along=roots)){
  # if(exists("DEBUG")) cat(root.no[r],":",roots[r],", ",sep="")
  if(exists("DEBUG")) cat(which(def.names==roots[r]),":",roots[r],", ",sep="")
  if(any(ch.no <-def.names==roots[r])){
    ch.no     <-seq(along=def.names)[ch.no]; rows<-NULL
    if(insert.comments) code.out<-c(code.out,
          # # # paste("#",root.no[r],":",sep=""), new 071114
                        paste("##",roots[r],":##",sep=""))
    for(i in ch.no){
       if((code.a[i]+1)<=code.z[i]){
         # # # rows<-c(rows, (code.a[i]+1):code.z[i]) # new:
            h<-code.a[i]+1
            rows<-c(rows, h:code.z[i])
            if(insert.comments) code.ch[h]<-paste("C#",i,":NeWlInE",code.ch[h],sep="")
            h<-code.z[i]
            if(insert.comments) code.ch[h]<-paste(code.ch[h] ,"NeWlInEC#:",i,sep="")
       }
    }
    code.stack<-code.ch[rows]
    code.stack<-unlist(strsplit(code.stack,"NeWlInE")) # new
    repeat{
      ⟨transferiere Codezeilen oder ersetze Verfeinerungen bis Ende erreicht tangleR 17⟩
    }
    if(insert.comments) 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.

17 ⟨*transferiere Codezeilen oder ersetze Verfeinerungen bis Ende erreicht* `tangleR` 17⟩≡

```
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]){
        if(insert.comments){
          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<-c(code.ch[row.no[1]:row.no[2]], code.stack)
        }
      }
    } # end of for
  } else code.stack <-code.stack[-1] # ignore not defined chunks!
  # 051219
}
```

18 ⟨*korrigiere ursprünglich mit @ versehene Zeichengruppen* `tangleR` 18⟩≡

```
code.out<-gsub("DoSpCloseKl-esc",">>",gsub("DoSpOpenKl-esc","<<",code.out))
```

19 ⟨*speichere* `code.out` `tangleR` 19⟩≡

```
if(missing(out.file)||in.file==out.file){
  out.file<-sub("\\.([A-Za-z])*$","",in.file)
}
if(add.extension&&0==length(grep("\\.R$",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

20 ⟨*Beispiel* – `tangleR` 20⟩≡

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

2.5 Help-Page

21 ⟨*define-tangleR-help* 21⟩≡

```
\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,insert.comments=TRUE,add.extension=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 }
  \item{insert.comments}{ if TRUE comments with chunk inofs will be added to the code }
  \item{add.extension}{ if TRUE output file name will get extension .R }
}
\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?

22 ⟨*Definition einer unnötigen Funktion* 22⟩≡

```
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:

23 ⟨*define-weaveR* 23⟩≡
```
weaveR<-function(in.file,out.file,show.code=TRUE,show.text=TRUE,
                 replace.umlaute=TRUE){
  # german documentation of the code:
  # look for file webR.pdf, P. Wolf 050204, 060517, 070307, 070830
  ⟨initialisiere weaveR 24⟩
  ⟨lese Datei ein weaveR 26⟩
  ⟨substituiere mit @ versehene Zeichengruppen weaveR 27⟩
  ⟨stelle Typ der Zeilen fest weaveR 36⟩
  ⟨extrahiere Header-, Code- und Verwendungszeilen weaveR 42⟩
  ⟨bestimme ggf. Zeile für Objekt-Index 57⟩
  ⟨bestimme ggf. Zeile für Liste der Chunks 51⟩
  ⟨bestimme used-in Informationen 46⟩
  ⟨bestimme ggf. Menge der eigenen R-Objekte und sammle Object Index Infos 58⟩
  ⟨erstelle Output weaveR 62⟩
  ⟨ersetze Umlaute weaveR 28⟩
  ⟨korrigiere ursprünglich mit @ versehene Zeichengruppen weaveR 30⟩
  ⟨schreibe die Makrodefinition für Randnummern vor die erste Zeile 31⟩
  ⟨entferne ggf. Code oder Text 33⟩
  ⟨entferne ggf. Umlaut-TeX-Makros aus Code-Zeilen 34⟩
  ⟨schreibe Ergebnis in Datei weaveR 35⟩
}
```

Das Encoding wird mittels tcltk festgestellt.

24 ⟨*initialisiere* weaveR 24⟩≡
```
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<length(grep("utf",UTF))
if(⟨DEBUG-Flag gesetzt 74⟩){
  if(UTF) cat("character set: UTF\n") else cat("character set: not utf\n")
}
if(!UTF){
      char267<-eval(parse(text='"\\267"'))
}
```

ALT: Zunächst fixieren wir die Suchmuster für wichtige Dinge. Außerdem 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)==""
```

25 ⟨*old* 25⟩≡

```
lcctype<-grep("LC_CTYPE",strsplit(Sys.getlocale(),";")[[1]],value=TRUE)
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.

26 ⟨*lese Datei ein* `weaveR` 26⟩≡

```
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
try(if(replace.umlaute&&UTF && any(is.na(iconv(input,"","LATIN1")))){
    # LATIN1-Dok :
    input<-iconv(input,"LATIN1","")
})
length.input<-length(input)
```

27 ⟨*substituiere mit @ versehene Zeichengruppen* `weaveR` 27⟩≡

```
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*.

28 ⟨*ersetze Umlaute* `weaveR` 28⟩≡

```
if(replace.umlaute){
 if(!UTF){
 # im Tcl/Tk-Textfenster eingegeben -> iso-8859-1 (man iso-8859-1 / Latin1 / unicode
    pc<-eval(parse(text='"\\283"'))  # UTF-8-pre-char
    uml.utf.8 <-eval(parse(text='"\\244\\266\\274\\204\\226\\234\\237"'))
    uml.latin1<-eval(parse(text='"\\344\\366\\374\\304\\326\\334\\337"'))
    input<-chartr(uml.utf.8,uml.latin1,gsub(pc,"",input)) # utfToLatin1
    input<-gsub(substring(uml.latin1,7,7),"{\\\\ss}",input) # replace sz
    uml.pattern<-eval(parse(text='"(\\344|\\366|\\374|\\304|\\326|\\334)"'))
    input<-gsub(uml.pattern,"\\\\\"\\1",input)  # replace Umlaute ae->&aeuml;
# replace Umlaute &aeuml;->&auml;
    input<-chartr(substring(uml.latin1,1,6),"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)
 }
 if(⟨DEBUG-Flag gesetzt 74⟩){
  cat("german Umlaute replaced\n")
 }
}
```

Diese Darstellung ließ sich unter utf-8 nicht anzeigen!

29 ⟨*alte Umlautersetzung weaveR* 29⟩≡
```
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)
}
```

Vor dem Wegschreiben müssen die besonderen Zeichengruppen zurückübersetzt werden.

30 ⟨*korrigiere ursprünglich mit @ versehene Zeichengruppen* `weaveR` 30⟩≡
```
input<-gsub("DoSpCloseKl-esc",">>",gsub("DoSpOpenKl-esc","<<",input))
input<-gsub("DoEckCloseKl-esc","]]",gsub("DoEckOpenKl-esc","[[",input))
```

Es kam beim Umschalten auf Code zu unschönen Zeilenabständen, die durch Vorschalten von `\\par\\vspace{-\\parskip}` ausgeschaltet werden sollen. Das schwierigste Makro ist `\makemarginno`, mit dem von Text- auf Code-Chunk umgestellt wird. Es wird der Paragraph abgeschlossen, ein Vorschub von einem halben Absatzabstand eingebaut, alle Umschaltungskommandos umgesetzt der Zähler für die Randnummern hochgesetzt und der Zähler eingetragen.

31 ⟨*schreibe die Makrodefinition für Randnummern vor die erste Zeile* 31⟩≡
```
input[1]<-paste(
     "\\newcounter{Rchunkno}",
     "\\newcounter{IsInCodeChunk}\\setcounter{IsInCodeChunk}{1}",
     "\\newcommand{\\codechunkcommands}{\\relax}",
     "\\newcommand{\\textchunkcommands}{\\relax}",
     "\\newcommand{\\makemarginno}",
          "{\\par\\vspace{-0.5\\parskip}\\codechunkcommands",
          "\\stepcounter{Rchunkno}",
          "\\setcounter{IsInCodeChunk}{1}",
          "\\noindent\\hspace*{-3em}",
          "\\makebox[0mm]{\\arabic{Rchunkno}}\\hspace*{3em}}",
     input[1],sep="")
```

32 ⟨*unused* 32⟩≡
```
\par\vspace{-0.5\parskip}\codechunkcommands
\stepcounter{Rchunkno}
\hspace*{-3em}\makebox[0mm]{\arabic{Rchunkno}}
\hspace*{3em}
```

Falls nur Text oder nur Code gewünscht wird, muss der Rest entfernt werden.

33 ⟨*entferne ggf. Code oder Text* 33⟩≡
```
if(show.code==FALSE){
   input[code.index] <-"."
   input[use.index] <-":"
   an<-grep("\\\\begin(.*)\\{document\\}",input)[1]
   if(length(tit<-grep("\\\\maketitle",input))>0) an<-tit
   input[an]<-paste(input[an],"${}^*$ --- only the TEXT of the paper ---\\par")
}
if(show.text==FALSE){
   input<-sub("^%.*","%",input)
   an<-grep("\\\\begin(.*)\\{document\\}",input)[1]
   en<-grep("\\\\end(.*)\\{document\\}",input)[1]
   text.index<-which(line.typ=="TEXT")
   text.index<-text.index[an<text.index&text.index<en]
   input[c(text.index, verb.index)] <-"."
   if(length(tit<-grep("\\\\maketitle",input))>0) an<-tit
   input[an]<-paste(input[an],"${}^*$ --- only the CODE of the paper ---\\par")
}
```

In den Code-Zeilen wollen wir die Umlaute wieder zurücksetzen. Dieses wollen wir nur machen, wenn keine Modulzeilen gefunden werden.

34 ⟨*entferne ggf. Umlaut-TeX-Makros aus Code-Zeilen* 34⟩≡
```
if(replace.umlaute && 0<length(ind<-grep(".newline.verb",input))){
   ind2<-grep("langle(.*)rangle",input[ind]); if(0<length(ind2)) ind<-ind[-ind2]
   if(0<length(ind)){
    inp<-input[ind];
    if(!UTF){
    # im Tcl/Tk-Textfenster eingegeben -> iso-8859-1 (man iso-8859-1 / Latin1 / unicode
      # \"a -> ae, ... oe, ue, Ae, Oe, Ue, ß
      u<-uml.latin1<-unlist(strsplit(eval(parse(text='"\\344\\366\\374\\304\\326\\334\\3
      inp<-gsub('\\\\"a',u[1],inp);inp<-gsub('\\\\"o',u[2],inp);inp<-gsub('\\\\"u',u[3],
      inp<-gsub('\\\\"A',u[4],inp);inp<-gsub('\\\\"O',u[5],inp);inp<-gsub('\\\\"U',u[6],
      inp<-gsub(".\\\\ss}",u[7],inp)
    }else{
      # pc<-eval(parse(text='"\\283"'))  # UTF-8-pre-char
      uml.utf.8 <-eval(parse(text='"\\283\\244\\283\\266\\283\\274\\283\\204\\283\\226\\
      u<-substring(uml.utf.8,1:7,1:7)
      inp<-gsub('\\\\"a',u[1],inp);inp<-gsub('\\\\"o',u[2],inp)
      inp<-gsub('\\\\"u',u[3],inp);inp<-gsub('\\\\"A',u[4],inp)
      inp<-gsub('\\\\"O',u[5],inp);inp<-gsub('\\\\"U',u[6],inp) ##{
      inp<-gsub(".\\\\ss}",u[7],inp)
    }
    input[ind]<-inp
   }
}
if(⟨DEBUG-Flag gesetzt 74⟩){
  cat("german Umlaute in code lines inserted\n")
}
```

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

35 ⟨*schreibe Ergebnis in Datei* `weaveR` 35⟩≡
```
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="")
base::cat(input,sep="\n",file=out.file)
base::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.

36 ⟨*stelle Typ der Zeilen fest* `weaveR` 36⟩≡
```
⟨checke Leer-, Textzeilen weaveR 37⟩
⟨checke verbatim-Zeilen weaveR 38⟩
⟨checke Header- und Use-Zeilen weaveR 39⟩
⟨checke normale Code-Zeilen weaveR 40⟩
⟨belege Typ-Vektor weaveR 41⟩
```

37 ⟨*checke Leer-, Textzeilen* `weaveR` 37⟩≡
```
empty.index<-grep(pat.leerzeile,input)
text.start.index<-which("@"==substring(input,1,1))
```

38 ⟨*checke verbatim-Zeilen* `weaveR` 38⟩≡
```
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)
```

39 ⟨*checke Header- und Use-Zeilen* `weaveR` 39⟩≡
```
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))]
```

40 ⟨*checke normale Code-Zeilen* weaveR 40⟩≡

```
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=FALSE]
b<-a[a[,2]!=c(-1,a[-length(a[,1]),2]),,drop=FALSE]
a<-rep(0,length.input); a[b[,1]]<-b[,2]
a<-cumsum(a); a[code.start.index]<-0
## a[empty.index]<-0 ?? this was not a good idea 070709
code.index<-which(a>0)
code.index<-code.index[is.na(match(code.index,use.index))]
```

41 ⟨*belege Typ-Vektor* weaveR 41⟩≡

```
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"

is.code.line<-text.start.indicator<-rep(0,length.input)
text.start.indicator[1]<-1; text.start.indicator[text.start.index]<-1
text.start.indicator<-cumsum(text.start.indicator)
is.code.line[code.start.index]<-0-text.start.indicator[code.start.index]
is.code.line<-cummin(is.code.line)
is.code.line<-(text.start.indicator+is.code.line) < 1
is.code.line[code.start.index]<-FALSE
## TSI<<-text.start.index; CSI<<-code.start.index; UI<<-use.index # just for debugging
```

42 ⟨*extrahiere Header-, Code- und Verwendungszeilen* weaveR 42⟩≡

```
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]
```

3.1.1 Referenzen und Indizes – Überlegungen und Entscheidungen

Ideen. Das literate programming bietet Entwicklern zwei wesentliche Pluspunkte: erstens lassen sich schwierige Probleme durch Zerlegung schrittweise lösen und zweitens können Lösungen in einer durch den Autor bestimmten Reihenfolge verfasst werden. Ein Leser kann dann Schritt für Schritt die Bausteine der Lösung nachvollziehen und verstehen. Das Gesamtkunstwerk ergibt sich aus der Aggregation der einzelnen Bausteine. Sowohl bei der Erstellung wie auch bei späteren Auseinandersetzungen mit einer literaten Lösung spielt der Zusammenhang der Bausteine eine zentrale Rolle. Deshalb ist es notwendig, dass Zusatzinformationen wie Verweise und Indizes über Beziehungen Auskunft geben. Ein Blick in D. E. Knuth (1982): *TEX the program* zeigt, dass solche Lesehilfen sehr komfortable gestaltet werden können. Die hier beschriebene `weave`-Komponente soll ebenfalls zweckmäßige Zusatzinformationen integrieren.

Wichtige Zusatzinformationen. Die folgende Aufzählung zeigt Typen von Zusatzinformationen in der Reihenfolge, wie ihre Wichtigkeit eingeschätzt wird:

expand in Wird ein Problem in Teilprobleme zerlegt, ist die wichtigste Information eine Angabe der Stellen, an denen die Teilprobleme weiter diskutiert werden. An der Stelle der Verwendung eines Code-Chunk sollte deshalb ein Verweis auf die Stelle der Definition des verwendeten Code-Chunk zu finden sein. *The definition is found in: xx*.

used in Wird eine Teillösung betrachtet, so interessiert, an welchen Stellen diese Teillösung Verwendung findet. Hierfür sollte an einem Code-Chunk eine Link-Information auf den Chunk bzw. die Chunks zu finden sein, in denen er als Verfeinerung eingesetzt wird. *This chunk is used in the chunks with the numbers: xx, yy*.

chunk list Wird nach einer Teillösung gesucht, benötigt man eine Übersicht der kleinen Lösungen. Dieses ist für kleine Probleme nicht so wichtig, jedoch schnell, wenn die Lösung umfangreicher wird. Eine Liste der Code-Chunk-Namen wird dann erforderlich. *The list of chunks with chunk numbers and page information: name – definition in xx, yy, zz – page of first definition*.

extensions Manchmal werden Bausteine erweitert, weil später unter einer schon behandelten Überschrift noch ein weiterer Aspekt ergänzen wird. Beispielsweise können wir uns vorstellen, dass Sammelbecken wie *definiere Ausgabe-Routinen* oder *definiere elementare Zugriffsfunktionen* an verschiedenen Stellen gefüllt werden. Bei strukturellen Veränderungen einer Teil-Lösung kann es erforderlich sein, alle zu einem Namen zugehörenden Chunks zu finden. Die Erweiterungsinformation kann an den Chunks oder aber an der Übersicht der Chunks angeheftet werden. *See chunk list: yy, zz*.

object index Letztlich ist es gerade für Entwickler wichtig, die Vorkommnisse bestimmter Objekte aufspüren zu können. Mit Suchfunktionen von Editoren ist das an sich kein Problem, jedoch ist in der Papierversion ein Objekt-Index überlegen. Die Realisation im *TEX – The Program* zeigt uns eine Luxusversion, bei der auf jeder Seite verwendete Variablen mit dem Ort ihrer Definition und ihrer Qualität aufgeführt sind. *Object Index: name – numbers of chunks using the object*.

Design-Entscheidungen für `weaveR`. Für die genannten Punkte soll für `weaveR` eine Umsetzung gefunden werden. Dazu sind Design-Entscheidungen zu fällen, die das Erscheinungsbild festlegen. Ausgehend von dem

Verwendungszweck werden die Code-Chunk-Referenzen über Code-Chunk-Nummern geregelt und nicht wie andernorts zu sehen über Seitenzahlen. Um Ablenkungen beim Lesen gering zu halten, sollen keine Text eingefügt werden, wie sie beispielsweise im letzten Abschnitt in englischer Sprache vorgeschlagen wurden. Stattdessen sollen einfache Symbole aus der Mengenlehre die Art der Beziehungen ausdrücken.

expand in Bei der Verwendung eines Code-Chunk soll hinter dem Namen – jedoch vor der den Namen abschließenden Klammer – die Nummer des (ersten) Chunk mit der Definition eingefügt werden. Falls es keine Definition gibt, erscheint dort ein `NA`.

used in Die Header-Zeile eines Code-Chunk soll um Verweise auf die Chunks, die die Definition einbauen, erweitert werden. Dazu sollen die Nummern der rufenden Chunks sowie als Trennzeichen "$\subset$" rechts neben dem Chunk-Namen notiert werden. Mit dem Teilmengenzeichen wird angezeigt, dass der vorliegende Chunk ein Teil von größeren Lösungen ist.

chunk list Die Chunk-Liste soll eine Kopie alle Code-Chunk-Header mit den Verwendungshinweisen darstellen. Zusätzlich sollen die Seitenzahlen des ersten Vorkommens der Definitionen angegeben werden.

extensions Mit anderen Web-Systemen lassen sich Referenzen an eine Code-Chunk-Definition angefügen, die auf Ergänzungsdefinitionen hinweisen. In der vorliegenden Fassung von `weaveR` werden Erweiterungsinformationen nur in der Chunk-Liste notiert. Dazu wird die Zahl, die die Code-Chunk-Nummer des ersten Vorkommens angibt, um die weiteren Nummern der Folgedefinition erweitert und durch das naheliegende Zeichen "$\cup$" abgetrennt.

object index In einem Objekt-Index sollen die Objekt-Namen mit den Nummern ihres Einsatzes gesammelt werdeb. Dazu sollen alle Zeichenketten vor Zuweisungsoperatoren untersucht und hieraus die Menge der Objekte gebildet werden. Namen mit nur einem Zeichen sollen ausgeklammert werden. Darüberhinaus sollen sich per Hand weitere Objekt-Namen festlegen lassen.

Optionen der Gestaltung. Wesentliche Dinge müssen sein. Dementsprechend soll der Punkt "expand in" immer wie beschrieben umgesetzt werden. Es könnte Situationen geben, in denen man "used in"-Informationen abstellen möchte. Die Übersicht "chunk list" inklusive der "extensions"-Informationen soll dagegen nur sofern gewünscht erstellt werden. Denn für kurze Abhandlungen oder bei sehr wenigen verschiedenen Header-Namen ist eine solche Übersicht verzichtbar. Entsprechendes gilt für den Objekt-Index. Zu diesem ist neben der automatischen Generierung der Liste eine Integration weiterer gewünschter Objektnamen notwendig. Denn einerseits kann der Automatismus eventuell Objektnamen übersehen. Andererseits ist es auch denkbar, dass ein Anwender Objektnamen, die nur aus einem Buchstaben bestehen, in die Liste aufnehmen will.

Syntax zu den Optionen. Für die beschriebenen Einflussmöglichkeiten muss es Befehlselemente geben, die prinzipiell entweder als Parameter beim `weaveR`-Aufruf oder direkt ins Quelldokument eingebracht werden könnten. Damit Dokumente ergebnisgleich rekonstruiert werden können, wurde die Parameter-Lösung ausgeschlossen. Optionen werden also über spezielle Anweisungen im Quelldokument festgelegt.

- `@no.used.in.infos`:
  Falls ein solcher Eintrag gefunden wird, wird keine Information bezüglich einer Verwendung eines Code-Chunk am Definitionsort eingebunden.

- `@list.of.chunks`:
  In Anlehnung an das Makro `\listoffigures`, jedoch für die Quellebene adaptiert, fordert der Anwender durch eine Zeile beginnend mit `@list.of.chunks` die Liste der Code-Chunk Namen an. Diese wird dort platziert, wo `@list.of.chunks` gefunden wird.
- `@index.of.objects`:
  Entsprechend wird mit `@index.of.objects` der Objekt-Index an der Fundstelle des Keyword in das Dokument eingesetzt.
- `@index`:
  Eigene Objektnamen werden durch Zeilen festgelegt, die mit `@index` beginnen und in denen nach diesem Keyword die Objektnamen folgen. Es werden jedoch nur Objektnamen in die Übersicht der Objekte aufgenommen, die auch in mindestens einem Code-Chunk gefunden werden.
- `@index.reduced`:
  Wird dieser Eintrag gefunden, wird keine automatische Suche nach Objektnamen durchgeführt und der Objektindex enthält nur manuell festgelegte Einträge.

Damit ist der Handlungsspielraum des Anwenders festgelegt und wir können uns der Umsetzung widmen.

3.1.2 Referenzen und Indizes – Umsetzungen

Für die Umsetzung lassen sich eine Reihe von Informationen und Variablen verwenden, die an dieser Stelle bekannt sind. Die zentrale Variable, die auf der aktuelle Dokument verwaltet wird, heißt `input`. Diese enthält zeilenweise das Quelldokument. Im Laufe des Verarbeitungs-Prozesses wird diese Variable solange aufbereitet, bis sie zum Schluss auf ihr das fertige LaTeX-Dokument steht.

expand in :

Das Grundprinzip der Umsetzung ist einfach:

1. suche alle Zeilen mit Chunk-Verwendungen
2. extrahiere aus diesen die Chunk-Namen der verwendeten Chunks
3. suche die extrahierten Chunk-Namen in der Menge aller Code-Chunk-Definitionen
4. schreibe die Code-Chunk-Nummer der jeweils gefundenen Definition hinter den Chunk-Namen an der Verwendungsstelle ein.

Die Chunk-Nummer der (ersten) Definition eines verwendeten Chunks wird während der Formatierung der Zeilen mit Verwendungen umgesetzt, bei der geeignete LaTeX-Befehle ergänzt werden. Es ist zu beachten, dass auch *Verwendungszeilen* behandelt werden, die in Text-Chunks gefunden werden.

Im Detail werden für die Formatierung zuerst die Leerzeichen am Anfang der Verwendungszeilen auf `leerzeichen.vor.use` (ergänzt um ein zusätzliches Leerzeichen) und die Zeilen ohne diese Leerzeichen auf `use.lines` abgelegt. Jede Verwendungszeile wird in einem Schleifendurchgang der for-Schleife untersucht: Dabei werden die Chunk-Namen solange aufgebrochen, bis alle in einzelnen Elementen von `uli` stehen. Die Stellen von `uli` mit Chunk-Namen werden auf `cand` gemerkt. `uli[-cand]` enhält alle Nicht-Chunk-Namen. Nun können die Chunk-Namen gesucht und um die Nummer `ref.no` ergänzt werden. Die mit Nummern und TeX-Klammern versehenen `uli`-Einträge werden zusammengepackt und auf `use.lines` zurückgespielt. Zuvor müssen die Nicht-Chunk-Namen jedoch ggf. noch wie Code behandelt werden!

43 ⟨*schreibe Code-Verwendungszeilen* `weaveR` 43⟩≡

```
# get use lines
use.lines<-input[use.index]; is.use.lines.within.code<-is.code.line[use.index]
# remove and save leeding blanks
leerzeichen.vor.use<-paste("\\verb|",
                           sub("[^ ](.*)$"," ",use.lines),
                           "|",sep="") ## plus 1 Leerzeichen
use.lines<-substring(use.lines,nchar(leerzeichen.vor.use)-7)
# loop along use lines
for(i in seq(along=use.lines)){
  # get single line
  uli<-use.lines[i]
  # split chunk names and other strings
  repeat{
    if(0==length(cand<-grep("<<(.*)>>",uli))) break
    uli.h<-gsub("(.*)<<(.*)>>(.*)","\\1bReAkuSeChUnK\\2bReAk\\3",uli)
    uli<-unlist(strsplit(uli.h,"bReAk"))
  }
  # find chunk names
  cand<-grep("uSeChUnK",uli); uli<-sub("uSeChUnK","",uli)
  # find chunk numbers of (first) definition
  ref.no<-match(uli[cand],code.chunk.names)
  # include number of definition chunk
  uli[cand]<-paste("$\\langle${\\it ",uli[cand],"} ",ref.no,"$\\rangle$",sep="")
  ⟨poliere use.lines, die aufgesplittet auf uli stehen 44⟩
  use.lines[i]<-paste(uli,collapse="")
}
# store modified use lines
input[use.index]<-ifelse(is.use.lines.within.code,
      paste("\\rule{0mm}{0mm}\\newline",leerzeichen.vor.use,use.lines,"%",sep=""),
      paste(leerzeichen.vor.use,use.lines,sep=""))
```

Für die Bearbeitung der Verwendungszeilen ist zu beachten, dass diese sowohl in einem Code-Chunk aber auch in einem Text-Chunk vorkommen können. Verwendungen im Text sind nicht weiter zu behandeln. Jedoch können bei einem Einsatz in einem Code-Chunk in der Zeile noch weitere Code-Stücke stehen. Die Zeile selbst ist so in Zeichenketten aufgeteilt und auf `uli` abgelegt, dass verwendete Code-Chunk-Namen anhand von `cand` erkannt werden. Alle übrigen Einträge auf also `uli[-cand]` sind Code und müssen entsprechend durch eine `\verb`-Konstruktion eingepackt (*poliert*) werden.

44 ⟨*poliere* `use.lines`*, die aufgesplittet auf* `uli` *stehen* 44⟩≡

```
# formating code within use references, in code chunk a little different
if(length(uli)!=length(cand)){
  if(is.use.lines.within.code[i]){
    # within code chunks: code (but no the chunk names) has to be escaped
    if(!UTF){
        uli[-cand]<-paste("\\verb",char267,uli[-cand],char267,sep="") #050612
    }else{
        uli[-cand]<-paste("\\verb\140",uli[-cand],"\140",sep="") #060516
    }
  }
}
```

used in :

Die *used-in*-Info wird zusammen mit anderen formatierungstechnischen Aufbereitungen der Header-Zeilen erledigt. Da für jeden einzelnen Chunk zu klären ist, wo er eingebaut wird, ist eine Schleife über die Chunk zur Klärung naheliegend.

1. füge TEX-Makro zur Erzeugung der laufenden Nummer am Rand an
2. ermittle aus allen Verwendungszeilen die enthaltenen Code-Chunk-Namen
3. suche die verwendeten Namen in der Liste aller Chunk-Namen
4. merke zu jeder Chunk-Definition die gefundenen rufenden Chunks
5. füge Referenz-Informationen an.

Betrachtet man einen isolierten Chunk, dann müssen in diesem die Header in unterschiedlicher Weise modiziert werden. Die einfachste Operation bestehen darin, eine laufende Nummer am Rand anzubringen. Weiter ist das Ersterscheinen von Code-Chunk-Namen hinter dem Namen zu vermerken. Drittens sind die "used in"-Informationen anzufügen.

Für die laufende Nummer wird das LATEX-Marko `\makemarginno` eingebaut, das später die Nummern erzeugt und Vorschub und Einzug regelt. Als zweites wird hinter dem Code-Chunk-Namen die Sektionsnummer des ersten Vorkommnisses des Namens eingetragen. Nur bei Erweiterungsdefinitionen, werden sich die laufende Randnummer und die Nummer direkt hinter dem Chunk-Namen unterscheiden. Die Position der Header-Zeilen kann der Variablen `code.start.index` entnommen werden.

45 ⟨*ergänze Randnummern und Ersterscheinen in Header-Zeilen* weaveR 45⟩≡
```
# find section numbers
no<-seq(along=code.start.index)
# find first occurences of code chunks
def.ref.no<-match(gsub("\\ ","",code.start.lines),
                  gsub("\\ ","",code.start.lines))
# construct modified header lines
code.start.lines<-paste("\\makemarginno ",
                        "$\\langle${\\it ",code.start.lines,"}\\ $",def.ref.no,
                        "\\rangle",ifelse(no!=def.ref.no,"+",""),"\\equiv$",sep="")
# save modified header lines
input[code.start.index]<-code.start.lines
```

Nun widmen wir uns dem Problem, die "used in"-Information zu ermitteln und bereitzustellen.

Die "used-in"-Infos erhalten wir, indem wir zunächst die Menge der relevanten Header-Zeilen extrahieren. Dann suchen wir alle Zeilen mit Verwendungen, von denen aber nur die interessieren, zu denen es bereits eine Definition gibt. Diese legen wir auf `names.use.cand` ab. Die Einträge (Zeilen) von `names.use.cand` gehen wir nacheinander durch und extrahieren aus diesen alle Chunk Verwendungen. Auf `names.use` werden dann alle Chunk-Verwendungen gesammelt sowie auf `lines.use` die Zeilen, in denen die Verwendungen stehen.

Mit allen Chunk-Namen und allen Verwendungsinformationen können wir nun alle irgendwo verwendeten Code-Chunks durchgehen und die jeweiligen Nummern der Verwendungsorte ermitteln: `used.in.no`. Mit diesen werden zum Schluss Zeichenketten mit den used in-Informationen zusammengesetzt. Unter `used.in.message` und `lines.used.in.message` werden die textuellen Anhängsel und die relevanten Zeilennummern abgelegt.

46 ⟨*bestimme used-in Informationen* 46⟩≡
```
ref.infos.found<-FALSE
# extract lines containing calls of other code chunks
lines.use<-which(line.typ=="USE"&is.code.line)
include.use.infos<-0==length(grep("^@no.used.in.infos",input))
if(include.use.infos&&length(lines.use)>0){
 ⟨ermittle Menge der Header 47⟩
 # lines with uses of code chunks
 ⟨ermittle Namen und Zeilen der verwendeten Code-Chunk 48⟩
 # chunk uses found: names (names.use) and lines (lines.use)
 # find headers that have been used, their lines and compute used-in-info
 # remove brackets etc.
 names.header<-sub(paste("^.*<","<(.*)>",">.*",sep=""),"\\1",names.header)
 ind<-!is.na(match(names.header,names.use))
 names.header.used<-names.header[ind]; lines.header.used<-code.start.index[ind] #
 if((anz<-length(names.header.used))>0){
   ref.infos.found<-TRUE; used.in.message<-rep("",anz)
   lines.used.in.message<-rep(0,anz)
   # find for each header of names.header.use the numbers of section of their uses
   ⟨ermittle für verwendete Header-Namen die rufenden Chunks und erstelle Meldung 49⟩
 }
}
```

Die Header-Menge enthält keine Wiederholungen von Chunk-Namen!

47 ⟨*ermittle Menge der Header* 47⟩≡
```
# find header lines
names.header<-input[code.start.index]
# extract set header names: remove "<", ">" and characters not belonging to the name
names.header.uniq<-sub(paste(pat.chunk.header,".*",sep=""),"\\1",
                       unique(sort(names.header)))
```

Lege die Namen der Code-Chunks, die verwendet werden, auf der Variablen `names.use` und die zugehörigen Zeilen auf `lines.use` ab.

48 ⟨*ermittle Namen und Zeilen der verwendeten Code-Chunk* 48⟩≡
```
names.use.cand<-input[lines.use]; l.u<-n.u<-NULL
for(ii in seq(along=lines.use)){ # aufknacken mehrerer Chunks in einer Code-Zeile
   h<-names.use.cand[ii]
   repeat{
     # if(!exists("max.wd")) max.wd<-10; max.wd<-max.wd-1; if(max.wd<1) break
     last<-sub(paste("^.*<","<(.*)>",">.*",sep=""),"\\1",h) # extract name
     if(last!=h){ # something found during substitution => chunk use found
       l.u<-c(l.u,lines.use[ii]); n.u<-c(n.u, last)
       h<-sub(paste("^(.*)<","<.*>",">.*",sep=""),"\\1",h) # rm identified chunk use
       if(nchar(h)==0) break
     } else break # no more chunk uses in line ii
   }
}
names.use<-n.u; lines.use<-l.u
```

49 ⟨*ermittle für verwendete Header-Namen die rufenden Chunks und erstelle Meldung* 49⟩≡
```
for(i in 1:anz){
     idx.found<-which(names.header.used[i]==names.use)
     l<-lines.use[idx.found]
     # find number of chunks calling names.header.used[i]
     used.in.no<-unique(unlist(lapply(l,function(x) sum(code.start.index<x)))) #
     # construct message and save line number of input that has to be changed
     used.in.message[i]<-paste("{\\quad$\\subset$ ",
                               paste(used.in.no,collapse=", "),"}")
     lines.used.in.message[i]<-lines.header.used[i]
}
```

Damit haben wir die "used-in"-Referenzinformationen ermittelt und aufbereitet auf `lines.used.in.message` und `used.in.message` abgelegt. Diese werden dann während der Output-Erstellung in dem Chunk ⟨*ergänze used-in-Infos, Chunk-Index und ggf. Object-Index*⟩ ausgewertet, also an passender Stelle in `input` eingetragen. Den Kern wollen wir jedoch hier vorstellen:

50 ⟨*lege "used in" Infos auf input ab* 50⟩≡
```
# include Referenzinformationen
if(include.use.infos&&ref.infos.found)
   input[lines.used.in.message]<-paste(input[lines.used.in.message],
                                       used.in.message)
```

chunk list :

Die Erstellung einer einfachen Chunk-Liste ist völlig einfach. Erst die Berücksichtigung von Erweiterungsdefinitionen und Seitenhinweise erfordert einige Überlegungen. Der Grundalgorithmus lässt sich so formulieren:

1. suche alle Header-Zeilen
2. ergänze die Erweiterungsinformationen
3. ergänze Seiteninformationen
4. führe noch ein paar Formatierungen durch.

Zunächst stellen wir fest, ob überhaupt der Wunsch nach einer Chunk-Liste besteht und merken uns die Position, an der die Liste eingefügt werden soll.

51 ⟨*bestimme ggf. Zeile für Liste der Chunks* 51⟩≡
```
pos.chunk.list<-grep("^@list.of.chunks",input)
```

Für die Chunk-Liste gehen wir pragmatisch vor und suchen aus `input` alle Header-Zeilen und entfernen alle unwichtigen Infos.

52 ⟨*bereite Chunk Index vor* 52⟩≡
```
# merke Header fuer Chunk Index
chunk.index<-NULL
ind<-0<length(pos.chunk.list)
if(ind){
  # Randnummer entfernen
  chunk.index<-sub("\\\\makemarginno.","",input[code.start.index])
  # + Zeichen entfernen
  chunk.index<-sub("rangle[+]*.equiv","rangle",chunk.index)
  first<-match(chunk.index,chunk.index)       # jeweils erste Chunks finden
  no.ext<-split(seq(along=chunk.index),first) # Nummern gleicher Chunks suchen
  no.ext<-unlist(lapply(no.ext,function(x){
         if(length(x)==1) " " else paste("\\cup",paste(x[-1],collapse="\\cup"))
                                } ))   # Erweiterungsnummern als String ablegen
  chunk.index<-unique(chunk.index); first<-unique(first)   # gleiche entfernen
  # Erweiterungs-Infos einbauen
  chunk.index<-paste(sub(".rangle.*","",chunk.index),no.ext,
                     sub(".*rangle","\\\\rangle",chunk.index),sep="")
  ⟨trage pageref-Einträge ein 55⟩
  chunk.index<-sort(chunk.index)              # sortieren
}
```

Für die Ergänzung von Seitenzahlen an die Chunk-Liste müssen wir an die Chunks LaTeX-Labels anfügen. Dazu generieren wir auf Basis der Chunk-Nummern Labelnamen. Die Labelnamen beginnen alle mit `CodeChunkLabel` und werden gefolgt von der codierten Nummer. Die Ziffer 0 wird durch ein `A` ersetzt, die Ziffer 1 durch ein `B` usw., so dass dem Code-Chunk mit der Nummer 123 das Label `CodeChunkLabelBCD` zugeordnet wird. Ein geeigneter Label-Makroaufruf wird an geeigneter Stelle den Header-Zeilen hinzugefügt.

53 ⟨*schreibe Label an Code-Chunk-Headers* 53⟩≡
```
label<-function(no){
  ⟨definiere Funktion find.label 54⟩
  label<-sapply(no,find.label)
  paste("\\label{",label,"}",sep="")
}
input[code.start.index]<-paste(input[code.start.index],
                               label(seq(along=code.start.index)))
```

Die Funktion `find.label` benötigen wir zweimal. Deshalb definieren wir sie in einem eigenen Chunk.

54 ⟨*definiere Funktion* `find.label` 54⟩≡
```
find.label<-function(no){
    label<-paste("CodeChunkLabel",
        paste(LETTERS[1+as.numeric(substring((no+10000),2:5,2:5))],
                    collapse=""),sep="")
}
```

Nachdem der Chunk-Index auf der Variablen `chunk.index` angekommen ist, können wir `\pageref`-Aufrufe an diese Variablen anhängen. Die Seitennummern wird LaTeX dann anfügen.

55 ⟨*trage pageref-Einträge ein* 55⟩≡
```
pageref<-function(no){
  ⟨definiere Funktion find.label 54⟩
  label<-sapply(no,find.label)
  paste("\\pageref{",label,"}",sep="")
}
chunk.index<-paste(chunk.index,"\\dotfill",pageref(first))
```

Zur Komplettierung wird nur noch etwas TeX-Kosmetik gemacht.

56 ⟨*erledige Restarbeiten für Chunk-Index* 56⟩≡
```
if(1<length(chunk.index)){
  chunk.index<-paste(chunk.index,collapse="\\\\") # newline ?
  chunk.index<-paste("{\\paragraph{Code Chunk Index}\\small",
                     "\\rule{0mm}{0mm}\\\\[1.5ex]",
                     chunk.index,"\\\\\\rule{0mm}{0mm}}")
}
```

extensions :

Diesen Punkt haben wir unter Chunk-Liste bereits erledigt.

object index :

Im groben wird ein Objekt-Index in folgenden Schritten erstellt, sofern keine Einschränkung durch `@index.reduced` gegeben ist:

1. suche mögliche Objektnamen durch Untersuchung von Zuweisungen
2. ergänze die gefundenen Namen um manuell angeforderte Namen
3. untersuche alle Code-Chunk, ob in ihnen Objekte der Liste der Objektnamen verwendet werden
4. bilde aus dem Untersuchungsergebnis den Objekt-Index.

Den Index der Objektnamen bilden wir nur, wenn ein Index-Wunsch besteht.

57 ⟨*bestimme ggf. Zeile für Objekt-Index* 57⟩≡
```
pos.obj.idx<-grep("^@index.of.objects",input)
```

Zur Ermittlung der Objekt-Namen werden alle Code-Zeilen mit Zuweisungen und in diesen alle Zeichen vor einem Zuweisungspfeil extrahiert. Die gefundenen Strings werden dann um die Zeichen bereinigt, die nicht zu einem Namen gehören oder auf kompliziertere Konstruktionen hinweisen. Auf diese Weise sollten die meisten selbst zugewiesenen Objekte gefunden werden. Dann werden noch alle Objekte ergänzt, die per `@index` manuell festgelegt worden sind. Falls kein Objekt-Index erwünscht wird, also ein Eintrag mit `@index.of.objects` nicht gefunden wird, bleibt `obj.index` leer. Wird zusätzlich `@index.reduced` gefunden, werden nur manuell eingetragene Objektnamen in den Index aufgenommen.

58 ⟨*bestimme ggf. Menge der eigenen R-Objekte und sammle Object Index Infos* 58⟩≡

```
obj.set<-obj.index<-NULL

ind<-0<length(pos.obj.idx) && 0==length(grep("^@index.reduced",input))
if(ind) {
  # Kandidatensuche
  a<-input[code.index]
  a<-sub("<-.*","",a[grep("<-",a)]) # Zuweisungen suchen
  a<-gsub(" ","",a[!is.na(a)])      # Leerzeichen entfernen
  # Indizes und Unternamen entfernen:
  a<-sub("\\[.*","",a); a<-sub("\\$.*","",a)
  a<-a[grep("^[a-zA-Z.]",a)]        # Beginn mit Nicht-Ziffer
  a<-a[grep("^[a-zA-Z.0-9_]*$",a)]  # erlaubte Zeichen
  a<-a[nchar(a)>1]                  # nur echte Strings merken
  obj.set<-sort(unique(a))          # Zusammenfassung
}
# explizite angegebene Namen
ind<-0<length(pos.obj.idx) && 0<length( a<-grep("^@index[^o]",input,value=TRUE) )
if(ind){
  a<-sub("^@index *","",a); a<-gsub(","," ",a);
  a<-unlist(strsplit(gsub(" +"," ",a)," "))
  obj.set<-sort(unique(c(a,obj.set))) # set of object names
}
if(length(obj.set)>0){
  ⟨ermittle über alle Code-Chunks die verwendeten Objekte 59⟩
}
```

Nachdem die Kandidatenliste der Objekte feststeht, muss noch die Verwendung der Objekte in den einzelnen Chunks geprüft werden. Dieser Schritt kostet relativ viel Zeit. In einer Schleife über alle Code-Chunks wird jeweils der Code extrahiert und in diesen nach Vorkommnissen aller Objekt-Kandidaten gesucht. Als Ergebnis wird eine zweizeilige Matrix gebildet, in deren zweiten Zeile die Chunk-Nummern stehen und in der ersten Zeile die Indizes der gefundenen Objekte bzgl. der Kandidatenliste. Dann können durch Aufsplitten des Code-Chunk-Nummern-Vektors gemäß der Objekt-Indizes die Informationen zu den einzelnen Objekte als Listenelemente zusammengefasst werden. Mit diesen Informationen erhält man schnell die grobe Objekt-Index-Liste `object.index`, die nur noch formatiert werden muss.

59 ⟨*ermittle über alle Code-Chunks die verwendeten Objekte* 59⟩≡
```
if(0<(anz<-length(code.start.index))){
  obj.used.in<-matrix(0,2,0)
  for(no in 1:anz){
    # extract code chunk no
    c.start<-code.start.index[no]+1
    c.end<-text.start.index[which(c.start<text.start.index)[1]]-1
    if(is.na(c.end)) c.end<-length(input)
    if(c.end<c.start) next
    a<-paste("",input[c.start:c.end],"") # code von code chunk no
    # check occurance of all candidats
    h<-sapply(obj.set,function(x)
       0<length(grep(paste("[^a-zA-Z.0-9]",x,"[^a-zA-Z.0-9]",sep=""),a)))
    if(any(h)) obj.used.in<-cbind(obj.used.in,rbind(which(h),no))
  }
  # obj.used[2,] shows chunk numbers, obj.used[1,] that candidates
  a<-split(obj.used.in[2,],obj.used.in[1,])
  # list element i stores the numbers of chunks where object i has been found
  a<-lapply(a,function(x){
         x<-paste(names(x)[1],"\\quad$\\in$",paste(x,collapse=", "))
              })
  obj.index<-paste(unlist(a),collapse="\\\\\n"); names(obj.index)<-NULL
}
```

Auf `obj.index` steht die Rohfassung des Objekt-Indizes, der in ⟨*ergänze used-in-Infos, Chunk-Index und ggf. Object-Index*⟩ weiterverarbeitet bzw. auf `input` abgelegt wird. Etwas Formatierung führt zur einbaufähigen Liste.

60 ⟨*formatiere Objekt-Index* 60⟩≡
```
if(0<length(obj.index)){
  obj.index<-gsub("_","\\\\_",obj.index)
  obj.index<-paste(obj.index,collapse="\\\\") # newline ?
  obj.index<-paste(
        "{\\paragraph{Object Index}\\footnotesize\\rule{0mm}{0mm}\\\\[1.5ex]",
        obj.index,"\\\\\\rule{0mm}{0mm}\\par}")
}
```

Jetzt können wir noch ein paar Restarbeiten erledigen.

61 ⟨*ergänze used-in-Infos, Chunk-Index und ggf. Object-Index* 61⟩≡
⟨*lege "used in" Infos auf input ab* 50⟩
⟨*bereite Chunk Index vor* 52⟩
⟨*erledige Restarbeiten für Chunk-Index* 56⟩
⟨*formatiere Objekt-Index* 60⟩
```
if(0<length(pos.obj.idx)) input[pos.obj.idx]<-obj.index
if(0<length(pos.chunk.list)) input[pos.chunk.list]<-chunk.index
```

3.1.3 Integration der Ergänzungen in die Output-Erzeugung

62 ⟨*erstelle Output* `weaveR` 62⟩≡
⟨*erledige Text-Chunk-Starts* `weaveR` 63⟩
⟨*ergänze Randnummern und Ersterscheinen in Header-Zeilen* `weaveR` 45⟩
⟨*schreibe Code-Verwendungszeilen* `weaveR` 43⟩
⟨*schreibe Code-Zeilen* `weaveR` 64⟩
⟨*setze Code in Text-, Header- und Verwendungszeilen* `weaveR` 65⟩
⟨*ergänze used-in-Infos, Chunk-Index und ggf. Object-Index* 61⟩
⟨*schreibe Label an Code-Chunk-Headers* 53⟩

Abschließend seien noch ein paar Bemerkungen zur Reihenfolge der Umsetzung gemacht. Dazu führen wir die relevanten Chunk-Namen in der Ordnung auf, wie sie späer in der Funktion auftauchen.

1. bestimme ggf. Zeile für Objekt-Index und für Liste der Chunks: Nach allgemeinen Untersuchungen des Inputs und der Extraktion von Header- und Code-Zeilen wird ermittelt, ob ein Objekt-Index und eine Chunk-Liste erstellt werden soll. Die Ergebnisse (Zeilen der gewünschten Einfügungsorte) werden auf den Variablen `pos.obj.idx` und `pos.chunk.list` vermerkt.
2. bestimme used-in Informationen: Es werden die `used-in`-Infos gesammelt und in `used.in.message` und `lines.used.in.message` abgelegt.
3. bestimme ggf. Menge der eigenen R-Objekte: Dann wird die Menge der R-Objekte ermittelt und der Objekt-Index vorbereitet.
4. erstelle Output: Zum Schluss wird der Output erstellt. Dabei werden weitere Teilarbeiten erledigt:
   (a) ergänze Randnummern
   (b) schreibe Code-Verwendungszeilen
   (c) ergänze used-in-Infos: lege used in Infos auf input ab, bereite Chunk Index vor, erledige Restarbeiten für Chunk-Index, formatiere Objekt-Index
   (d) schreibe Label an Code-Chunk-Headers

Es ist festzustellen, dass bestimme Dinge für einzelne Jobs wiederholt umgesetzt werden. Jedoch hat dieses während der Entwicklung den Vorteil, dass die einzelnen Aufgabenbereiche getrennt geprüft und verbessert werden konnten und ggf. einzelne Elemente ausgeklammert werden können.

3.1.4 Hinweis für eigene Verweise

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.
```

Zur Aufbereitung der Text-Chunks-Starts.

Es müssen nur die Klammeraffen entfernt werden. Einfacher ist es, den entsprechenden Zeilen etwas Leeres zuzuweisen. Vor `textchunkcommands` wurde noch eine Paragraphenerzeugung vorangestellt, damit bei Schriftwechseln der Zeilenabstand stimmt.

63 ⟨*erledige Text-Chunk-Starts* `weaveR` 63⟩≡
```
## input[text.start.index]<-""
input[text.start.index]<-paste(
                        "\\ifodd\\value{IsInCodeChunk}",
                        "\\setcounter{IsInCodeChunk}{0}",
                        "\\vspace{-\\parskip}\\par\\hspace*{-\\parindent}",
                        "\\textchunkcommands\\fi",
                        sep="")
```

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.

64 ⟨*schreibe Code-Zeilen* `weaveR` 64⟩≡
```
if(!UTF){
  input[code.index]<-paste("\\rule{0mm}{0mm}\\newline\\verb",char267," ",code.lines," ",
}else{
  input[code.index]<-paste("\\rule{0mm}{0mm}\\newline\\verb\140",code.lines,
     "\140%") #060516 070706
}
```

65 ⟨*setze Code in Text-, Header- und Verwendungszeilen* `weaveR` 65⟩≡
```
typ<-"TEXT"
```
⟨*setze Code in Zeilen vom Typ* `typ` `weaveR` 66⟩
```
typ<-"HEADER"
```
⟨*setze Code in Zeilen vom Typ* `typ` `weaveR` 66⟩
```
typ<-"USE"
```
⟨*setze Code in Zeilen vom Typ* `typ` `weaveR` 66⟩

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.

66 ⟨*setze Code in Zeilen vom Typ* `typ` `weaveR` 66⟩≡
⟨*suche in Zeilen des Typs nach Code im Text* `code.im.text.index` `weaveR` 67⟩
```
if(0<length(code.im.text.index)){
  lines.to.check<-input[code.im.text.index]
```
⟨*ersetze zusammenhängende Wortstücke* `weaveR` 68⟩
⟨*checke und ersetze Code im Text mit Leerzeichen* `weaveR` 69⟩
```
  input[code.im.text.index]<-lines.to.check
}
```

67 ⟨*suche in Zeilen des Typs nach Code im Text* `code.im.text.index` `weaveR` 67⟩≡

```
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.

68 ⟨*ersetze zusammenhängende Wortstücke* `weaveR` 68⟩≡

```
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("\\\\","BaCkSlAsH",cand)
      cand<-gsub("([#$&_%{}])","\\\\\\1",cand) #2.1.0
      cand<-gsub("BaCkSlAsH","{\\\\char'134}",cand)
      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.

69 ⟨*checke und ersetze Code im Text mit Leerzeichen* `weaveR` 69⟩≡

```
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("\\\\","BaCkSlAsH",h);
        h<-gsub("([#$&_%{}])","\\\\\\1",h)
        h<-gsub("BaCkSlAsH","{\\\\char'134}",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
}
```

70 ⟨*checke und ersetze Code im Text mit Leerzeichen* `weaveR`*, old* 70⟩≡

```
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`.

71 ⟨*teste Funktion* `weaveR` 71⟩≡
⟨*definiere Funktion* `weaveR` (never defined)⟩
```
weaveR("out.rev"); system("cat q|latex out.tex")
```

3.2 Help-Page

72 ⟨*define-weaveR-help* 72⟩≡

```
\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,show.code=TRUE,show.text=TRUE,replace.umlaute=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{.tex} }
  \item{show.code}{ if FALSE the code will not be copied to the output file }
  \item{show.text}{ if FALSE the text will not be copied to the output file }
  \item{replace.umlaute}{ if TRUE german umlaute will be replaced by
   TeX sequences}
}
\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
  together with some information showing where are code is used.
  If you want to suppress these hints you can include a line
  beginning with the string \code{no.used.in.infos}.

  Hint for references:
  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\} }.

  Chunk Index:
  The user will get a list of all the code chunks if a line is found containing
  \code{@list.of.chunks}. However, there must be a minimum number of
```

```
  two list entries. Otherwise no index is appended to the text.

  Object Index:
  An index of the objects
  will be generated if a line is found with the string \code{@index.of.objects}.
  Object names consisting of one character are excluded from the search.
  However, you can define object names that should appear in the index by hand.
  This requires a line (or lines) in the document beginning with the string \code{@index
  followed by the names of the objects that should appear in the index.
  This features is helpful if some objects are missing on the
  list or if a name of an object to be listed is of length 1 only.
  Bear in mind that the computation of the object index is time consuming.
  If you do not want an index computed automatically you can include a line
  with the string \code{@index.reduced} at the beginning of the line. Then the
  object index will contain the object names of the \code{@index} statements only.
}
\value{
  a latex file is generated
}
\references{ by using noweb is an alternative approach, see:
            \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:

73 ⟨*define-weaveRhtml* 73⟩≡
```
weaveRhtml<-function(in.file,out.file,replace.umlaute=TRUE){
  # german documentation of the code:
  # look for file webR.pdf, P. Wolf 060920 / 070309 / 070830 / 071016
  ⟨initialisiere weaveRhtml 75⟩
  ⟨lese Datei ein weaveRhtml 77⟩
  ⟨entferne Kommentarzeichen weaveRhtml 79⟩
  ⟨substituiere mit @ versehene Zeichengruppen weaveRhtml 78⟩
  ⟨stelle Typ der Zeilen fest weaveRhtml 95⟩
  ⟨erstelle Output weaveRhtml 101⟩
  ⟨ersetze Umlaute weaveRhtml 80⟩
  ⟨korrigiere ursprünglich mit @ versehene Zeichengruppen weaveRhtml 83⟩
  ⟨formatiere Überschriften weaveRhtml 86⟩
  ⟨definiere einfachen head weaveRhtml 88⟩
  ⟨setze Schriften um weaveRhtml 92⟩
  ⟨entferne unbrauchbare Makros weaveRhtml 89⟩
  ⟨baue ggf. Rweb-Felder ein 90⟩
  ⟨integriere Newline hinter Zeilenumbrüchen 93⟩
  ⟨schreibe Ergebnis in Datei weaveRhtml 94⟩
}
```

74 ⟨*DEBUG-Flag gesetzt* 74⟩≡
```
exists("DEBUG")
```

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

75 ⟨*initialisiere* `weaveRhtml` 75⟩≡
```
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<length(grep("utf",UTF))
if(⟨DEBUG-Flag gesetzt 74⟩){
   if(UTF) cat("character set: UTF\n") else cat("character set: not utf\n")
}
```

76 ⟨*old* 25⟩+≡
```
lcctype<-grep("LC_CTYPE",strsplit(Sys.getlocale(),";")[[1]],value=TRUE)
UTF<-(1==length(grep("UTF",lcctype)))
is.utf<-substring(intToUtf8(as.integer(999)),2,2)==""
UTF<- UTF | is.utf
if(⟨DEBUG-Flag gesetzt 74⟩){
  if(UTF) cat("character set: UTF\n") else cat("character set: ascii\n")
}
```

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

77 ⟨*lese Datei ein* `weaveRhtml` 77⟩≡
```
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)
try(if(replace.umlaute&&UTF && any(is.na(iconv(input,"","LATIN1")))){  # LATIN1-Dok
          input<-iconv(input,"LATIN1","")
})
input<-gsub("\t","      ",input)
input<-c(input,"@")
length.input<-length(input)
```

78 ⟨*substituiere mit @ versehene Zeichengruppen* `weaveRhtml` 78⟩≡
```
input<-gsub("@>>","DoSpCloseKl-ESC",gsub("@<<","DoSpOpenKl-ESC",input))
input<-gsub("@\\]\\]","DoEckCloseKl-ESC",gsub("@\\[\\[","DoEckOpenKl-ESC",input))
```

Zuerst werden TeX-Kommentar-Zeichen vor Latex-Formel-Bildern entfernt, dann werden html-Kommentar-Zeichen vor der LaTeX-Formel vom unnötigen %-Zeichen befreit und zum Schluss werden alle LaTeX-Kommentare entfernt.

79 ⟨*entferne Kommentarzeichen* `weaveRhtml` 79⟩≡
```
input<-sub('^% (<p><img src="t)','\\1',input)
input<-sub('^%(<!--latex-)','\\1',input)
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*. `.Tcl("set XYZ [encoding system]"); tclvalue("XYZ")`, z.B.: iso8859-15, utf-8 `localeToCharset()` liefert: "ISO8859-1" bzw.: `[1] "UTF-8" "ISO8859-1"`

80 ⟨*ersetze Umlaute* `weaveRhtml` 80⟩≡
```
if(replace.umlaute){
 if(!UTF){
  # im Tcl/Tk-Textfenster eingegeben -> iso-8859-1 (man iso-8859-1 / Latin1 / unicode
    pc<-eval(parse(text='"\\283"'))  # UTF-8-pre-char
    uml.utf.8 <-eval(parse(text='"\\244\\266\\274\\204\\226\\234\\237"'))
    uml.latin1<-eval(parse(text='"\\344\\366\\374\\304\\326\\334\\337"'))
    input<-chartr(uml.utf.8,uml.latin1,gsub(pc,"",input)) # utfToLatin1
    input<-gsub(substring(uml.latin1,7,7),"&szlig;",input) # replace sz
    uml.pattern<-eval(parse(text='"(\\344|\\366|\\374|\\304|\\326|\\334)"'))
    input<-gsub(uml.pattern,"&\\1uml;",input)  # replace Umlaute ae->&aeuml;
    # replace Umlaute &aeuml;->&auml;
    input<-chartr(substring(uml.latin1,1,6),"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(⟨DEBUG-Flag gesetzt 74⟩) cat("german Umlaute replaced\n")
```

81 ⟨*old Umlaut-Replace* 81⟩≡
```
if(replace.umlaute){
 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(⟨DEBUG-Flag gesetzt 74⟩) cat("german Umlaute replaced\n")
```

82 ⟨*alternative zu* `!UTF` 82⟩≡
```
    # input<-iconv(input,"utf-8","")
    # input<-gsub("ß","&szlig;",input)
    # input<-gsub("(ä|ö|ü|Ä|Ö|Ü)","&\\1uml;",input)
    # input<-chartr("äöüÄÖÜ","aouAOU",input)
```

Vor dem Wegschreiben müssen die besonderen Zeichengruppen zurückübersetzt werden.

83 ⟨*korrigiere ursprünglich mit @ versehene Zeichengruppen* `weaveRhtml` 83⟩≡
```
#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` entstanden aus `transform.command`.

84 ⟨*initialisiere* `weaveRhtml` 75⟩+≡

```
get.argument<-function(command,txt,default="",kla="{",kle="}",
    dist=TRUE,not.found.info="no parameter"){
  ## print("get.argument")
  command<-paste("\\\\",command,sep="")
  if(0==length(grep(command,txt))) return(not.found.info)
  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]
       ifelse(k<=2,default,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)&&!is.na(en))
       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))){return(txt)}
  txt<-unlist(strsplit(paste(txt,collapse="\n"),command))
  tx<-unlist(lapply(txt[-1],function(x){
       n<-nchar(x); if(n<3) 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)||is.na(en)||(an+1)>(en-1)) x<-paste(x,collapse="") else
         paste(paste(if(an==1)""else x[1:(an-1)],collapse=""),
                       atag,paste(x[(an+1):(en-1)],collapse=""),etag,
                       paste(if(en==n)""else x[-(1:en)],collapse=""),sep="")
  }))
  unlist(strsplit(c(txt[1],tx),"\n"))
}
```

85 ⟨*some old function* 85⟩≡

```
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
}
```

86 ⟨*formatiere Überschriften* `weaveRhtml` 86⟩≡

```
# find sections, subsections, subsubsections, paragraphs
atag<-"<h2>"; etag<-"</h2>"; command<-"section"
⟨formatiere Strukturkommandos weaveRhtml 87⟩
sec.links<-command.links; sec.no<-com.lines
atag<-"<h3>"; etag<-"</h3>"; command<-"subsection"
⟨formatiere Strukturkommandos weaveRhtml 87⟩
subsec.links<-command.links; subsec.no<-com.lines
atag<-"<h4>"; etag<-"</h4>"; command<-"subsubsection"
⟨formatiere Strukturkommandos weaveRhtml 87⟩
subsubsec.links<-command.links; subsubsec.no<-com.lines
atag<-"<br><b>"; etag<-"</b>"; command<-"paragraph"
⟨formatiere Strukturkommandos weaveRhtml 87⟩
parsec.links<-command.links; parsec.no<-com.lines
sec.typ<-rbind(cbind(c(0,sec.no),1),cbind(c(0,subsec.no),2),
                        cbind(c(0,subsubsec.no),3),cbind(c(0,parsec.no),4))
sec.typ<-sec.typ[sec.typ[,1]!=0,,drop=FALSE]; contents<-" "
if(length(sec.typ>2)){
  ind<-order(sec.typ[,1]); sec.typ<-sec.typ[ind,,drop=FALSE]
  links<-c(sec.links,subsec.links,subsubsec.links,parsec.links)[ind]
  # append a column with *section numbers
  sec.typ<-cbind(sec.typ,"0")
  sec.counter<-subsec.counter<-subsubsec.counter<-par.counter<-0
  for(i in 1:nrow(sec.typ)){
   if(sec.typ[i,2]=="1"){
      sec.counter<-sec.counter+1
      subsec.counter<-subsubsec.counter<-par.counter<-0
      sec.typ[i,3]<-sec.counter
   }
   if(sec.typ[i,2]=="2"){
      subsec.counter<-subsec.counter+1
      subsubsec.counter<-par.counter<-0
      sec.typ[i,3]<-paste(sec.counter,".",subsec.counter,sep="")
   }
   if(sec.typ[i,2]=="3"){
      subsubsec.counter<-subsubsec.counter+1
      par.counter<-0
      sec.typ[i,3]<-paste(sec.counter,".",subsec.counter,".",subsubsec.counter,sep="")
   }
   if(sec.typ[i,2]=="4"){
      par.counter<-par.counter+1
      sec.typ[i,3]<-paste(sec.counter,".",subsec.counter,".",subsubsec.counter,".",
                                      par.counter,sep="")
   }
  }
  # construct table of contents with links
  contents<-paste(sec.typ[,3],links)
}
```

Dieser Chunk beschreibt die Umsetzung von Gliederungsbefehlen wie section, subsection, ...

87 ⟨*formatiere Strukturkommandos* `weaveRhtml` 87⟩≡

```
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(along=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(along=com.lines),
                                       "\">",sec,"</a>\n",sep="")
}
```

88 ⟨*definiere einfachen head* `weaveRhtml` 88⟩≡

```
## if(  DEBUG-Flag gesetzt  ) 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])]
  titel.titel<-gsub("\n","--",paste(titel,collapse="--"))
  titel.titel<-gsub("<br>","--",titel.titel)
  titel.titel<-gsub("\\\\","--",titel.titel)
  input[1]<-paste(collapse="\n",
  "<!--  generated by weaveRhtml --><html><head>",
  "<meta content=\"text/html; charset=ISO-8859-1\">",
  "<title>",titel.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>"
)
```

Achtung: Falls diese Dinge im Code vorkommen, wird es einen Fehler geben.

89 ⟨*entferne unbrauchbare Makros* `weaveRhtml` 89⟩≡

```
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]
```

Die Rweb-Funktionalität ist entstanden in Anlehnung an Seiten von Charlie Geyer `http://www.stat.umn.edu/~charlie/`, wie:
`http://www.stat.umn.edu/geyer/3011/examp/reg.html`

90 ⟨*baue ggf. Rweb-Felder ein* 90⟩≡

```
txt<-input
ind.Rweb<-grep("^<a name=\"codechunk.*<i>&lt;Rweb",txt) ; txt[ind.Rweb]
ind.p   <-grep(paste("^<","p>",sep=""),txt) ; txt[ind.p]
if(length(ind.p)>0){
  ind.Rweb.codes<-lapply(ind.Rweb,function(x) (x+1):(ind.p[ind.p>x][1]-1))
  ## if( DEBUG-Flag gesetzt ) print(ind.Rweb.codes)
  Rwebbegin<-paste(c(  # Rweb start
    '<hr SIZE=3><form onSubmit =" return checkData(this)"',
    '  action="http://rweb.stat.umn.edu/cgi-bin/Rweb/Rweb.cgi" ',
    '  enctype="multipart/form-data" method="post" target="_blank"><p>',
    '<label for="filedata" size=20>data frame (load LOCAL file): </label>',
    '<input type="file" name="FileData" id="filedata" size=30 >',
    '<br><br>',
    '<label for="urldata" size=20>data frame (via www -> URL): </label>',
    '<input type="TEXT" name="URLData" size=40 id="urldata" YyYyY>',
    '<br><br>',
    '<label for="Rcode">R-Code:</label>',
    '<textarea name="Rcode" rows="XxXxX" cols="80" id="Rcode">\n'
    ),collapse=" ")
  Rwebend<-  paste(c( # Rweb end
    '</textarea><p>',
    '<input type="submit" value="compute via Rweb">',
    '<input type="reset" value="reset">',
    '</form><hr SIZE=3>',
    ' '),collapse=" ")
  for(i in seq(along=ind.Rweb.codes)){
    h<-sub("<code>","",txt[ind.Rweb.codes[[i]]])
    h<-sub("<br>","",h)
    h<-sub("</code>","",h)
    h<-gsub("[&]nbsp[;]"," ",h)
    data.http<-""    # ; data.loc<-""
    ind<-length(grep("read.table",h))
    if(0<length(ind)) {
      h[ind]<-paste("#",h[ind],"# choose data by input field!!")
      ind<-ind[1]
      if(0<length(grep("http:",h[ind]))) {
         data.http<-sub(".*(http)","\"\\1",h[ind])
         data.http<-paste("value=",sub(".*(\".*\").*$","\\1",data.http),sep="")
      }
    }
    rb<-sub("YyYyY",data.http,Rwebbegin)
    h[1]<-paste(sub("XxXxX",as.character(length(h)),rb),h[1],sep="")
    h[length(h)]<-paste(h[length(h)],Rwebend)
    txt[ind.Rweb.codes[[i]]]<-h
  }
  input<-txt
}
```

91 ⟨*zentriere und quote* `weaveRhtml` 91⟩≡
```
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("\\\\begin\\{enumerate}","<ul>",input)
input<-sub("\\\\end\\{enumerate}","</ul>",input)
input<-sub("\\\\item\\[([^]]*)]","<br>\\1 ",input)
input<-sub("\\\\item","</li><li>",input)
```

92 ⟨*setze Schriften um* `weaveRhtml` 92⟩≡
```
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>")
}
if(0<length(h<-grep("\\\\emph",input))){
  input<-transform.structure.command("emph",input,
                     atag="<i>",etag="</i>",kla="{",kle="}")
}
if(0<length(h<-grep("\\\\textbf",input))){
  input<-transform.structure.command("textbf",input,
                     atag="<b>",etag="</b>",kla="{",kle="}")
}
```

Nur für die Lesbarkeit der Html-Dateien werden noch ein paar Zeilenumbrüche eingefügt.

93 ⟨*integriere Newline hinter Zeilenumbrüchen* 93⟩≡
```
input<-gsub("<br>","<br>\n",input)
```

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

94 ⟨*schreibe Ergebnis in Datei* `weaveRhtml` 94⟩≡
```
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.

95 ⟨*stelle Typ der Zeilen fest* `weaveRhtml` 95⟩≡
```
⟨checke Leer-, Textzeilen weaveRhtml 96⟩
⟨behandle verbatim-Zeilen weaveRhtml 97⟩
⟨checke Header- und Use-Zeilen weaveRhtml 98⟩
⟨checke normale Code-Zeilen weaveRhtml 99⟩
⟨belege Typ-Vektor weaveRhtml 100⟩
```

96 ⟨*checke Leer-, Textzeilen* `weaveRhtml` 96⟩≡
```
empty.index<-grep(pat.leerzeile,input)
text.start.index<-which("@"==substring(input,1,1))
```

statt dem `code`- wird nun das `pre`-Tag probiert. br hinter /pre?? und input[verb.index]¡-paste(input[verb.index],"¡br¿")

97 ⟨*behandle verbatim-Zeilen* `weaveRhtml` 97⟩≡
```
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]<-"<pre><FONT COLOR=\"#0000FF\">"
  input[a==-1]<-"</font></pre>"
  a<-cumsum(a)
}
verb.index<-which(a>0)
##input[verb.index]<-paste(input[verb.index],"<br>") # not used because of pre
```

98 ⟨*checke Header- und Use-Zeilen* `weaveRhtml` 98⟩≡
```
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))]
```

99 ⟨*checke normale Code-Zeilen* `weaveRhtml` 99⟩≡
```
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=FALSE]
b<-a[a[,2]!=c(-1,a[-length(a[,1]),2]),,drop=FALSE]
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))]
```

Auf `is.code.line` legen wir ab, ob eine Zeile zu einem Code-Chunk gehört.

100 ⟨*belege Typ-Vektor* `weaveRhtml` 100⟩≡
```
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"

is.code.line<-text.start.indicator<-rep(0,length.input)
text.start.indicator[1]<-1; text.start.indicator[text.start.index]<-1
text.start.indicator<-cumsum(text.start.indicator)
is.code.line[code.start.index]<-0-text.start.indicator[code.start.index]
is.code.line<-cummin(is.code.line)
is.code.line<-(text.start.indicator+is.code.line) < 1
is.code.line[code.start.index]<-FALSE
```

101 ⟨*erstelle Output* `weaveRhtml` 101⟩≡
⟨*zentriere und quote* `weaveRhtml` 91⟩
⟨*erledige Text-Chunk-Starts* `weaveRhtml` 102⟩
⟨*ersetze Befehl zur Bildeinbindung* 103⟩
⟨*extrahiere Header-, Code- und Verwendungszeilen* `weaveRhtml` 104⟩
⟨*schreibe Header-Zeilen* `weaveRhtml` 105⟩
⟨*schreibe Code-Verwendungszeilen* `weaveRhtml` 106⟩
⟨*schreibe Code-Zeilen* `weaveRhtml` 108⟩
⟨*setze Code in Text-, Header- und Verwendungszeilen* `weaveRhtml` 109⟩

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

102 ⟨*erledige Text-Chunk-Starts* `weaveRhtml` 102⟩≡
```
input[text.start.index]<-"<p>"    # vorher: @
lz<-grep("^[ ]*$",input)
if(0<length(lz)) input[lz]<-"<br>"
```

Die Höhe 12cm entspricht auch einem 18 Zoll Terminal `height=420`. Ist eine Bildhöhe von xcm erwünscht müssen wir rechnen: `x*420/12=x*35`. Die Größenanpassung ist jedoch noch in Vorbereitung und zur Zeit auskommentiert.

103 ⟨*ersetze Befehl zur Bildeinbindung* 103⟩≡

```
plz.ind<-grep("\\\\includegraphics",input)
if(0<length(plz.ind)){
  plz<-input[plz.ind]
  h<-unlist(get.argument("includegraphics",plz))
  #hh<-unlist(get.argument("includegraphics",plz,
  #                    default="height=xxcm",kla="[",kle="]"))
  #l.unit<-match(sub("^.*[0-9]([a-z][a-z]).*$","\\1",hh),c("cm","mm","in"))
  #l.unit<-ifelse(is.na(l.unit),1,l.unit)
  #l.unit<-c(1,.1,2.54)[ifelse(is.na(l.unit),1,l.unit)]
  #hh<-sub("^.*[=]([0-9.]+).*$","\\1",hh)
  #hh<-floor(as.numeric(hh)*35*l.unit)
  #hh<-ifelse(is.na(hh),"",paste(" height=",hh,sep=""))
  #h<-paste("<img SRC=\"",sub(".ps$",".jpg",h),"\"",hh,">",sep="")
  h<-paste("<img SRC=\"",sub(".ps$",".jpg",h),"\">",sep="")
  input[plz.ind]<-h
}
```

104 ⟨*extrahiere Header-, Code- und Verwendungszeilen* `weaveRhtml` 104⟩≡

```
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])
```

105 ⟨*schreibe Header-Zeilen* `weaveRhtml` 105⟩≡

```
no<-seq(along=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>",sep="")
input[code.start.index]<-code.start.lines
```

106 ⟨*schreibe Code-Verwendungszeilen* `weaveRhtml` 106⟩≡

```
use.lines<-input[use.index]; is.use.lines.within.code<-is.code.line[use.index]
leerzeichen.vor.use<-sub("[^ ](.*)$","",use.lines)
use.lines<-substring(use.lines,nchar(leerzeichen.vor.use))
h<-gsub("\\ "," ",leerzeichen.vor.use)
leerzeichen.vor.use<-ifelse(is.use.lines.within.code,h,leerzeichen.vor.use)
for(i in seq(along=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)
  if(is.use.lines.within.code[i]){
    uli[cand]<-paste("</code>&lt;",uli[cand]," ",ref.no,"&gt;<code>",sep="")
  }else{
    uli[cand]<-paste(" &lt;",uli[cand]," ",ref.no,"&gt; ",sep="")
  }
 #  formating code within use references weaveRhtml
  if(length(uli)!=length(cand)){
    if(!UTF){
      uli[-cand]<-paste("",uli[-cand],"",sep="") #050612
    }else{
      uli[-cand]<-paste("",uli[-cand],"",sep="") #060516
    }
  }
  if(is.use.lines.within.code[i]){
    use.lines[i]<-paste("<code>",paste(uli,collapse=""),"</code>")
  }else{
    use.lines[i]<-paste(" ",paste(uli,collapse=""),"  ")
  }
}
input[use.index]<-paste("<br>",leerzeichen.vor.use,use.lines)
```

107 ⟨*ddd* 107⟩≡

```
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.

108 ⟨*schreibe Code-Zeilen* `weaveRhtml` 108⟩≡

```
leerzeichen.vor.c<-gsub("\t","      ",code.lines)
leerzeichen.vor.c<-sub("[^ ](.*)$","",leerzeichen.vor.c)
leerzeichen.vor.c<-gsub("\\ "," ",leerzeichen.vor.c)
# special case "<letter" has to be handled
code.lines<-gsub("<(.)","&lt;\\1",code.lines)
code.lines<-gsub("\\ "," ",code.lines) ## multiple blanks in code lines
if(!UTF){
  input[code.index]<-paste("<br>",leerzeichen.vor.c,"<code>",code.lines,"</code>")
}else{
  input[code.index]<-paste("<br>",leerzeichen.vor.c,"<code>",code.lines,"</code>")
}
```

109 ⟨*setze Code in Text-, Header- und Verwendungszeilen* `weaveRhtml` 109⟩≡

```
typ<-"TEXT"
⟨setze Code in Zeilen vom Typ typ weaveRhtml 110⟩
typ<-"HEADER"
⟨setze Code in Zeilen vom Typ typ weaveRhtml 110⟩
typ<-"USE"
⟨setze Code in Zeilen vom Typ typ weaveRhtml 110⟩
```

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.

110 ⟨*setze Code in Zeilen vom Typ* `typ` `weaveRhtml` 110⟩≡

```
⟨suche in Zeilen des Typs nach Code im Text code.im.text.index weaveRhtml 111⟩
if(0<length(code.im.text.index)){
  lines.to.check<-input[code.im.text.index]
  ⟨ersetze zusammenhängende Wortstücke weaveRhtml 112⟩
  ⟨checke und ersetze Code im Text mit Leerzeichen weaveRhtml 114⟩
  input[code.im.text.index]<-lines.to.check
}
```

111 ⟨*suche in Zeilen des Typs nach Code im Text* `code.im.text.index` `weaveRhtml` 111⟩≡

```
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.

112 ⟨*ersetze zusammenhängende Wortstücke* `weaveRhtml` 112⟩≡

```
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(..))
```

113 ⟨*old* 25⟩+≡

```
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.

114 ⟨*checke und ersetze Code im Text mit Leerzeichen* `weaveRhtml` 114⟩≡

```
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.

115 ⟨*lege bin-Datei* `weaveRhtml` *an* 115⟩≡

```
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`.

116 ⟨*teste Funktion* `weaveRhtml` 116⟩≡

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

117 ⟨*t* 117⟩≡

```
⟨teste Funktion weaveRhtml 116⟩
```

4.2 Help-Page

118 ⟨*define-weaveRhtml-help* 118⟩≡

```
\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,replace.umlaute=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 this argument is missing the extension of the
   input file is turned to \code{.html} }
  \item{replace.umlaute}{ if TRUE german umlaute will be replaced by
   html sequences}
}
\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.

  Rweb speciality: A code chunk with a code chunk name containing  \code{"Rweb"}
  as the first four characters will be translated to a text input field and
  a submit button \code{compute via Rweb}.
  By pressing this button the code of the text field will be
  sent for evaluation to Rweb \code{http://rweb.stat.umn.edu/Rweb/}
  and the results appears in a new browser window.
  This link to Rweb has been inspirited by web pages like
  \url{http://www.stat.umn.edu/geyer/3011/examp/reg.html} written
  by Charlie Geyer \url{http://www.stat.umn.edu/~charlie}.

  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}
```