The *rmoparser* : Parser combinator in R

Juan Gea Rosat, Ramon Martínez Coscollà

November 21, 2011

Contents

Chapter 1

Introduction

1.1 Package functionality

The aim of this package is to provide a collection of functions which allow easily programming of parsers, for not so common data files formats. To achieve this objective it is enough to have a very basic collection of functions. As the first practical applications, recognition of PC-AXIS format is implemented.

Theoretical references about grammars, parsers, parser combinator can be found in:

- Context-free grammar.

  `http://en.wikipedia.org/wiki/Context-free_grammar`

- Backus–Naur Form.

  `http://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form`

- Extended Backus–Naur Form

  `http://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form`

- Parser combinator

  `http://en.wikipedia.org/wiki/Parser_combinator`

- Parsing

  `http://en.wikipedia.org/wiki/List_of_algorithms#Parsing`

and references about PC-AXIS:

- PC-Axis file format

  `http://www.scb.se/Pages/List____314011.aspx`

- PC-Axis file format manual. Statistics of Finland.
  `http://tilastokeskus.fi/tup/pcaxis/tiedostomuoto2006_laaja_en.pdf`

1.2 Package descriptive files in R

1.2.1 Field parametrization

Package name

It is convenient to use a system and language portable name.

⟨*PACKAGE*⟩≡

qmrparser

Name of file containing this document

Extension cannot be tex since a latex file is generated with this name

⟨*NAMEPKGLP*⟩≡

< < PACKAGE > >.nw

Package title

⟨*TITLE*⟩≡

Parser combinator in R

Description

⟨*DESCRIPTIONFIELD*⟩≡

Basic functions for building parsers, with an application to PC−AXIS format files.

Author

⟨*AUTHOR*⟩≡

Juan Gea Rosat, Ramon Martínez Coscollà

Version

⟨*VERSION*⟩≡

0.1.1

Date

⟨*DATE*⟩≡

2011−11−21

License

⟨*LICENSE*⟩≡

GPL (>= 3)

1.2.2 DESCRIPTION

⟨*DESCRIPTION*⟩≡

Package: < < PACKAGE > >
Type: Package
Title: < < TITLE > >
Version: < < VERSION > >
Date: < < DATE > >
Author: < < AUTHOR > >.
Maintainer: <juangea@geax.net>
Description: < < DESCRIPTIONFIELD > >
License: < < LICENSE > >
Depends: R (>= 2.11.1)
Suggests: RUnit
LazyLoad: yes
Encoding: UTF−8

1.2.3 qmrparser-package

⟨*qmrparser-package.Rd*⟩≡

\name{< < PACKAGE > >−package}
\alias{< < PACKAGE > >−package}
\alias{< < PACKAGE > >}
\docType{package}
title{
< < TITLE > >
}
\description{
< < DESCRIPTIONFIELD > >
}
\details{
tabular{ll}{
Package: \tab < < PACKAGE > >**cr**
Type: \tab Package**cr**
Version: \tab < < VERSION > >**cr**
Date: \tab < < DATE > >**cr**
License: \tab < < LICENSE > >**cr**
LazyLoad:\tab yes**cr**
}
Collection of functions to build programs to read complex data files
  formats, with an application to the case of PC−AXIS format.
}
author{
 < < AUTHOR > >

Maintainer: Juan Gea Rosat <juangea@geax.net>
}
\references{

Parser combinator.
\url{http://en.wikipedia.org/wiki/Parser_combinator}

Context−free grammar.
\url{http://en.wikipedia.org/wiki/Context−free_grammar}

PC−Axis file format.
\url{http://www.scb.se/Pages/List_____314011.aspx}

Type \code{RShowDoc("index",package="< < PACKAGE > >")} at
  the R command line to open the package vignette.

Type \code{ RShowDoc("< < PACKAGE > >",package="< <
  PACKAGE > >")} to open pdf developer guide.

Source code used in literate programming can be found in folder '
  noweb'.
}
\keyword{ package }
\keyword{ parser combinator }
\keyword{ token }
\keyword{ PC−AXIS }

1.2.4 NAMESPACE

⟨*NAMESPACE*⟩≡

```
exportPattern("^[[:alpha:]]+")
```

Chapter 2

Parser combinator

2.1 StreamParser

2.1.1 Motivation

While syntactically analysing, text characters are processed sequentially. In each point of the process, there may be several applicable rules and, for this reason, the chosen and applied rule may turn out to be incorrect after one or some steps. In this case, analysis cannot go forward and parser needs to go backwards in text. Therefore, an interface which allows this translation throughout the text is required.

2.1.2 Generic interface

Generic interface defining four functions which allows character processing.

They are implemented as list elements, although S3 or S4 could have been used.

From now on, this interface is called streamParser.

⟨*streamParser.Rd*⟩≡

\name{streamParser}
\alias{streamParserNextChar}
\alias{streamParserNextChar}
\alias{streamParserNextCharSeq}
\alias{streamParserPosition}
\alias{streamParserClose}
title{
  Generic interface for character processing, allowing forward and backwards translation.
}
\description{
  Generic interface for character processing. It allows going forward sequentially or backwards to a previous arbitrary position.

  Each one of these functions performs an operation on or obtains information from a character sequence (stream).
}
\usage{
streamParserNextChar(stream)
streamParserNextCharSeq(stream)
streamParserPosition(stream)
streamParserClose(stream)
}
\arguments{
item{stream}{object containing information about the text to be processed and, specifically, about the next character to be read}
}

\details{
  itemize{
    item{streamParserNextChar}{

      Reads next character, checking if position to be read is correct.
    }

    item{streamParserNextCharSeq}{

      Reads next character, without checking if position to be read is correct. Implemented since it is faster than streamParserNextChar}

    item{streamParserPosition}{

      Returns information about text position being read.}

```
    \item{streamParserClose}{

      Closes the stream}
  }
}

\value{
\item{streamParserNextChar and streamParserNextCharSeq}{
    Three field list:
    \itemize{
    \item{status}{

      "ok" or "eof"}
    \item{char}{

      Character read (ok) or "" (eof)}
    \item{stream}{

      With information about next character to be read or same
  position if end of file has been reached ("eof")}
    }
}

\item{streamParserPosition}{
  Three field list:
    \itemize{
      \item{fileName}{
        File name or "" if the stream is not associated with a file name
  }
      \item{line}{

        line number}
      \item{linePos}{

        character to be read position within its line}

      \item{streamPos}{

        character to be read position from the text beginning}
    }
}

\item{streamParserClose}{NULL}
}
```

```
\seealso{
  \code{\link{streamParserFromFileName}}
  \code{\link{streamParserFromString}}
}

\examples{

stream<- streamParserFromString("Hello world")

cstream <- streamParserNextChar(stream)

while( cstream$status == "ok" ) {
    print(streamParserPosition(cstream$stream))
    print(cstream$char)
    cstream <- streamParserNextCharSeq(cstream$stream)
}

streamParserClose(stream)

}

\keyword{streamParser}
```

⟨*streamParserNextChar.R*⟩≡

```
streamParserNextChar <- function(stream) stream$
  streamParserNextChar(stream)
```

⟨*streamParserNextCharSeq.R*⟩≡

```
streamParserNextCharSeq <- function(stream) stream$
  streamParserNextCharSeq(stream)
```

⟨*streamParserPosition.R*⟩≡

```
streamParserPosition <- function(stream) stream$
  streamParserPosition(stream)
```

⟨*streamParserClose.R*⟩≡

```
streamParserClose <- function(stream) stream$streamParserClose(
  stream)
```

2.1.3 StreamParser from file name

Implements a streamParser from a file name

⟨*streamParserFromFileName.Rd*⟩≡

\name{streamParserFromFileName}
\alias{streamParserFromFileName}
title{
  Creates a streamParser from a file name
}
\description{
  Creates a list of functions which allow streamParser manipulation (
  when defined from a file name)
}
\usage{
streamParserFromFileName(fileName,encoding = getOption("encoding
  "))
}
\arguments{
item{fileName}{file name}
item{encoding}{file encoding}
}
\details{

    See \link[< < PACKAGE > >]{streamParser}

    This function implementation uses function \link{seek}.

    Documentation about this function states:

      "
      Use of 'seek' on Windows is discouraged. We have found so many
      errors in the Windows implementation of file positioning that
      users are advised to use it only at their own risk, and asked not
      to waste the R developers' time with bug reports on Windows'
      deficiencies.
    "

    If "fileName" is a url, \link{seek} is not possible.

  In order to cover these situations, streamPaserFromFileName
  functions are converted in:

      \code{
        \link{streamParserFromString}(\link{readLines}(fileName,
  encoding=encoding))
      }

    Alternatively, it can be used:

\code{\link{streamParserFromString}} with:
\code{
\link{streamParserFromString}(\link{readLines}(fileName))
}

or

\code{
\link{streamParserFromString}(
\link{iconv}(\link{readLines}(fileName),
encodingOrigen,encodingDestino)
)
}

Since streamParserFromFileName also uses \code{\link{readChar
}}, this last option is the one advised in Linux if encoding is different
from Latin−1 or UTF−8. As documentation states, \code{\link{
readChar}} may generate problems if file is in a multi−byte non
UTF−8 encoding:

"
'nchars' will be interpreted in bytes not characters in a
non−UTF−8 multi−byte locale, with a warning.
"
}
value{
A list of four functions which allow stream manipulation:

item{streamParserNextChar}{Function which takes a streamParser
as argument and returns a \code{list(status,char,stream)}}

item{streamParserNextCharSeq}{Function which takes a
streamParser as argument and returns \code{list(status,char,stream)
}}

item{streamParserPosition}{Function which takes a streamParser as
argument and returns position of next character to be read}

item{streamParserClose}{Closes the stream}
}

\examples{

```
  name <- system.file("extdata","datInTest01.txt", package = "< <
  PACKAGE > >")

  stream <- streamParserFromFileName(name)

  cstream <- streamParserNextChar(stream)

  while( cstream$status == "ok" ) {
    print(streamParserPosition(cstream$stream))
    print(cstream$char)
    cstream <- streamParserNextCharSeq(cstream$stream)
  }

  streamParserClose(stream)

}
```

\keyword{streamParser}

Direct calls to '.Internal(seek(' in this function are motivated by performance optimization.

⟨*streamParserFromFileName.R*⟩≡

```
streamParserFromFileName <- function(fileName,encoding =
  getOption("encoding")) {

  ## streamParseFromFileName can be used?
  if( Sys.info()["sysname"] == "Windows" ) {

    fromString <- TRUE

  } else {
    conn <- file(fileName,"r",encoding =encoding)
    if ( ! isOpen(conn) ) stop(paste("Error: file cannot be opened",
  fileName))
    fromString <- tryCatch({ seek(conn) ; FALSE}, error =function
  (e) TRUE, finally= close(conn) )

  }

  if ( fromString ) return( streamParserFromString( readLines(
  fileName, encoding=encoding)) )
  else
    return( list(
                  streamParserNextChar = function(stream) {

                    ## if ( stream$pos != seek(stream$conn) ) seek(
  stream$conn,stream$pos)
                    if ( stream$pos != .Internal(seek(stream$conn, as
  .double(NA), 1, 1)) ) .Internal(seek(stream$conn, as.double(
  stream$pos), 1, 1))

                    char <- readChar(stream$conn,nchars=1,
  useBytes = FALSE)

                    if (length(char) == 0)
                      list(status="eof",char="" ,stream=stream)
                    else {

                      ## stream$pos <- seek(stream$conn)
                      stream$pos <- .Internal(seek(stream$conn, as.
  double(NA), 1, 1))

                      if ( char == "\n" ) {
                        stream$line <- stream$line + 1
                        stream$linePos <- 0
                      } else {
```

```
      stream$linePos <- stream$linePos + 1
    }

    list(status="ok" ,char=char,stream=stream)
  }
},

streamParserNextCharSeq = function(stream) {
  char <- readChar(stream$conn,nchars=1,
useBytes = FALSE)
  if (length(char) == 0)
    list(status="eof",char="" ,stream=stream)

  else {

    ## stream$pos <- seek(stream$conn)
    stream$pos <- .Internal(seek(stream$conn, as.
double(NA), 1, 1))


    if ( char == "\n" ) {
      stream$line <- stream$line + 1
      stream$linePos <- 0
    } else {
      stream$linePos <- stream$linePos + 1
    }

    list(status="ok" ,char=char,stream=stream)
  }
},

streamParserClose = function(stream) { close(
stream$conn) ; stream$conn <- -1 ; invisible(NULL)
                                                    },

streamParserPosition = function(stream) { list(
fileName=stream$fileName, line=stream$line, linePos=stream$
linePos+1, streamPos=stream$pos+1)
                                                    },
conn = local({
  conn <- file(fileName,"r",encoding =encoding)
  if ( ! isOpen(conn) ) stop(paste("Error: file cannot
 be opened.",fileName))
```

```
                    tryCatch( seek(conn) , error =function(e) stop(
  paste("Error: 'seek' not enabled for this connection", fileName)))
                    conn
                  }),
                  pos = 0,
                  line = 1,
                  linePos = 0,
                  fileName = fileName
                  )
            )
}
```

2.1.4 Stream parser from character string

Implements a streamParser from a character string.

⟨*streamParserFromString.Rd*⟩≡

\name{streamParserFromString}
\alias{streamParserFromString}
title{
  Creates a streamParser from a string
}
\description{
  Creates a list of functions which allow streamParser manipulation (
  when defined from a character string)
}
\usage{
streamParserFromString(string)
}
\arguments{
item{string}{string to be recognised}
}
\details{

    See \link[< < PACKAGE > >]{streamParser}
  }

value{

  A list of four functions which allow stream manipulation:

item{streamParserNextChar}{Functions which takes a streamParser
  as argument ant returns a \code{list(status,char,stream)}}

item{streamParserNextCharSeq}{Function which takes a
  streamParser as argument and returns a \code{list(status,char,
  stream)}}

item{streamParserPosition}{Function which takes a streamParser as
  argument and returns position of next character to be read}

item{streamParserClose}{Function which closes the stream}
}

\examples{
# reads one character
streamParserNextChar(streamParserFromString("\U00B6"))

# reads a string
stream <- streamParserFromString("Hello world")

```
cstream <- streamParserNextChar(stream)

while( cstream$status == "ok" ) {
    print(streamParserPosition(cstream$stream))
    print(cstream$char)
    cstream <- streamParserNextCharSeq(cstream$stream)

streamParserClose(stream)
}

}

\keyword{streamParser}
```

⟨*streamParserFromString.R*⟩≡

```
streamParserFromString <- function(string) {

  stringIn <- paste(string,collapse="\n")
  nchars <- nchar(stringIn)

  ## One at a time character processing multiplies space by 8 (in 64
  bits OS) but reduces time /40
  stringIn <- strsplit(stringIn,split=character(0))[[1]]
  string <- stringIn
  if( nchars > 0 && length(string) < 1 ) stop("Error: encoding error
  ")

  return(
        list(
              streamParserNextChar = function(stream) {
                if ( stream$pos + 1 > stream$lenchar ) list(status="
  eof",char="" ,stream=stream)
                else {

                  stream$pos <- stream$pos+1

## version for substr char <- substr(string, stream$pos, stream$pos)
                  char <- string[stream$pos]

                  if ( char == "\n" ) {
                    stream$line <- stream$line + 1
                    stream$linePos <- 0
                  } else {
                    stream$linePos <- stream$linePos + 1
                  }

                  list(status="ok" ,char=char,stream=stream)
                }
              },

              streamParserNextCharSeq = function(stream) {
                if ( stream$pos + 1 > stream$lenchar ) list(status="
  eof",char="" ,stream=stream)
                else {

                  stream$pos <- stream$pos+1

## version for substr char <- substr(string,stream$pos, stream$pos)
                  char <- string[stream$pos]
```

```
                    if ( char == "\n" ) {
                      stream$line <- stream$line + 1
                      stream$linePos <- 0
                    } else {
                      stream$linePos <- stream$linePos + 1
                    }

                    list(status="ok" ,char=char,stream=stream)
                  }
                },

                streamParserClose = function(stream) { lenchar <- 0 ;
  invisible(NULL)
                                                       },
                streamParserPosition = function(stream) { list(
  fileName="", line=stream$line, linePos=stream$linePos+1,
  streamPos=stream$pos+1 )
                                                       },

## lenchar = nchar(stringIn) , ## version for substr
                lenchar = length(stringIn),
                pos = 0,
                line = 1,
                linePos = 0
                )
        )
}
```

2.2 Character sets

The first step in syntactical analysis is breaking the text into words. Word separation is determined by the definition of characters separating words and the collection of characters that can make up a valid word. This schema requires, therefore, a character classification, according to the role played in word separation or word formation.

This section includes predicates (functions returning TRUE or FALSE) to determine if a character belongs to a set of characters playing a specific role.

2.2.1 isWhitespace

⟨*isWhitespace.Rd*⟩≡

\name{isWhitespace}
\alias{isWhitespace}
title{
Is it a white space?
}
\description{
Checks whether a character belongs to the set \{blank, tabulator, new line, carriage return, page break \}.
}
\usage{
isWhitespace(ch)
}
\arguments{
item{ch}{character to be checked}
}
value{
TRUE/FALSE, depending on character belonging to the specified set.
}
\examples{
isWhitespace(' ')
isWhitespace('\n')
isWhitespace('a')
}

\keyword{set of character}

this version is slower

```
isWhitespace <- function(ch) ch %in% c(' ' , '\t' , '\f' , '\r' , '\n' )
```

than

⟨*isWhitespace.R*⟩≡

```
isWhitespace <- function(ch) switch(ch,' '= , '\t'= , '\f'= , '\r'= , '\n'=TRUE,FALSE)
```

2.2.2 isNewline

⟨*isNewline.Rd*⟩≡

\name{isNewline}
\alias{isNewline}
title{
Is it a new line character?
}
\description{
Checks whether a character is a new line character.
}
\usage{
isNewline(ch)
}
\arguments{
item{ch}{character to be checked}
}
value{
TRUE/FALSE, depending on character being a newline character
}
\examples{
isNewline(' ')
isNewline('\n')
}

\keyword{set of character}

⟨*isNewline.R*⟩≡

isNewline <− **function**(ch) ch == '\n'

2.2.3 isDigit

⟨*isDigit.Rd*⟩≡

\name{isDigit}
\alias{isDigit}
title{
Is it a digit?
}
\description{
Checks whether a character is a digit: \{ 0 .. 9 \}.
}
\usage{
isDigit(ch)
}
\arguments{
item{ch}{character to be checked}
}
value{
TRUE/FALSE, depending on the character being a digit.
}
\examples{
isDigit('9')
isDigit('a')
}

\keyword{set of character}

This version

```
isDigit <- function(ch) ch %in% c('0' , '1' , '2' , '3' , '4' , '5' , '6' , '7
  ' , '8' , '9')
```

is slower than

⟨*isDigit.R*⟩≡

```
isDigit <- function(ch) switch(ch,'0'= , '1'= , '2'= , '3'= , '4'= , '5'
  = , '6'= , '7'= , '8'= , '9'=TRUE,FALSE)
```

2.2.4 isUppercase

⟨*isUppercase.Rd*⟩≡

\name{isUppercase}
\alias{isUppercase}
title{
Is it an upper case?
}
\description{
Checks whether a character is an upper case.

Restricted to ASCII character (does not process ñ, ç, accented vowels
  ...)
}
\usage{
isUppercase(ch)
}
\arguments{
item{ch}{character to be checked}
}
value{
TRUE/FALSE, depending on character being an upper case character.
}
\examples{
isUppercase('A')
isUppercase('a')
isUppercase('9')
}

\keyword{set of character}

⟨*isUppercase.R*⟩≡

isUppercase <− **function**(ch) 'A' <= ch **&&** ch <= 'Z'

2.2.5 isLowercase

⟨*isLowercase.Rd*⟩≡

\name{isLowercase}
\alias{isLowercase}
title{
Is it a lower case?
}
\description{
Checks whether a character is a lower case.

Restricted to ASCII character (does not process ñ, ç, accented vowels
  ...)
}
\usage{
isLowercase(ch)
}
\arguments{
item{ch}{character to be checked}
}
value{
TRUE/FALSE, depending on character being a lower case character.
}
\examples{
isLowercase('A')
isLowercase('a')
isLowercase('9')
}

\keyword{set of character}

⟨*isLowercase.R*⟩≡

isLowercase <− **function**(ch) 'a' <= ch **&&** ch <= 'z'

2.2.6 isLetter

⟨*isLetter.Rd*⟩≡

\name{isLetter}
\alias{isLetter}
title{
Is it a letter?
}
\description{
Checks whether a character is a letter

Restricted to ASCII character (does not process ñ, ç, accented vowels
  ...)
}
\usage{
isLetter(ch)
}
\arguments{
item{ch}{character to be checked}
}
value{
TRUE/FALSE, depending on the character being a letter.
}
\examples{
isLetter('A')
isLetter('a')
isLetter('9')
}

\keyword{set of character}

⟨*isLetter.R*⟩≡

isLetter <− **function**(ch) isUppercase(ch) || isLowercase(ch)

2.2.7 isSymbol

⟨*isSymbol.Rd*⟩≡

\name{isSymbol}
\alias{isSymbol}
title{
Is it a symbol?
}
\description{
Checks whether a character is a symbol, a special character.
}
\usage{
isSymbol(ch)
}
\arguments{
item{ch}{character to be checked}
}
\details{
These characters are considered as symbols:

'**!**' , '\%' , '**&**' , '**$**' , '#' , '+' , '−' , '/' , ':' , '<' , '=' , '>' , '?' , '@' , '\|' , '~' , '^' , '|' , '*****'

}
value{
TRUE/FALSE, depending on character being a symbol.
}
\examples{
isSymbol('+')
isSymbol('A')
isSymbol('a')
isSymbol('9')
}

\keyword{set of character}

⟨*isSymbol.R*⟩≡

isSymbol <− **function**(ch) ch %in% **c**('**!**' , '%' , '**&**' , '**$**' , '#' , '+' , '−' , '/' , ':' , '<' , '=' , '>' , '?' , '@' , '\\' , '~' , '^' , '|' , '*****')

2.2.8 isHex

⟨*isHex.Rd*⟩≡

\name{isHex}
\alias{isHex}
title{
Is it an hexadecimal digit?
}
\description{
Checks whether a character is an hexadecimal digit.
}
\usage{
isHex(ch)
}
\arguments{
item{ch}{character to be checked}
}
value{
TRUE/FALSE, depending on character being an hexadecimal digit.
}
\examples{
isHex('+')
isHex('A')
isHex('a')
isHex('9')
}

\keyword{set of character}

⟨*isHex.R*⟩≡

```
isHex <- function(ch) ( '0' <= ch && ch <= '9' ) || ('a' <= ch &&
    ch <= 'f') || ('A' <= ch && ch <= 'F')
```

2.3 Tokens/Words

Word (token) recognition is a basic action in syntactic analysis.

Functions to break text into words. The list of valid words depend on analysed language. A typical set of functions for word/token recognition is included.

These functions share some homogeneity, since they must return the same data type in order to be able to combine them to create more complex parsers.

- Functions share, at least, these input parameters:
  ⟨*TokenArguments*⟩≡

item{action}{Function to be executed if recognition succeeds. Character stream making up the token is passed as parameter to this function}

item{error}{Function to be executed if recognition does not succeed. Position of \code{\link[< < PACKAGE > >]{ streamParser}} obtained with \code{\link{ streamParserPosition}} is passed as parameter to this function}

- These functions always have the next value:
  ⟨ *TokenValue*⟩≡

  Anonymous function, returning a list.

  \code{function(stream)} −−> \code{ list(status,node, stream) }

  From input parameters, an anonymous function is defined. This function admits just one parameter, stream, with type \code{\ link[< < PACKAGE > >]{streamParser}}, and returns a three −field list:
  itemize{
  item{status}{

  "ok" or "fail"}

  item{node}{

  With \code{action} or \code{error} function output, depending on the case}

  item{stream}{

  With information about the input, after success or failure in recognition}

  }

2.3.1 eofMark

⟨ *eofMark.Rd*⟩≡

```
\name{eofMark}
\alias{eofMark}
\title{
        End of file token
}
\description{
Recognises the end of input flux as a token.

When applied, it does not make use of character and, therefore,
  end of input can be recognised several times.
}
\usage{
  eofMark(action = function(s) list(type="eofMark",value=s),
          error = function(p) list(type="eofMark",pos =p ) )
}
\arguments{
< < TokenArguments > >
}
\details{
  When succeeds, parameter \code{s} takes the value "".
}
\value{
  < < TokenValue > >
}
\examples{

# fail
stream <- streamParserFromString("Hello world")
( eofMark()(stream) )[c("status","node")]

# ok
stream <- streamParserFromString("")
( eofMark()(stream) )[c("status","node")]

}

\keyword{token}
```

⟨*eofMark.R*⟩≡

```
eofMark <- function(action = function(s) list(type="
  eofMark",value=s),
                      error = function(p) list(type="eofMark",
  pos=p ))

  function (stream) {

    cstream <- streamParserNextChar(stream)

    if ( cstream$status != "eof" ) return(list(status="fail",
  node=error(streamParserPosition(stream)), stream=stream))

    return(list(status="ok",node=action(""),stream=cstream$
  stream))
  }
```

Test for RUnit

⟨*testTokens01*⟩≡

```
checkTokenParserOk ("",eofMark(),"eofMark")
```

2.3.2 whitespace

⟨*whitespace.Rd*⟩≡

```
\name{whitespace}
\alias{whitespace}
\title{
        White sequence token.
}
\description{
  Recognises a white character sequence (this sequence may be
  empty).

}
\usage{
  whitespace(action = function(s) list(type="white",value=s),
             error = function(p) list(type="white",pos =p) )
}
\arguments{
< < TokenArguments > >
}
\details{
  A character is considered a white character when function \
  code{\link{isWhitespace}} returns TRUE
}
\value{
  < < TokenValue > >
}
\examples{

# ok
stream <- streamParserFromString("Hello world")
( whitespace()(stream) )[c("status","node")]

# ok
stream <- streamParserFromString(" Hello world")
( whitespace()(stream) )[c("status","node")]

# ok
stream <- streamParserFromString("")
( whitespace()(stream) )[c("status","node")]

}

\keyword{token}
```

⟨*whitespace.R*⟩≡

```
whitespace <- function(action = function(s) list(type="
  white",value=s),
                       error = function(p) list(type="white",
  pos =p) )

  function (stream) {

    cstream <- streamParserNextChar(stream)

    if ( cstream$status == "eof" || ! isWhitespace(cstream$
  char) ) return(list(status="ok",node=action(""),stream=
  stream))

    s <- cstream$char
    repeat {
      stream <- cstream$stream
      cstream <- streamParserNextCharSeq(stream)

      if ( cstream$status == "eof" || ! isWhitespace(cstream$
  char) ) return(list(status="ok",node=action(paste(s,
  collapse="")),stream=stream))

      s <- c(s,cstream$char)
    }
  }
```

Test for RUnit

⟨*testTokens01*⟩+≡

```
checkTokenParserOk ("" ,whitespace(),"white")
checkTokenParserOk (" ",whitespace(),"white")
checkTokenParserOk ("\n",whitespace(),"white")
checkTokenParserOk ("A" ,whitespace(),"white","")
checkTokenParserOk ("1" ,whitespace(),"white","")
```

2.3.3 separator

⟨*separator.Rd*⟩≡

```
\name{separator}
\alias{separator}
\title{
        Generic word separator token.
}
\description{
  Recognises a white character sequence, with comma or
  semicolon optionally inserted in the sequence.
  Empty sequences are not allowed.
}
\usage{
separator(action = function(s) list(type="separator",value=s) ,
          error = function(p) list(type="separator",pos =p) )
}
\arguments{
  < < TokenArguments > >
}
\details{
  A character is considered a white character when function \
  code{\link{isWhitespace}} returns TRUE
}
\value{
  < < TokenValue > >
}

\note{ PC-Axis has accepted the delimiters comma, space,
  semicolon, tabulator. }

\examples{

# ok
stream <- streamParserFromString("; Hello world")
( separator()(stream) )[c("status","node")]

# ok
stream <- streamParserFromString(" ")
( separator()(stream) )[c("status","node")]


# fail
stream <- streamParserFromString("Hello world")
( separator()(stream) )[c("status","node")]

# fail
```

stream <− streamParserFromString(””)
(separator()(stream))[c(”status”,”node”)]

}

\keyword{token}

⟨*separator.R*⟩≡

```
separator <- function(action= function(s) list(type="
  separator",value=s),
                       error = function(p) list(type="separator"
  ,pos=p ))
  function (stream) {
    cstream <- streamParserNextChar(stream)

    if ( cstream$status == "eof" ) return(list(status="fail",
  node=error(streamParserPosition(stream)),stream=stream))

    ## possible whites
    s <- ""
    repeat {
      if ( cstream$status == "eof" || ! isWhitespace(cstream$
  char) ) break()
      s <- c(s,cstream$char)
      stream <- cstream$stream
      cstream <- streamParserNextCharSeq(stream)
    }

    ##possible , ;
    if ( cstream$status == "eof" || ( cstream$char != ',' &&
  cstream$char != ';') ) {
      if ( length(s) > 1 ) return(list(status="ok",node=
  action(paste(s,collapse=""")),stream=stream))
      else
        return(list(status="fail",node=error(
  streamParserPosition(stream)),stream=stream))
    }
    ##
    s <- c(s,cstream$char)

    ## possible white
    repeat {
      stream <- cstream$stream
      cstream <- streamParserNextCharSeq(stream)

      if ( cstream$status == "eof" || ! isWhitespace(cstream$
  char) ) return(list(status="ok",node=action(paste(s,
  collapse=""")),stream=stream))

      s <- c(s,cstream$char)
    }
  }
```

Test for RUnit

⟨*testTokens01*⟩+≡

```
checkTokenParserFail ("" ,separator(),"separator")
checkTokenParserFail ("A" ,separator(),"separator")
checkTokenParserFail ("1" ,separator(),"separator")
checkTokenParserOk (" " ,separator(),"separator")
checkTokenParserOk ("\n" ,separator(),"separator")
checkTokenParserOk ("," ,separator(),"separator")
checkTokenParserOk (";" ,separator(),"separator")
checkTokenParserOk (" ;" ,separator(),"separator")
checkTokenParserOk ("; " ,separator(),"separator")
checkTokenParserOk (" ; " ,separator(),"separator")
checkTokenParserOk (" , " ,separator(),"separator")
```

2.3.4 numberNatural

⟨*numberNatural.Rd*⟩≡

```
\name{numberNatural}
\alias{numberNatural}
\title{
  Natural number token.
}
\description{
  A natural number is a sequence of digits.
}
\usage{
  numberNatural(action = function(s) list(type="numberNatural
  ",value=s),
                error = function(p) list(type="numberNatural",
  pos =p))
}
\arguments{
  < < TokenArguments > >
}
\value{
  < < TokenValue > >
}
\examples{

# fail
stream <- streamParserFromString("Hello world")
( numberNatural()(stream) )[c("status","node")]

# ok
stream <- streamParserFromString("123")
( numberNatural()(stream) )[c("status","node")]

}

\keyword{token}
```

⟨*numberNatural.R*⟩≡

```
numberNatural <- function(action = function(s) list(type="
  numberNatural",value=s),
                          error = function(p) list(type="
  numberNatural",pos =p) )

  function (stream) {
    cstream <- streamParserNextChar(stream)

    if ( cstream$status == "eof" || ! isDigit(cstream$char) )
  return(list(status="fail",node=error(streamParserPosition(
  stream)),stream=stream))

    s <- cstream$char

    repeat {

      stream <- cstream$stream
      cstream <- streamParserNextCharSeq(stream)

      if ( cstream$status == "eof" || ! isDigit(cstream$char) )
  return(list(status="ok",node=action(paste(s,collapse=""))),
  stream=stream))

      s <- c(s,cstream$char)

    }
  }
```

Test for RUnit

⟨*testTokens01*⟩+≡

```
checkTokenParserOk ("123" ,numberNatural() ,"numberNatural"
  )
```

2.3.5 numberInteger

⟨*numberInteger.Rd*⟩≡

```
\name{numberInteger}
\alias{numberInteger}
\title{
  Integer number token.
}
\description{
  Recognises an integer, i.e., a natural number optionally
  preceded by a + or − sign.

}
\usage{
  numberInteger(action = function(s) list(type="numberInteger",
  value=s),
                error = function(p) list(type="numberInteger",
  pos =p))

}
\arguments{
  < < TokenArguments > >
}
\value{
  < < TokenValue > >
}
\examples{

# fail
stream <− streamParserFromString("Hello world")
( numberInteger()(stream) )[c("status","node")]

# ok
stream <− streamParserFromString("−1234")
( numberInteger()(stream) )[c("status","node")]

}

\keyword{token}
```

⟨*numberInteger.R*⟩≡

```
numberInteger <- function(action = function(s) list(type="
  numberInteger",value=s),
                          error = function(p) list(type="
  numberInteger",pos =p))

  function (stream) {
    cstream <- streamParserNextChar(stream)

    if ( cstream$status == "eof" ) return(list(status="fail",
  node=error(streamParserPosition(stream)),stream=stream))

    if ( cstream$char %in% c("+","-") ) {
      s <- cstream$char
      cstream <- numberNatural()(cstream$stream)
    }
    else {
      s <- ""
      cstream <- numberNatural()(stream)
    }
    if ( cstream$status == "fail" ) return(list(status="fail",
  node=error(streamParserPosition(stream)),stream=stream))
    return(list(status="ok",node=action(paste(s,cstream$
  node$value,sep="")),stream=cstream$stream))

  }
```

Test for RUnit

⟨*testTokens01*⟩+≡

```
checkTokenParserOk ("123" ,numberInteger() ,"numberInteger" )
checkTokenParserOk ("+123" ,numberInteger() ,"numberInteger"
   )
checkTokenParserOk ("-123" ,numberInteger() ,"numberInteger"
   )
```

2.3.6 numberFloat

⟨*numberFloat.Rd*⟩≡

```
\name{numberFloat}
\alias{numberFloat}
\title{
  Floating−point number token.
}
\description{
  Recognises a floating−point number, i.e., an integer with a
  decimal part. One of them (either integer or decimal part) must
   be present.
}
\usage{
  numberFloat(action = function(s) list(type="numberFloat",
  value=s),
                error = function(p) list(type="numberFloat",pos =
  p))

}
\arguments{
  < < TokenArguments > >
}
\value{
  < < TokenValue > >
}
\examples{

# fail
stream <− streamParserFromString("Hello world")
( numberFloat()(stream) )[c("status","node")]

# ok
stream <− streamParserFromString("−456.74")
( numberFloat()(stream) )[c("status","node")]

}

\keyword{token}
```

⟨*numberFloat.R*⟩≡

```
numberFloat <- function(action = function(s) list(type="
  numberFloat",value=s),
                          error = function(p) list(type="
  numberFloat",pos=p) )

  function (stream) {
    streamFail <- stream
    cstream <- streamParserNextChar(stream)

    if ( cstream$status == "eof" ) return(list(status="fail",
  node=error(streamParserPosition(streamFail)),stream=
  streamFail))

    ## sign
    if ( cstream$char %in% c("+","-") ) {
      signo <- cstream$char
      stream <- cstream$stream
      cstream <- streamParserNextChar(stream)

      if ( cstream$status == "eof" ) return(list(status="fail",
  node=error(streamParserPosition(streamFail)),stream=
  streamFail))

    } else {
      signo <- ""
    }
    if ( cstream$char == '.' ) {
      entero <- ""
      punto <- cstream$char
      cstream <- numberNatural()(cstream$stream)
      if ( cstream$status == "ok" ) {
        decimal <- cstream$node$value
        stream <- cstream$stream
      } else {
        return(list(status="fail",node=error(
  streamParserPosition(streamFail)),stream=streamFail))

      }
    }
    else {
      cstream <- numberNatural()(stream)

      if (cstream$status=="fail") return(list(status="fail",
  node=error(streamParserPosition(streamFail)),stream=
```

```
  streamFail))

      entero <- cstream$node$value
      stream <- cstream$stream
      cstream <- streamParserNextChar(stream)
      if ( cstream$char == '.' ) {
        punto <- cstream$char
        stream <- cstream$stream
        cstream <- numberNatural()(stream)
        if ( cstream$status == "ok" ) {
          decimal <- cstream$node$value
          stream <- cstream$stream
        } else {
          decimal <- ""
        }
      } else {
        punto <- ""
        decimal <- ""
      }
    }
    return(list(status="ok",node=action(paste(signo,entero,
  punto,decimal,sep="")),stream=stream))

  }
```

Test for RUnit

⟨*testTokens01*⟩+≡

```
checkTokenParserOk ("123" ,numberFloat() ,"numberFloat" )
checkTokenParserOk ("+123" ,numberFloat() ,"numberFloat" )
checkTokenParserOk ("−123" ,numberFloat() ,"numberFloat" )
checkTokenParserOk ("123." ,numberFloat() ,"numberFloat" )
checkTokenParserOk ("+123." ,numberFloat() ,"numberFloat" )
checkTokenParserOk ("−123." ,numberFloat() ,"numberFloat" )
checkTokenParserOk (".123" ,numberFloat() ,"numberFloat" )
checkTokenParserOk ("+.123" ,numberFloat() ,"numberFloat" )
checkTokenParserOk ("−.123" ,numberFloat() ,"numberFloat" )
```

2.3.7 numberScientific

This function is made up of the previous function code. They are not directly called because of performance optimization.

⟨*numberScientific.Rd*⟩≡

```
\name{numberScientific}
\alias{numberScientific}
\title{
  Number in scientific notation token.
}
\description{
  Recognises a number in scientific notation, i.e., a floating−point
   number with an (optional) exponential part.
}
\usage{
  numberScientific(action = function(s) list(type="
  numberScientific",value=s),
                       error = function(p) list(type="
  numberScientific",pos=p) )

}
\arguments{
  < < TokenArguments > >
}
\value{
  < < TokenValue > >
}
\examples{

# fail
stream <− streamParserFromString("Hello world")
( numberScientific()(stream) )[c("status","node")]

# ok
stream <− streamParserFromString("−1234e12")
( numberScientific()(stream) )[c("status","node")]

}

\keyword{token}
```

⟨*numberScientific.R*⟩≡

```
numberScientific <- function(action = function(s) list(type=
  "numberScientific",value=s),
                               error = function(p) list(type="
  numberScientific",pos=p )
                               )
  function (stream) {
    streamFail <- stream

#### numberFloat
    cstream <- streamParserNextChar(stream)

    if ( cstream$status == "eof" ) return(list(status="fail",
  node=error(streamParserPosition(streamFail)),stream=
  streamFail))

    if( cstream$char == "-" || cstream$char == "+" ){
      signo <- cstream$char
      stream <- cstream$stream
      cstream <- streamParserNextChar(stream)

      if ( cstream$status == "eof" ) return(list(status="fail",
  node=error(streamParserPosition(streamFail)),stream=
  streamFail))
    } else {
      signo <- ""
    }
    if ( cstream$char == '.' ) {
      entero <- ""
      punto <- cstream$char

##### numberNatural
      cstream <- streamParserNextChar(cstream$stream)

      if ( cstream$status == "eof" || ! isDigit(cstream$char) )
  return(list(status="fail",node=error(streamParserPosition(
  streamFail)),stream=streamFail))

      s <- cstream$char
      repeat {
        stream <- cstream$stream
        cstream <- streamParserNextCharSeq(stream)
        if ( cstream$status == "eof" || ! isDigit(cstream$char)
  ) break()
        s <- c(s,cstream$char)
```

```
      }
#####
      decimal <- paste(s,collapse="")
    }
    else {
##### numberNatural
      if ( ! isDigit(cstream$char) ) return(list(status="fail",
  node=error(streamParserPosition(streamFail)),stream=
  streamFail))
      s <- cstream$char
      repeat {
        stream <- cstream$stream
        cstream <- streamParserNextCharSeq(stream)
        if ( cstream$status == "eof" || ! isDigit(cstream$char)
  ) break()
        s <- c(s,cstream$char)
      }
#####
      entero <- paste(s,collapse="")
##
      cstream <- streamParserNextChar(stream)
      if ( cstream$char == '.' ) {
        punto <- cstream$char
        stream <- cstream$stream
##### numberNatural
        cstream <- streamParserNextChar(stream)
        if ( cstream$status == "eof" || ! isDigit(cstream$char)
  ) {
          decimal <- ""
        } else {
          s <- cstream$char
          repeat {
            stream <- cstream$stream
            cstream <- streamParserNextCharSeq(stream)
            if ( cstream$status == "eof" || ! isDigit(cstream$
  char) ) break()
            s <- c(s,cstream$char)
          }
          decimal <- paste(s,collapse="")
        }
#####
      } else {
        punto <- ""
        decimal <- ""
```

```
      }
    }
    mantisa <- paste(signo,entero,punto,decimal,sep="")
#
    cstream <- streamParserNextChar(stream)
    if ( cstream$char == "E" || cstream$char == "e" ) {
      E <- cstream$char
      stream <- cstream$stream
      cstream <- streamParserNextChar(stream)

      if ( cstream$status == "eof" ) return(list(status="fail",
  node=error(streamParserPosition(streamFail)),stream=
  streamFail))
#####
      if ( cstream$char == "-" || cstream$char == "+" ) {
        signoE <- cstream$char
        stream <- cstream$stream
        cstream <- streamParserNextChar(stream)

        if ( cstream$status == "eof" ) return(list(status="fail
  ",node=error(streamParserPosition(streamFail)),stream=
  streamFail))
      } else {
        signoE <- ""
      }
##### numberNatural
      if ( ! isDigit(cstream$char) ) return(list(status="fail",
  node=error(streamParserPosition(streamFail)),stream=
  streamFail))
      s <- cstream$char
      repeat {
        stream <- cstream$stream
        cstream <- streamParserNextCharSeq(stream)
        if ( cstream$status == "eof" || ! isDigit(cstream$char)
  ) break()
        s <- c(s,cstream$char)
      }
#####
      exponente <- paste(signoE,paste(s,collapse=""),sep="")
    } else {
      E <- ""
      exponente <- ""

    }
```

return(**list**(**status**=”ok”,node=**action**(**paste**(mantisa,E,
exponente,sep=””)),stream=stream))
}

Test for RUnit
⟨*testTokens01*⟩+≡

```
checkTokenParserOk ("127" ,numberScientific(),"numberScientific
  " )
checkTokenParserOk ("+127" ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("-127" ,numberScientific()," 
  numberScientific" )

checkTokenParserOk ("127." ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("+127." ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("-127." ,numberScientific()," 
  numberScientific" )

checkTokenParserOk (".127" ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("+.127" ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("-.127" ,numberScientific()," 
  numberScientific" )

checkTokenParserOk ("0123.123" ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("+0123.123" ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("-0123.123" ,numberScientific()," 
  numberScientific" )

checkTokenParserOk ("127E33" ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("+127E33" ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("-127E33" ,numberScientific()," 
  numberScientific" )

checkTokenParserOk ("127.E33" ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("+127.E33" ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("-127.E33" ,numberScientific()," 
  numberScientific" )

checkTokenParserOk (".127E33" ,numberScientific()," 
  numberScientific" )
```

```
checkTokenParserOk ("+.127E33" ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("−.127E33" ,numberScientific()," 
  numberScientific" )

checkTokenParserOk ("0123.123E33" ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("+0123.123E33" ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("−0123.123E33" ,numberScientific()," 
  numberScientific" )

checkTokenParserOk ("−121.01e−222" ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("1211e+33" ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("−123e−222" ,numberScientific()," 
  numberScientific" )

checkTokenParserOk ("+0123.123e+11" ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("−1233.e−66" ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("−1233.e+66" ,numberScientific()," 
  numberScientific" )
checkTokenParserOk ("+.123E600" ,numberScientific()," 
  numberScientific" )

checkTokenParserFail("+.E600" ,numberScientific()," 
  numberScientific" )
checkTokenParserFail(".E600" ,numberScientific()," 
  numberScientific" )
```

2.3.8 symbolic

⟨*symbolic.Rd*⟩≡

```
\name{symbolic}
\alias{symbolic}
\title{
  Alphanumeric token.
}
\description{
  Recognises an alphanumeric symbol. By default, a sequence of
  alphanumeric, numeric and dash symbols, beginning with an
  alphabetical character.
}
\usage{
symbolic (charFirst=isLetter,
                charRest=function(ch) isLetter(ch) || isDigit(ch) ||
  ch == "-",
                action = function(s) list(type="symbolic",value=s)
  ,
                error = function(p) list(type="symbolic",pos =p))

}
\arguments{
\item{charFirst}{Predicate of valid characters as first symbol
  character}
\item{charRest}{Predicate of valid characters as the rest of
  symbol characters}
  < < TokenArguments > >
}
\value{
  < < TokenValue > >
}
\examples{

# fail
stream <- streamParserFromString("123")
( symbolic()(stream) )[c("status","node")]

# ok
stream <- streamParserFromString("abc123_2")
( symbolic()(stream) )[c("status","node")]

}

\keyword{token}
```

⟨*symbolic.R*⟩≡

```
symbolic <- function(charFirst=isLetter,
                     charRest=function(ch) isLetter(ch) ||
  isDigit(ch) || ch == "-",
                     action = function(s) list(type="symbolic"
  ,value=s),
                     error = function(p) list(type="symbolic",
  pos =p))

  function (stream) {
    cstream <- streamParserNextChar(stream)

    if ( cstream$status == "eof" || ! charFirst(cstream$char) )
  return(list(status="fail",node=error(streamParserPosition(
  stream)),stream=stream))

    s <- cstream$char
    repeat {
      stream <- cstream$stream
      cstream <- streamParserNextCharSeq(stream)

      if ( cstream$status == "eof" || ! charRest(cstream$char)
  ) return(list(status="ok",node=action(paste(s,collapse="")
  ),stream=stream))

      s <- c(s,cstream$char)
    }
  }
```

Test for RUnit

⟨*testTokens01*⟩+≡

```
checkTokenParserOk ("abd" ,symbolic(),"symbolic")
checkTokenParserOk ("ac12" ,symbolic(),"symbolic")
checkTokenParserOk ("asd:" ,symbolic(),"symbolic","asd")
```

2.3.9 string

⟨*string.Rd*⟩≡

```
\name{string}
\alias{string}
\title{
Token string
}
\description{
 Any character sequence, by default using simple or double
  quotation marks.
}
\usage{
string(isQuote= function(c) switch(c,'"'=,"'"=TRUE,FALSE),
        action = function(s) list(type="string",value=s),
        error = function(p) list(type="string",pos =p))

}
\arguments{
\item{isQuote}{Predicate indicating whether a character begins
  and ends a string}
  < < TokenArguments > >
}
\details{
  Characters preceded by \ are not considered as part of string
  end.
}
\value{
  < < TokenValue > >
}
\examples{

# fail
stream <- streamParserFromString("Hello world")
( string()(stream) )[c("status","node")]

# ok
stream <- streamParserFromString("'Hello world'")
( string()(stream) )[c("status","node")]

}

\keyword{token}
```

⟨*string.R*⟩≡

```
string <- function(isQuote= function(c) switch(c,"'"=,'"'=
  TRUE,FALSE),
                   action = function(s) list(type="string",
  value=s),
                   error = function(p) list(type="string",pos
  =p))

  function (stream) {
    cstream <- streamParserNextChar(stream)

    if ( cstream$status == "eof" || ! isQuote(cstream$char) )
  return(list(status="fail",node=error(streamParserPosition(
  stream)),stream=stream))

    delimiter <- cstream$char

    charPrev <- ""
    s <- ""
    repeat {
      cstream <- streamParserNextCharSeq(cstream$stream)

      if ( cstream$status == "eof" ) return(list(status="fail",
  node=error(streamParserPosition(stream)),stream=stream))

      if ( cstream$char == delimiter && charPrev != "\\" )
  return(list(status="ok" ,node=action(paste(s,collapse="")),
   stream=cstream$stream))

      charPrev <- cstream$char
      s <- c(s,cstream$char)
    }
  }
```

Test for RUnit

⟨*testTokens01*⟩+≡

```
checkTokenParserOk ('"aaaa"' ,string(),"string","aaaa")
checkTokenParserOk ("'aaaa'" ,string(),"string","aaaa")
checkTokenParserOk ('"aaa4567"' ,string(),"string","aaa4567")
checkTokenParserOk ("'aa aa'" ,string(),"string","aa aa")
checkTokenParserOk ("'aa\\' aa'",string(),"string","aa\\' aa")
checkTokenParserOk ("?aaaa?" ,string(isQuote=function(c) c
  =="?"),"string","aaaa")
```

2.3.10 commentParser

⟨*commentParser.Rd*⟩≡

```
\name{commentParser}
\alias{commentParser}
\title{
  Comment token.
}
\description{
  Recognises a comment, a piece of text delimited by two
  predefined tokens.
}
\usage{

  commentParser(beginComment,endComment,
          action = function(s) list(type="commentParser",value
  =s),
          error = function(p) list(type="commentParser",pos =p
  ))

}
\arguments{
  \item{beginComment}{String indicating comment beginning}
  \item{endComment}{String indicating comment end}
  < < TokenArguments > >
}
\details{
  Characters preceded by \ are not considered as part of
  beginning of comment end.
}
\value{
  < < TokenValue > >
}
\examples{

# fail
stream <- streamParserFromString("123")
( commentParser("(*","*)")(stream) )[c("status","node")]

# ok
stream <- streamParserFromString("(*123*)")
( commentParser("(*","*)")(stream) )[c("status","node")]

}

\keyword{token}
```

$\langle commentParser.R \rangle \equiv$

```
commentParser <- function(beginComment,endComment,
                    action = function(s)list(type="
  commentParser",value=s),
                    error = function(p)list(type="
  commentParser",pos=p ))

  function (stream) {

    ## begin comment
    s <- as.character()
    cstream <- streamParserNextChar(stream)

    for( i in 1:nchar(beginComment) ) {

      if ( cstream$status == "eof" || cstream$char != substr(
  beginComment,i,i) ) return(list(status="fail",node=error(
  streamParserPosition(stream)),stream=stream))

      s <- c(s,cstream$char)

      cstream <- streamParserNextCharSeq(cstream$stream)
    }

    previo <- ""
    lenEndComment <- nchar(endComment)
    repeat {
      ## end comment
      for( i in 1:lenEndComment ) {

        if ( cstream$status == "eof" ) return(list(status="fail
  ",node=error(streamParserPosition(stream)),stream=stream))

        s <- c(s,cstream$char)

        if ( cstream$char != substr(endComment,i,i) ) break()
        if ( previo == "\\" ) break()

        if ( i == lenEndComment) return(list(status="ok",
  node=action(paste(s,collapse="")), stream=cstream$stream))

        cstream <- streamParserNextCharSeq(cstream$stream)
      }
        previo <- cstream$char
        cstream <- streamParserNextCharSeq(cstream$stream)
```

```
    }

  }
```

Test for RUnit
⟨*testTokens01*⟩+≡

```
checkTokenParserOk ("(**)" ,commentParser("(*","*)"), "
  commentParser")
checkTokenParserOk ("(* *)" ,commentParser("(*","*)"), "
  commentParser")
checkTokenParserOk ("(* 123 *)" ,commentParser("(*","*)"), "
  commentParser")
checkTokenParserOk ("(* (* *)" ,commentParser("(*","*)"), "
  commentParser")
checkTokenParserOk ("(* (* * ) *)" ,commentParser("(*","*)"),
  "commentParser")
checkTokenParserOk ("/* (* * ) */" ,commentParser("/*","*/")
  , "commentParser")
checkTokenParserOk ("-- aadad\n" ,commentParser("--","\n"
  ), "commentParser")

checkTokenParserOk ("/* \\*/ */" ,commentParser("/*","*/"),
   "commentParser")
checkTokenParserOk ("\\*\\*" ,commentParser("\\*","\\*"), "
  commentParser")
checkTokenParserOk ("\\* sdas \\*" ,commentParser("\\*","\\*
  "), "commentParser")
checkTokenParserOk ("\\* \\\\* \\*",commentParser("\\*","\\
  *"), "commentParser")

checkTokenParserFail("(*" ,commentParser("(*","*)"), "
  commentParser")
checkTokenParserFail("(* " ,commentParser("(*","*)"), "
  commentParser")
checkTokenParserFail("( * *)" ,commentParser("(*","*)"), "
  commentParser")

checkTokenParserFail("/* \\*/" ,commentParser("/*","*/"), "
  commentParser")
checkTokenParserFail("\\* \\\\* " ,commentParser("\\*","\\*")
  , "commentParser")
```

2.3.11 keyword

⟨*keyword.Rd*⟩≡

```
\name{keyword}
\alias{keyword}
\title{
        Arbitrary given token.
}
\description{
  Recognises a given character sequence.
}
\usage{

  keyword(word,
          action = function(s) list(type="keyword",value=s),
          error = function(p) list(type="keyword",pos =p))

}
\arguments{
\item{word}{Symbol to be recognised.}
  < < TokenArguments > >
}
\value{
  < < TokenValue > >
}
\examples{

# fail
stream <- streamParserFromString("Hello world")
( keyword("world")(stream) )[c("status","node")]

# ok
stream <- streamParserFromString("world")
( keyword("world")(stream) )[c("status","node")]

}

\keyword{token}
```

⟨*keyword.R*⟩≡

```
keyword <- function(word,
                    action = function(s) list(type="keyword",
  value=s),
                    error = function(p) list(type="keyword",
  pos =p))

  function (stream) {

    ## begin word
    s <- as.character()
    cstream <- streamParserNextChar(stream)

    for( i in 1:nchar(word) ) {

      if ( cstream$status == "eof" || cstream$char != substr(
  word,i,i) ) return(list(status="fail",node=error(
  streamParserPosition(stream)),stream=stream))

      s <- c(s,cstream$char)

      if( i == nchar(word) )
      return(list(status="ok", node=action(paste(s,collapse=
  "")), stream=cstream$stream))

      cstream <- streamParserNextChar(cstream$stream)

    }

  }
```

Test for RUnit

⟨*testTokens01*⟩+≡

```
checkTokenParserOk ("else" ,keyword("else"),"keyword")
```

2.3.12 dots

⟨*dots.Rd*⟩≡

```
\name{dots}
\alias{dots}
\title{
        Dots sequence token.
}
\description{
  Recognises a sequence of an arbitrary number of dots.
}
\usage{

  dots(action = function(s) list(type="dots",value=s),
       error = function(p) list(type="dots",pos =p))

}
\arguments{
  < < TokenArguments > >
}
\value{
  < < TokenValue > >
}
\examples{

# fail
stream <- streamParserFromString("Hello world")
( dots()(stream) )[c("status","node")]

# ok
stream <- streamParserFromString("..")
( dots()(stream) )[c("status","node")]

}

\keyword{token}
```

⟨*dots.R*⟩≡

```
dots <- function(action = function(s) list(type="dots",value
  =s),
                 error = function(p) list(type="dots",pos =p))

  function (stream) {
    cstream <- streamParserNextChar(stream)

    if ( cstream$status == "eof" || cstream$char != "." )
  return(list(status="fail",node=error(streamParserPosition(
  stream)),stream=stream))
    s <- cstream$char
    repeat {
      stream <- cstream$stream
      cstream <- streamParserNextCharSeq(stream)

      if ( cstream$status == "eof" || ! ( cstream$char == "." )
   ) return(list(status="ok",node=action(paste(s,collapse="")
  ),stream=stream))

      s <- c(s,cstream$char)
    }
  }
```

Test for RUnit

⟨*testTokens01*⟩+≡

```
checkTokenParserOk ("." ,dots(),"dots")
checkTokenParserOk ("..." ,dots(),"dots")
checkTokenParserFail("a..." ,dots(),"dots")
```

2.3.13 empty

⟨*empty.Rd*⟩≡

```
\name{empty}
\alias{empty}
\title{
Empty token
}
\description{
  Recognises a null token. This parser always succeeds.
}
\usage{

  empty(action = function(s) list(type="empty",value=s),
        error = function(p) list(type="empty",pos =p))

}
\arguments{
  < < TokenArguments > >
}
\value{
  < < TokenValue > >
}
\details{
  \code{action} \code{s} parameter is always "".
  Error parameters exists for the sake of homogeneity with the
  rest of functions. It is not used.
}
\examples{

# ok
stream <- streamParserFromString("Hello world")
( empty()(stream) )[c("status","node")]

# ok
stream <- streamParserFromString("")
( empty()(stream) )[c("status","node")]

}

\keyword{token}
```

⟨*empty.R*⟩≡

```
empty <- function(action = function(s) list(type="empty",
  value=s),
                    error = function(p) list(type="empty",pos
  =p))

  function (stream) return(list(status="ok" ,node=action("")
  , stream=stream))
```

Test for RUnit

⟨*testTokens01*⟩+≡

```
checkTokenParserOk ("" ,empty(),"empty","")
checkTokenParserOk ("12",empty(),"empty","")
```

2.3.14 charParser

⟨*charParser.Rd*⟩≡

```
\name{charParser}
\alias{charParser}
\title{
  Specific single character token.
}
\description{
  Recognises a specific single character.
}
\usage{

  charParser(char,
             action = function(s) list(type="char",value=s),
             error = function(p) list(type="char",pos =p))

}
\arguments{
\item{char}{character to be recognised}
  < < TokenArguments > >
}
\value{
  < < TokenValue > >
}
\seealso{
  \code{\link{keyword}}
}
\examples{

# fail
stream <- streamParserFromString("H")
( charParser("a")(stream) )[c("status","node")]

# ok
stream <- streamParserFromString("a")
( charParser("a")(stream) )[c("status","node")]

# ok
( charParser("\U00B6")(streamParserFromString("\U00B6")) )[c
  ("status","node")]

}

\keyword{token}
```

⟨*charParser.R*⟩≡

```
charParser <- function(char,
                       action = function(s) list(type="char",
  value=s),
                       error = function(p) list(type="char",pos
  = p))

  function(stream) {
    cstream <- streamParserNextChar(stream)

    if ( cstream$status == "eof" ) return(list(status="fail",
  node=error(streamParserPosition(stream)),stream=stream))

    if ( cstream$char != char ) return(list(status="fail",node
  =error(streamParserPosition(stream)),stream=stream))
    return(list(status="ok" ,node=action(char),stream=
  cstream$stream))
  }
##
```

Test for RUnit

⟨*testTokens01*⟩+≡

```
checkTokenParserOk (":" ,charParser(":"),"char")
checkTokenParserFail(";" ,charParser(":"),"char")
```

———-

2.3.15 charInSetParser

⟨*charInSetParser.Rd*⟩≡

```
\name{charInSetParser}
\alias{charInSetParser}
\title{
  Single character, belonging to a given set, token
}
\description{
  Recognises a single character satisfying a predicate function.
}
\usage{

  charInSetParser(fun,
                  action = function(s) list(type="charInSet",value
  =s),
                  error = function(p) list(type="charInSet",pos =
  p))

}
\arguments{
\item{fun}{Function to determine if character belongs to a set.
          Argument "fun" is a signature function: character ->
  logical (boolean)}
  < < TokenArguments > >
}
\value{
  < < TokenValue > >
}
\examples{

# fail
stream <- streamParserFromString("H")
( charInSetParser(isDigit)(stream) )[c("status","node")]

# ok
stream <- streamParserFromString("a")
( charInSetParser(isLetter)(stream) )[c("status","node")]


}

\keyword{token}
```

⟨*charInSetParser.R*⟩≡

```
charInSetParser <- function(fun,
                            action = function(s) list(type="
  charInSet",value=s),
                            error = function(p) list(type="
  charInSet",pos =p))

  function(stream) {
    cstream <- streamParserNextChar(stream)

    if ( cstream$status == "eof" ) return(list(status="fail",
  node=error(streamParserPosition(stream)),stream=stream))

    if ( fun(cstream$char) )
      return(list(status="ok", node=action(cstream$char),
  stream=cstream$stream))
    return(list(status="fail",node=error(streamParserPosition(
  stream)),stream=stream))
  }
```

Test for RUnit

⟨*testTokens01*⟩+≡

```
checkTokenParserOk ("a" ,charInSetParser(isLetter),"charInSet")
checkTokenParserOk ("a" ,charInSetParser(function (char)
  TRUE),"charInSet")

checkTokenParserFail("a" ,charInSetParser(isDigit ),"charInSet")
checkTokenParserFail("a" ,charInSetParser(function (char)
  FALSE),"charInSet")
```

2.3.16 Tests

Combined tests

Tests using several functions for token recognition are included.

Next function executes, sequentially, recognition of several tokens.

⟨*testTokens01*⟩+≡

```
parser12 <- function(p1,p2) { function(stream) { r <- p1(
  stream); if (r$status=="ok") p2(r$stream) else return(r) }}
```

And tests combining several tokens recognition:

⟨*testTokens01*⟩+≡

```
checkTokenParserOk (" −1233.e+66" ,parser12(whitespace(),
  numberScientific()),"numberScientific" ,"−1233.e+66")
checkTokenParserFail("asd−1233.e+66" ,parser12(symbolic() ,
  numberScientific()),"numberScientific" ,"−1233.e+66")
checkTokenParserOk ("asd+1233.e+66" ,parser12(symbolic() ,
  numberScientific()),"numberScientific" ,"+1233.e+66")
checkTokenParserOk ("−1233.e+66asd" ,parser12(
  numberScientific(),symbolic()),"symbolic" ,"asd")
checkTokenParserOk("elseasd" ,parser12(keyword("else"),
  symbolic()),"symbolic" ,"asd")
checkTokenParserOk ("elseasd" ,symbolic(),"symbolic" ,"elseasd")
```

and the same exercises, from a file:

⟨*testTokens02*⟩≡

```
parseList <- list(
                    whitespace(),
                    string(),whitespace(),
                    numberScientific(),whitespace(),
                    numberNatural(),whitespace(),
                    numberScientific(),whitespace(),
                    numberScientific(),whitespace(),
                    numberInteger(),whitespace(),
                    numberFloat(),whitespace(),
                    numberFloat(),whitespace(),
                    numberScientific(),whitespace(),
                    numberScientific(),whitespace(),
                    numberScientific(),whitespace(),
                    numberScientific(),whitespace(),
                    symbolic(),whitespace(),
                    string(),whitespace(),
                    eofMark()
            )


    for( i in 1:length(parseList) ) {

      stream <- streamParserFromFileName( system.file(
"extdata","datInTest01.txt", package = "< < PACKAGE >
>"))

      for( j in 1:i ) {
        cstream <- parseList[[j]](stream)
        stream <- cstream$stream
      }
      streamParserClose(stream)
      checkEquals("ok",cstream$status,msg=paste(" hasta
",as.character(i)))
    }
```

RUnit

Here, test for previous functions are assembled.

This function for test from streamParseFromString.

⟨*runit.01tokens.RUnit*⟩≡

```
  < < testFuncionesUtiles > >
test.tokens01 <− function() {
  < < testTokens01 > >
}
```

This function for test from streamParseFromFileName.

⟨*runit.01tokens.RUnit*⟩+≡

```
test.tokens02 <− function() {
  < < testTokens02 > >
}
```

Utility functions

Functions used to carry out tests with RUnit.

printCStream Shows relevant information about a StreamParse.

⟨*testFuncionesUtiles*⟩≡

```
printCStream <− function(cstream) print(paste("[",cstream$
  node$value,"] [",cstream$status,"]",sep=""))
```

checkTokenParserOk Returns TRUE if token recognition is correct, when it is expected to be correct. Status ("ok"), token type and recognised character stream are checked.

⟨*testFuncionesUtiles*⟩+≡

```
checkTokenParserOk <− function(text,parse,type,token=text,
  msg=paste(type,": [",token,"]",sep=""))
                checkIdentical(list(status="ok",node=list(type
  =type ,value=token)),parse(streamParserFromString(text)) [c
  ("status","node")],msg=msg)
```

checkTokenParserFail Returns TRUE if token recognition does not succeed, when it is expected not to succeed.

⟨*testFuncionesUtiles*⟩+≡

```
checkTokenParserFail <- function(text,parse,type,token=text
  ,msg=paste(type,": [",token,"]",sep=""))
                          checkIdentical("fail",parse(
  streamParserFromString(text))$status,msg=msg)
```

checkSentParserOk Returns TRUE if parser recognition is correct, when it is expected to be correct. It is used to test parsers not being just tokens.

⟨*testFuncionesUtiles*⟩+≡

```
checkSentParserOk <- function(text,parse,type,sent=text,
  msg=paste(type,": [",sent,"]",sep="")) {
                                  r <- parse(
  streamParserFromString(text)) [c("status","node")]
                                  r$node$value <- paste(r$node
  $value,collapse="",sep="")
#print(r)
                                  checkIdentical(list(status="ok",
  node=list(type=type ,value=sent)),r,msg=msg)
}
```

checkSentParserFail Returns TRUE if parser recognition does not succeed, when it is expected not to succeed. It is used to test parser not being just tokens.

⟨*testFuncionesUtiles*⟩+≡

```
checkSentParserFail <- function(text,parse,type,sent=text,
  msg=paste(type,": [",sent,"]",sep="")) {
                checkIdentical("fail",parse(
  streamParserFromString(text))$status,msg=msg)
}
```

RUnit in R CMD check This function aim is to cause unitary tests to be executed with R CMD check.

The function is a slightly modified version of the one in: unit testing in R. `http://rwiki.sciviews.org/doku.php?id=developers:runit`

⟨*test.RUnit.tstR*⟩≡

```
## unit tests will not be done if RUnit is not available

if(require("RUnit", quietly=TRUE)) {
  ##
  ## --- Setup ---
  ##
  pkg <- "< < PACKAGE > >" # <-- Change to package
  name!
                                        #3
  ## When in developing
  path <- file.path(getwd(), "runit")
  if ( file.exists(path) ) {
    ## When in developing, developing environment is
  prioritised
    .libPaths(c(file.path(getwd(), "..",.libPaths()))
  } else {
    ## Otherwise, check if package already installed
    path <- system.file(package=pkg, "tests", "runit")
  }
  ##
  ##
  ##
  cat("\nRunning unit tests\n")
  print(list(pkg=pkg, getwd=getwd(), pathToUnitTests=path
  ))
  ##
  require(package=pkg, character.only=TRUE)
  ##
  wd <- setwd(path) ; path <- "."

  ## If desired, load the name space to allow testing of private
  functions
  ## if (is.element(pkg, loadedNamespaces()))
  ## attach(loadNamespace(pkg), name=paste("namespace", pkg
  , sep=":"), pos=3)
  ##
  ## or simply call PKG:::myPrivateFunction() in tests

  ## --- Testing ---

  ## Define tests
  testSuite <- defineTestSuite(
                name=paste(pkg, "unit testing"),
                dirs=path,
```

```
                testFileRegexp = "^runit.+\\.[rR]$",
                testFuncRegexp = "^test.+")
  ## Run
  tests <- runTestSuite(testSuite)

  ## Default report name
  pathReport <- file.path(path, "report")
  dir.create(path=pathReport, recursive = TRUE)

  ## Report to stdout and text files
  cat("------------------- UNIT TEST
  SUMMARY -------------------\n\n")
  printTextProtocol(tests, showDetails=TRUE,
                    fileName=file.path(pathReport, "report.txt"
  ))
  printTextProtocol(tests, showDetails=FALSE,
                    fileName=file.path(pathReport, "Summary.
  txt"))

  ## Report to HTML file
  testFileToSFLinkMap <- function(testFileName, testDir = "
  ..") {
    return(file.path("..",basename(testFileName)))
  }

  printHTMLProtocol(tests, fileName=file.path(pathReport, "
  report.html"),
                    testFileToLinkMap = testFileToSFLinkMap)

  setwd(wd)
  ## Return stop() to cause R CMD check stop in case of
  ## - failures i.e. FALSE to unit tests or
  ## - errors i.e. R errors
  tmp <- getErrors(tests)
  if(tmp$nFail > 0 | tmp$nErr > 0) {
    stop(paste("\n\nunit testing failed (#test failures: ", tmp$
  nFail,
               ", #R errors: ", tmp$nErr, ")\n\n", sep=""))
  }
} else {
  warning("cannot run unit tests -- package RUnit is not
  available")
}
```

2.4 Parsers combining parsers

In this section, functions which combine basic, primitives or combined-created parser are described. These functions are used to create new parsers.

Since these functions may be combined to construct a new parser, they must return the same data type and, therefore, they share certain homogeneity.

– As these functions are used to combine parsers, they will have as input parameters at least one parser and, moreover:

⟨*ParserCombinatorArguments*⟩≡

item{action}{Function to be executed if recognition succeeds. It takes as input parameters information derived from parsers involved as parameters}

item{error}{Function to be executed if recognition does not succeed. I takes two parameters:
itemize{
item \code{p}

with position where parser, \code{\link[< < PACKAGE > >]{streamParser}}, starts its recognition, obtained with \code{\link{streamParserPosition}}

item \code{h}

with information obtained from parsers involved as parameters, normally related with failure(s) position in component parsers.

Its information depends on how parser involved as parameters are combined and on the \code{ error} definition in these parsers.
}
}

– These functions, just as parsers which process tokens, return always the following value:

⟨*ParserCombinatorValue*⟩≡

Anonymous functions, returning a list.

\code{function(stream)} −−> \code{ list(status,node, stream) }

From these input parameters, an anonymous function is constructed. This function admits just one parameter, stream, with \code{\link[< < PACKAGE > >]{ streamParser}} class, and returns a three−field list:
itemize{
item{status}{

"ok" or "fail"}

item{node}{

With \code{action} or \code{error} function output, depending on the case}

item{stream}{

With information about the input, after success or failure in recognition}

}

2.4.1 alternation

⟨*alternation.Rd*⟩≡

\name{alternation}
\alias{alternation}
title{
Alternative phrases
}
\description{
  Applies parsers until one succeeds or all of them fail.
}
\usage{

  alternation(...,
              action = function(s) list(type="alternation",
  value=s),
              error = function(p,h) list(type="alternation",
  pos =p,h=h))

}
\arguments{
  item{...}{list of alternative parsers to be executed}
< < ParserCombinatorArguments > >
}
\details{

  In case of success, \code{action} gets the \code{node}
  from the first parse to succeed.

  In case of failure, parameter \code{h} from \code{error}
  gets a list, with information about failure from all the
  parsers processed.

}

value{
< < ParserCombinatorValue > >
}
\examples{

ok
stream <− streamParserFromString("123 Hello world")
(alternation(numberNatural(),symbolic())(stream))[c("
  status","node")]

```
# fail
stream <- streamParserFromString("123 Hello world")
( alternation(string(),symbolic())(stream) )[c("status","node
  ")]


}
\keyword{parser combinator}
```

⟨*alternation.R*⟩≡

```
alternation <- function(...,
                        action = function(s) list(type="
  alternation",value=s),
                        error = function(p,h) list(type="
  alternation",pos =p,h=h) )

  function(stream) {
    dots <- list(...)
    h <- rep(list(list()),length(dots))

    for( i in 1:length(dots) ) {
      cstream <- (dots[[i]])(stream)

      if (cstream$status=="ok") return(list(status="ok"
  ,node=action(cstream$node),stream=cstream$stream))

      h[[i]] <- cstream$node
    }
    return(list(status="fail",node=error(
  streamParserPosition(stream),h),stream=stream))

  }
```

Tests for RUnit

⟨*testRules01*⟩≡

```
# alternation
checkSentParserOk (
"'aaa 123'",
alternation(string(action=actionString),numberNatural(
  action=action)),
"alternation")

checkSentParserOk (
"1234",
alternation(string(action=actionString),numberNatural(
  action=action)),
"alternation")
```

2.4.2 option

⟨*option.Rd*⟩≡

\name{option}
\alias{option}
title{
Optional parser
}
\description{
Applies a parser to the text. If it does not succeed, an empty token is returned.

Optional parser never fails.
}
\usage{

```
option(ap,
        action = function(s ) list(type="option",value=s ),
        error = function(p,h) list(type="option",pos =p,h
=h))
```

}
\arguments{
item{ap}{Optional parser}
< < ParserCombinatorArguments > >
}
\details{

In case of success, \code{action} gets the \code{node} returned by parser passed as optional. Otherwise, it gets the \code{node} corresponding to token \code{\link{empty}}: \code{list(type="empty" ,value="")}

Function \code{error} is never called. It is defined as parameter for the sake of homogeneity with the rest of functions.

}
value{
< < ParserCombinatorValue > >
}
\examples{

```
# ok
stream <- streamParserFromString("123 Hello world")
( option(numberNatural())(stream) )[c("status","node")]
```

```
# ok
stream <− streamParserFromString(”123 Hello world”)
( option(string())(stream) )[c(”status”,”node”)]

}
\keyword{parser combinator}
```

⟨*option.R*⟩≡

```
option <− function(ap,
                   action = function(s) list(type=”
  option”,value=s ),
                   error = function(p,h) list(type=”
  option”,pos=p, h=h))

  function(stream) {
    cstream <− ap(stream)

    if (cstream$status==”ok”)
      return(list(status=”ok”,node=action(cstream$node
  ) ,stream=cstream$stream))

    else return(list(status=”ok”,node=action(list(type=
  ”empty” ,value=””)),stream=stream))

  }
```

Tests for RUnit

⟨*testRules01*⟩+≡

```
# option
checkSentParserOk (
"'aaa1234'",
option(string(action=actionString)),
"option")

checkSentParserOk (
"1234",
option(string(action=actionString)),
"option","empty")

checkSentParserOk (
"1234",
option(string(action=actionString),action=function(s) if(
  is.list(s) ) list(type="option",value=s$value) else list(
  type="option",value=s)),
"option","")

checkSentParserOk (
"'1234'",
option(string(action=actionString),action=function(s) if(
  is.list(s) ) list(type="option",value=s$value) else list(
  type="option",value=s)),
"option")
```

2.4.3 concatenation

⟨*concatenation.Rd*⟩≡

\name{concatenation}
\alias{concatenation}
title{
One phrase then another
}
\description{
  Applies to the recognition a parsers sequence. Recognition
  will succeed as long as all of them succeed.
}
\usage{

  concatenation(...,
                action = function(s) list(type="
  concatenation",value=s),
                error = function(p,h) list(type="
  concatenation",pos=p ,h=h))

}
\arguments{
item{...}{list of parsers to be executed}
< < ParserCombinatorArguments > >
}
\details{

  In case of success, parameter \code{s} from \code{action}
   gets a list with information about \code{node} from all
  parsers processed.

  In case of failure, parameter \code{h} from \code{error}
  gets the value returned by the failing parser.

}
value{
< < ParserCombinatorValue > >
}
\examples{

ok
stream <− streamParserFromString("123Hello world")
(concatenation(numberNatural(),symbolic())(stream))[c("
  status","node")]

fail

```
stream <- streamParserFromString("123 Hello world")
( concatenation(string(),symbolic())(stream) )[c("status","
  node")]

}
\keyword{parser combinator}
```

⟨*concatenation.R*⟩≡

```
concatenation <- function(...,
                          action = function(s) list(type=
  "concatenation", value=s),
                          error = function(p,h) list(type=
  "concatenation", pos=p ,h=h))

  function(stream) {

    streamFail <- stream
    dots <- list(...)
    value <- rep(list(list()),length(dots))

    for( i in 1:length(dots) ) {
      cstream <- (dots[[i]])(stream)

      if (cstream$status == "fail") return(list(status="
  fail",node=error(streamParserPosition(streamFail),cstream
  $node),stream=streamFail))

      value[[i]] <- cstream$node
      stream <- cstream$stream
    }
    return(list(status="ok",node=action(value),stream=
  stream))
##
  }
```

Test for RUnit

⟨*testRules01*⟩+≡

```
# concatenation
checkSentParserOk (
" 123",
concatenation(whitespace(action=action),numberNatural(
  action=action)),
"concatenation")

checkSentParserOk (
" 'abs 21'",
concatenation(whitespace(action=action),string(action=
  actionString)),
"concatenation")

checkSentParserOk (
" a123",
concatenation(whitespace(action=action),symbolic(action
  =action)),
"concatenation")

checkSentParserOk (
" 123 abcd",
concatenation(whitespace(action=action),numberNatural(
  action=action),whitespace(action=action),symbolic(
  action=action)),
"concatenation")

checkSentParserOk (
" 123abcd",
concatenation(whitespace(action=action),numberNatural(
  action=action),whitespace(action=action),symbolic(
  action=action)),
"concatenation")

checkSentParserOk (
" 'aa' 123",
concatenation(whitespace(action=action),string(action=
  actionString),whitespace(action=action),numberNatural(
  action=action)),
"concatenation")

checkSentParserFail(
"1 'aa' 123",
concatenation(whitespace(action=action),string(action=
  actionString),whitespace(action=action),numberNatural(
```

action=**action**)),
"concatenation")

2.4.4 repetition1N

⟨*repetition1N.Rd*⟩≡

\name{repetition1N}
\alias{repetition1N}
title{
Repeats a parser, at least once.
}
\description{
Repeats a parser application indefinitely while it is
successful. It must succeed at least once.

}
\usage{

repetition1N(rpa,
action = function(s) list(type="repetition1N
",value=s),
error = function(p,h) list(type="repetition1N
",pos=p,h=h))

}
\arguments{
item{rpa}{ parse to be applied iteratively }
< < ParserCombinatorArguments > >
}
\details{
In case of success, \code{action} gets a list with
information about the \code{node} returned by the
applied parser. List length equals the number of successful
repetitions.

In case of failure, parameter \code{h} from \code{error}
gets error information returned by the first attempt of
parser application.
}
value{
< < ParserCombinatorValue > >
}
\examples{

ok
stream <− streamParserFromString("Hello world")
(repetition1N(symbolic())(stream))[c("status","node")]

fail

stream <− streamParserFromString("123 Hello world")
(repetition1N(symbolic())(stream))[c("status","node")]

}

\keyword{parser combinator}

This function preallocates the list which will contain values returned by the iterative parser application. It may be the case that this list exceeds the actual required length. However, this implementation is preferred due to the extreme slowness in R when appending elements to a long list.

⟨*repetition1N.R*⟩≡

```
repetition1N <- function(rpa,
                         action = function(s) list(type="
  repetition1N",value=s ),
                         error = function(p,h) list(type="
  repetition1N",pos=p,h=h))


  function(stream) {
    cstream <- rpa(stream)

    if ( cstream$status == "fail" ) return(list(status="
  fail",node=error(streamParserPosition(stream),cstream$
  node),stream=stream))

    iinc <- 1000
    value <- rep(list(list()),iinc)
    imax <- iinc
    i <- 1
    value[[i]] <- cstream$node
    stream <- cstream$stream

    while( TRUE ) {
      cstream <- rpa(stream)
      if ( cstream$status == "fail" ) break()
      i <- i + 1
      if ( i >= imax ) {
        value <- c(value, rep(list(list()),iinc))
        imax <- imax + iinc
      }
      value[[i]] <- cstream$node
      stream <- cstream$stream
    }
    return(list(status="ok",node=action(value[1:i]),
  stream=stream))
  }
```

Tests for Runit.

⟨*testRules01*⟩+≡

 # repetition1N
checkSentParserOk (
"'1234'",
repetition1N(string(**action**=actionString)),
"repetition1N")

checkSentParserOk (
"'1234''1234'",
repetition1N(string(**action**=actionString)),
"repetition1N")

checkSentParserOk (
"'1234''1234''1234'",
repetition1N(string(**action**=actionString)),
"repetition1N")

2.4.5 repetition0N

⟨*repetition0N.Rd*⟩≡

\name{repetition0N}
\alias{repetition0N}
title{
Repeats one parser
}
\description{

Repeats a parser indefinitely, while it succeeds. It will return an empty token if the parser never succeeds,

Number of repetitions may be zero.
}
\usage{

repetition0N(rpa0,
action = function(s) list(type="repetition0N",value=s),
error = function(p,h) list(type="repetition0N",pos=p,h=h))

}
\arguments{
item{rpa0}{parse to be applied iteratively}
< < ParserCombinatorArguments > >
}
\details{

In case of at least one success, \code{action} gets the \code{node} returned by the parser \code{\link{repetition1N}} after applying the parser to be repeated. Otherwise, it gets the \code{node} corresponding to token \code{\link{empty}}: \code{list(type="empty" ,value="")}

Function\code{error} is never called. It is defined as parameter for the sake of homogeneity with the rest of functions.

}
value{
< < ParserCombinatorValue > >
}
\examples{

ok

```
stream <− streamParserFromString("Hello world")
( repetition0N(symbolic())(stream) )[c("status","node")]


# ok
stream <− streamParserFromString("123 Hello world")
( repetition0N(symbolic())(stream) )[c("status","node")]

}
```

\keyword{parser combinator}

⟨*repetition0N.R*⟩≡

```
repetition0N <− function(rpa0,
                         action = function(s) list(type
  ="repetition0N",value=s ),
                         error = function(p,h) list(type
  ="repetition0N",pos=p,h=h))

  option(repetition1N(rpa0),action=action,error=error)
```

Tests for RUnit

⟨*testRules01*⟩+≡

```
# repetition0N

checkSentParserOk (
"1234",
repetition0N(string(action=actionString)),
"repetition0N","empty")

checkSentParserOk (
"'1234'",
repetition0N(string(action=actionString)),
"repetition0N",
"repetition1Nlist(\"'1234'\")"
)
checkSentParserOk (
"'1234''1234'",
repetition0N(string(action=actionString)),
"repetition0N",
"repetition1Nlist(\"'1234'\", \"'1234'\")"
)
checkSentParserOk (
"'1234''1234''1234'",
repetition0N(string(action=actionString)),
"repetition0N",
"repetition1Nlist(\"'1234'\", \"'1234'\", \"'1234'\")"
)

checkSentParserOk (
"'1234''1234''1234'",
repetition0N(string(action=actionString),action=
  function(s) if( is.list(s) ) list(type="repetition0N",value
  =s$value) else list(type="repetition0N",value=s)),
"repetition0N",
)

checkSentParserOk (
"1234'1234''1234'",
repetition0N(string(action=actionString),action=
  function(s) if( is.list(s) ) list(type="repetition0N",value
  =s$value) else list(type="repetition0N",value=s)),
"repetition0N",""
)
```

2.4.6 Tests

Combined tests

Tests using several combining parsers functions are included.
Tests for RUnit
⟨*testRules02*⟩≡

```
# combinations

checkSentParserOk (
" 'aa'",
concatenation(whitespace(action=action),alternation(string
   (action=actionString),numberNatural(action=action),
   action=action)),
"concatenation")

checkSentParserOk (
" 123",
concatenation(whitespace(action=action),alternation(string
   (action=actionString),numberNatural(action=action),
   action=action)),
"concatenation")

checkSentParserOk (
" 123 ",
concatenation(whitespace(action=action),alternation(string
   (action=actionString),numberNatural(action=action),
   action=action),whitespace(action=action)),
"concatenation")
```

The same exercise from a file
⟨*testRules02*⟩+≡

```
        parser <− concatenation(
                            whitespace(),
                            string(),whitespace(),
                            repetition1N(concatenation(
  numberScientific(),whitespace()))),
                            symbolic(),whitespace(),
                            string(),whitespace(),
                            eofMark()
                    )

        stream <− streamParserFromFileName(system.file
  ("extdata","datInTest01.txt", package = "< <
  PACKAGE > >"))

        cstream <− parser(stream)
        streamParserClose(cstream$stream)
        checkEquals("ok",cstream$status)
```

RUnit

Here, tests associated with previous functions are assembled for RUnit.
This function in the case of streamParseFromString.

⟨*runit.02rules.RUnit*⟩≡

```
  < < testFuncionesUtiles > >
test.rules01 <− function() {
  < < testRules01 > >
}
```

And this one in the case of streamParseFromString.

⟨*runit.02rules.RUnit*⟩+≡

```
test.rules02 <− function() {
  < < testRules02 > >
}
```

Utility functions

Functions used to perform test with RUnit.

action In order to be used as action function and convert syntactical tree in one string.

⟨*testFuncionesUtiles*⟩+≡

```
action <- function(s) paste(s,collapse="",sep="")
```

actionString In order to be used as action function and convert syntactical tree in one quoted string (simple-quotation).

⟨*testFuncionesUtiles*⟩+≡

```
actionString <- function(s) paste("'",s,"'",collapse="",sep
  ="")
```

Chapter 3

PC-AXIS

3.1 Introduction

3.2 PC-AXIS file format syntactical analysis

3.2.1 Notes

Notes extracted from 'PC-Axis file format manual. Statistics of Finland.'

`http://tilastokeskus.fi/tup/pcaxis/tiedostomuoto2006_laaja_en.pdf`

The kind of sentences that may contain a file are:

- a table−specific keyword has the form KEYWORD=
- a variable−specific keyword has the form KEYWORD(" Variable")=
- a value−specific keyword has the form KEYWORD(" Variable","Value")=
- the form of cell and special keywords is given in the specifications of each keyword (e.g. CELLNOTE).

The definition of variables in the file:

Variables are described in PC−Axis with keywords STUB (row variables) HEADING (column variables).
It is also possible to create a variable consisting of various content variables; the keyword CONTVARIABLE is used for this.

Domains in these variables can be informed through codes, values or both.

The variable is divided into values, which are expressed as texts and possible codes. The values of variables (classes, headings)
are indicated by variable with the VALUES and codes with the CODES keywords.

Numerical values format:

The numerical values of a statistical table are given in the numerical table following the DATA
decimal separator of the figures is a full stop, the minus sign is in front of a negative figure and
separators are not used. The figures are separated from one another with space or tabulator. The
data row must have a space before the line feed. In addition to numerical values, missing, masked
hidden information can be expressed in PC−Axis data part by means of so−called dot codes.

Statistics Finlands application guideline: The following notations in accordance with the Official Statistics of
Finland standard accepted in 2005 are in use:
0.0 Magnitude less than half of unit employed (not working properly in PC−Axis)
"." Category not applicable
".." Data not available or too uncertain for presentation
"..." Data subject to secrecy
"...." Magnitude nil (do not use a dash**!**)
"....."
"......"
"−" Magnitude nil (do not use in PC−Axis because it is interpreted as zero).

It must be pointed out that some files follow PC-AXIS definition in a quite lax manner. Grammar has been adapted in order to be able to read the widest number of versions possible.

On the other hand, there may be some problems with some 'encoding'. This piece of comment is parametrized here, so it can be repeated throughout the documentation.

⟨*encodingProblem*⟩≡

```
#
# Error messages like
# " ... invalid multibyte string ... "
# or warnings
# " input string ... is invalid in this locale"
#
# For example, in Linux the error generated by this code:
 name <- "http://www.ine.es/pcaxisdl//t20/e245/p04/
a2009/l0/00000008.px"
stream <- streamParserFromString( readLines( name ) )
cstream <- pcAxisParser(stream)
if ( cstream$status == 'ok' ) cube <- pcAxisCubeMake
(cstream)
#
# is caused by files with a non-readable 'encoding'. In the
case where it could be read, there may also be problems
with string-handling functions, due to multibyte
characters. In Windows, according to \code{link{Sys.
getlocale}()}, file may be read but accents, ñ, ... may not
be correctly recognised.

#
# There are, at least, the following options:
# - File conversion to utf-8, from the OS, with
# "iconv - Convert encoding of given files from one
encoding to another"
#
# - File conversion in R:
name <- "http://www.ine.es/pcaxisdl//t20/e245/p04/
a2009/l0/00000008.px"
stream <- streamParserFromString( iconv( readLines(
name ), "IBM850", "UTF-8") )
cstream <- pcAxisParser(stream)
if ( cstream$status == 'ok' ) cube <- pcAxisCubeMake
(cstream)
#
# In the latter case, latin1 would also work, but accents, ñ
, ... would not be correctly read.
#
# - Making the assumption that the file does not contain
multibyte characters:
#
localeOld <- Sys.getlocale("LC_CTYPE")
Sys.setlocale(category = "LC_CTYPE", locale = "C")
```

```
#
name <−
  "http://www.ine.es/pcaxisdl//t20/e245/p04/a2009/l0/
00000008.px"
stream <− streamParserFromString( readLines( name ) )
cstream <− pcAxisParser(stream)
if ( cstream$status == 'ok' ) cube <− pcAxisCubeMake
(cstream)
#
Sys.setlocale(category = "LC_CTYPE", locale =
localeOld)
#
# However, some characters will not be correctly read (
accents, ñ, ...)
```

3.2.2 man

⟨*pcAxisParser.Rd*⟩≡

\name{pcAxisParser}
\alias{pcAxisParser}
title{
Parser for PC−AXIS format files
}
\description{
  Reads and creates the syntactical tree from a PC−AXIS
  format file or text.
}
\usage{
    pcAxisParser(streamParser)
}
\arguments{
item{streamParser}{stream parse associated to the file/
  text to be recognised}
}
\details{

Grammar definition, wider than the strict PC−AXIS
 definition
 \preformatted{

pcaxis = { rule } , eof ;

rule = keyword ,
                  ['[' , language , ']'] ,
                  ['(' , parameterList , ')'] ,
                  = ,
                  ruleRight ;

parameterList = parameter , { ',' , parameterList } ;

ruleRight = string , string , { string } , ';'
                | string , { ',' , string } , ';'
                | number , sepearator , { , number } , (';' |
 eof)
                | symbolic
                | 'TLIST' , '(' , symbolic ,
                                  ((')' , { ',' , string })
                                    |
                                    (',' , string , '−' , string ,
  ')')
) , ';'
                ;

keyword = symbolic ;

language = symbolic ;

parameter = string ;

separator = ' ' | ',' | ';' ;

eof = ? eof ? ;

string = ? string ? ;

symbolic = ? symbolic ? ;

number = ? number ? ;
}

Normally, this function is a previous step in order to
eventually call \code{pcAxisCubeMake}:

\code{
cstream <- pcAxisParser(stream)
if (cstream$status == 'ok') cube <- pcAxisCubeMake
(cstream)
}
}
value{
Returns a list with "status" "node" "stream":
item{status}{ "ok" or "fail"}

item{stream}{Stream situation after recognition}

item{node}{List, one node element for each "keyword" in
PC-AXIS file.
Each node element is a list with: "keyword" "language" "
parameters" "ruleRight":
itemize{
item{keyword}{

PC-AXIS keyword}
item{language}{

language code or ""}

```
\item{parameters}{

   null or string list with parenthesised values associated to
  keyword }
\item{ruleRight}{

   is a list of two elements, "type" "value" :

 If type = "symbol", value = symbol

 If type = "liststring", value = string vector, originally
  delimited by ","

 If type = "stringstring", value = string vector, originally
  delimited by blanks, new line, ...

 If type = "list" , value = numerical vector, originally
  delimited by ","

 If type = "tlist" , value = (frequency, "limit" keyword ,
  lower-limit , upper-limit) or (frequency, "list" keyword ,
  periods list ) }

  }}
}
\examples{

  \dontrun{
    ## significant time reductions may be achieve by doing:
    library("compiler")
    enableJIT(level=3)
  }

  name <- system.file("extdata","datInSFexample6__1.px",
   package = "< < PACKAGE > >")
  stream <- streamParserFromFileName(name,encoding="
  UTF-8")
  cstream <- pcAxisParser(stream)
  if ( cstream$status == 'ok' ) {

    ## HEADING
    print(Filter(function(e) e$keyword=="HEADING",
  cstream$node)[[1]] $ruleRight$value)
```

```
    ## STUB
    print(Filter(function(e) e$keyword==”STUB”,cstream$
  node)[[1]] $ruleRight$value)

    ## DATA
    print(Filter(function(e) e$keyword==”DATA”,cstream$
  node)[[1]] $ruleRight$value)

  }

  \dontrun{
    < < encodingProblem > >
    }

}

\references{

PC−Axis file format.

\url{http://www.scb.se/Pages/List_____314011.aspx}


PC−Axis file format manual. Statistics of Finland.

\url{http://tilastokeskus.fi/tup/pcaxis/tiedostomuoto2006
  _laaja_en.pdf}

}

\keyword{PC−AXIS}
```

3.2.3 code

⟨*pcAxisParser.R*⟩≡

```
pcAxisParser <− function(streamParser) {
```

Function to print where the error happens

⟨*pcAxisParser.R*⟩+≡

```
errorFun <- function(strmPosition,h=NULL,type="") {
  if ( is.null(h) || type != "concatenation" )
    print(paste("Error from line:",strmPosition$line,"
position:",strmPosition$linePos))
  else errorFun(h$pos,h$h,h$type)

  return(list(type=type,pos=strmPosition,h=h))
}
```

Auxiliary function to recognise frequency and date intervals in time series

⟨*pcAxisParser.R*⟩+≡

```
tlistParse <−
  concatenation(keyword(”TLIST”),
                whitespace(),
                charParser(”(”),
                whitespace(),
                symbolic(),
                whitespace(),
                alternation(
                        concatenation(charParser(”)”),
                                repetition1N(
                                        concatenation
  (
whitespace(),

charParser(”,”),
whitespace(),
string(action = function(s) s),
action = function(s) s[[4]]),

action = function(s) list(type=”list”,value=unlist(s))),
                                        action =
  function(s) s[[2]] ),
                        concatenation(charParser(”,”),
                                        whitespace(),
                                        string(action
  = function(s) s),
                                        whitespace(),
                                        charParser(”−”
  ),
                                        whitespace(),
                                        string(action
  = function(s) s),
                                        whitespace(),
                                        charParser(”)”)
  ,
                                        action =
  function(s) list(type=”limit”,value=s[c(3,7)]) ),
                                action = function(s) s),
                whitespace(),
                charParser(”;”),
                action = function(s) list(type=”tlist”,
  value=s[c(5,7)]) )
```

Here it is implemented the left hand side, before '=' sign in 'rule'.

⟨*pcAxisParser.R*⟩+≡

```
##
  rule <- concatenation(
# left rule
# keyword
                        symbolic(action = function(s)
  s),
# optional multi-language
                        option(concatenation(
                                              whitespace
  (),
                                              charParser(
  "["),
                                              whitespace
  (),
                                              symbolic(
  action = function(s) s),
                                              whitespace
  (),
                                              charParser(
  "]"),
                                              action =
  function(s) s[4]),
                                    action = function(s) if
  (!is.null(s$type) && s$type=="empty") NULL else s),
# optional variables / values
                        option(concatenation(
                                              whitespace
  (),
                                              charParser(
  "("),
                                              whitespace
  (),
                                              string(
  action = function(s) s),
                                              repetition0N
  (
                                                          concatenation
  (
whitespace(),
charParser(","),
whitespace(),
string(action = function(s) s),
action = function(s) s[4]), # end concatenation
 action = function(s) if (!is.null(s$type) && s$type=="
```

```
empty") NULL else s$value),# end repetition0N
                                                    whitespace
(),
                                                    charParser(
")"),
                                                    action =
function(s) if( is.null(s[[5]]) ) s[c(4)] else c(s[c(4)],unlist
(s[c(5)])), # end concatenation
                                        action = function(s) if
(!is.null(s$type) && s$type=="empty") NULL else s),
# end option
                                whitespace(),
                                charParser("="),
```

Rule right hand side, after '=" sign

⟨*pcAxisParser.R*⟩+≡

```
whitespace(),
alternation(
```

Alternatively, it may be:

A string, followed by other strings delimited by blanks.

⟨*pcAxisParser.R*⟩+≡

```
        concatenation(
            string(action = function(s)s), repetition1N(
 concatenation(
                                    whitespace(),
                                    string(action = function(s)
 s),
                                    action = function(s) s[[2]]),
                                action = function(s) s ),
                whitespace(),
                charParser(";"),
                action = function(s) list(type="stringstring
 ",value=c(s[c(1)],unlist(s[c(2)])) ),
```

Strings lists, delimited by commas.

⟨*pcAxisParser.R*⟩+≡

```
concatenation( string(action = function(s) s) ,
               repetition0N(
                            concatenation(
                                          whitespace(),
                                          charParser(","),
                                          whitespace(),
                                          string(action =
  function(s) s),
                                          action =
  function(s) s[[4]]),
                            action = function(s) if (!is.
  null(s$type) && s$type=="empty") NULL else s$value)
  ,
               whitespace(),
               charParser(";"),
               action = function(s) list(type="liststring",
  value=if( is.null( s[[2]] ) ) s[c(1)] else c(s[c(1)],unlist(s[c
  (2)],use.names = FALSE))) ),
```

Numbers list or the DATA= format, which may include variables codes and may end without ";" and with end-of-file marks.

⟨*pcAxisParser.R*⟩+≡

```
concatenation(
                alternation(
                                numberScientific(action =
  function(s) s),
                                string(action = function(s) s),
                                dots (action = function(s) s),
                                action = function(s) s) ,
                repetition0N(
                                  concatenation(
                                                separator(),
                                                alternation(
                                                                numberScientific
  ( action = function(s) s),
                                                                string(
  action = function(s) s),
                                                                dots (
  action = function(s) s),
                                                                action
   = function(s) s) ,
                                                action =
  function(s) s[[2]]),
                 action = function(s) if (!is.null(s$type) &
  & s$type==”empty”) NULL else s$value), # en
  repetition0N
                whitespace(),
                alternation(
                                concatenation(charParser(”;”),
                                                whitespace(),
                                                option(
  concatenation( charParser(”;”), whitespace())),
                                                option(
  concatenation( charParser(”\032”), whitespace()))
                                                ),
                                concatenation(charParser(”\032”),
                                                whitespace()
                                                ),
                                eofMark()),
                action = function(s) list(type=”list”,value=
  if( is.null( s[[2]] ) ) s[c(1)] else c(s[c(1)],unlist(s[c(2)],
  use.names = FALSE))) ),
```

Frequency and date intervals in time series.

⟨*pcAxisParser.R*⟩+≡

tlistParse,

Right hand side, make up by only one symbol.

⟨*pcAxisParser.R*⟩+≡

```
concatenation(symbolic(action = function(s) s),
                whitespace(),
                charParser(";"),
                action = function(s) list(type="symbol",
  value=s[[1]])),
```

When finishing the rule, after ";", blanks are allowed and, within each rule, relevant information is contained in position 1, 2, 3, 7, which may be null and correspond to "keyword", "language", "parameters", "ruleRight".

⟨*pcAxisParser.R*⟩+≡

```
                action = function(s) s), ## end alternation
                whitespace(), ## blanks behind ;
                                action = function(s) { rule <-
   s[c(1,2,3,7)] ; names(rule) <- c("keyword","language","
  parameters","ruleRight") ; rule }
                                )
```

Finally, a PC-AXIS file consists in the repetition of 'rule', n times, followed by the end-of-file mark.

⟨*pcAxisParser.R*⟩+≡

```
cstream <- concatenation(repetition1N(rule,action =
   function(s) s) ,eofMark(error=errorFun),action =
   function(s) s[[1]]) (streamParser)
   return(cstream)
}
```

3.2.4 test

⟨*runit.PCAXIS*⟩≡

```
# Check kinds of tokens in PC−AXIS files

name <− system.file("extdata","datInSFexample6_1.px",
  package = "< < PACKAGE > >")
rule <− alternation(symbolic(),string(),numberScientific(),
                    charParser("("),charParser(")"),
                    charParser("["),charParser("]"),
                    charParser(","),charParser(";"),
  charParser("−"),charParser("="),
                    eofMark(),
                    charParser(' '),
                    charParser('\n'),charParser("\r"),
  charParser("\t"))

stream <− streamParserFromFileName(name,encoding="
  UTF−8")

cstream <− rule(stream)
printCStream(cstream) ;
checkEquals("ok",cstream$status,'tokens read')

while( cstream$status == "ok" && cstream$node$value$
  type != "eofMark" ) {
  cstream <− rule(cstream$stream)
  printCStream(cstream) ;
  checkEquals("ok",cstream$status,'tokens read')
}

if( cstream$status == "fail" ) {
  print(streamParserNextChar(cstream$stream)) ;
    print(streamParserPosition(cstream$stream)) ;
}
streamParserClose(cstream$stream)


# step by step PC−AXIS format checking
checkEquals("ok",pcAxisParser(streamParserFromString('
  CHARSET="ANSI";'))$status,'CHARSET')

checkEquals("ok",pcAxisParser(streamParserFromString('
  AXIS−VERSION="2000"; '))$status,'AXIS−VERSION')

checkEquals("ok",pcAxisParser(streamParserFromString('
  DECIMALS=0;'))$status,'DECIMALS')
```

```
checkEquals("ok",pcAxisParser(streamParserFromString('
  SUBJECT-AREA="Väestö";'))$status,'SUBJECT-
  AREA')

checkEquals("ok",pcAxisParser(streamParserFromString('
  SUBJECT-AREA="Väestö";
'))$status,'SUBJET-AREA/TITLE/CONTENTS')

checkEquals("ok",pcAxisParser(streamParserFromString('
  TITLE="Väestö 31.12. muuttujina Sukupuoli, Kunta,
  Vuosi ja Siviilisääty";'))$status,'TITLE')

checkEquals("ok",pcAxisParser(streamParserFromString('
  SUBJECT-AREA="Väestö";
TITLE="Väestö 31.12. muuttujina Sukupuoli, Kunta, Vuosi
   ja Siviilisääty";
CONTENTS="Väestö 31.12.";'))$status,'SUBJET-AREA/
  TITLE/CONTENTS')

checkEquals("ok",pcAxisParser(streamParserFromString('
  HEADING="Vuosi","Siviilisääty";'))$status,'HEADING'
  )

checkEquals("ok",pcAxisParser(streamParserFromString('
  VALUES("Kunta")="Espoo","Helsinki","Vantaa";'))$
  status,'VALUES')

checkEquals("ok",pcAxisParser(streamParserFromString('
  DATA=57516 43030 100546 57516 43030 100546 91202
  67623 158825 91202 67623 158825 ; ' ))$status,'DATA 01'
  )

checkEquals("ok",pcAxisParser(streamParserFromString('
  DATA=57516 43030 100546 57516 43030 100546 91202
  67623 158825 91202 67623 158825 ' ))$status,'DATA 02')

checkEquals("ok",pcAxisParser(streamParserFromString('
  DATA=
57516 43030 100546 57516 43030 100546
144564 85339 229903 144564 85339 229903
47131 33688 80819 47131 33688 80819
53821 43375 97196 53821 43375 97196
151536 86184 237720 151536 86184 237720
```

```
44071 33935 78006 44071 33935 78006
111337 86405 197742 111337 86405 197742
296100 171523 467623 296100 171523 467623
91202 67623 158825 91202 67623 158825;’))$status,’DATA’)

checkEquals(”ok”,pcAxisParser(streamParserFromString(’
  DATA=
39669394 19399549 20269845
13782827 6554619 7228208
    52709 28052 24657
   739409 382664 356745
   800097 406813 393284
  1444239 727941 716298
  3129220 1563613 1565607
  3599227 1789972 1809255
  4525296 2241138 2284158
  4996377 2467192 2529185
  3157049 1549638 1607411
  3442944 1687907 1755037
        0 0 0
\032’))$status,’DATA’)
```

3.3 PC-AXIS cube : Semantics (PC-AXIS file content

3.3.1 Notes

Notes extracted from 'PC-Axis file format manual. Statistics of Finland.'

`http://tilastokeskus.fi/tup/pcaxis/tiedostomuoto2006_laaja_en.pdf`

Here, the meaning of the most relevant metadata keywords in a PC-AXIS cube is outlined. In order to extract numerical data, keywords STUB, HEADING, VALUES, CODES are relevant.

"Managing the character set", CHARSET . CODEPAGE
"Basic language","PC−Axis compatibility" AXIS−VERSION
"Creation and updating", "Information about the creation" CREATION−DATE
"The information on statistical topics" SUBJECT−AREA, SUBJECT−CODE,DATABASE, MATRIX
"Table titling" TITLE, DESCRIPTION, DESCRIPTIONDEFAULT , CONTENTS
"Measurement units" UNITS
"Variables", "Row variables" STUB
"Variables", "Column variables" HEADING
"The classifications (variable values)" VALUES, CODES.
"Decimal numbers", "The number of saved decimal" DECIMALS
"Decimal numbers" , "Presentation precision" SHOWDECIMALS
"Creation and updating","Information about updating" LAST−UPDATED
"Source and contact information" SOURCE, CONTACT.
"COPYRIGHT" COPYRIGHT
"saving and display mode of variable texts and codes", PRESTEXT,
0,"Do not use**!**"
1,"texts and codes are saved to their own keywords"
2,"save texts as codes and vice versa"
3 "indicates thatboth texts and codes are shown in the table."

"the file name of the table specification attached to the table without the extension.",INFOFILE

”the name of the map template file corresponding to the
table.”, MAP
”the time series data are working−day adjusted. YES/NO
”, DAYADJ
”the time series data are working−day adjusted. YES/NO
”, SEASADJ

TIMEVAL is a variable−specific specification that can
appear only once in a table.
The keyword value is the TLIST specification expressing
the time scale and period, which can be given either in
time limit or list format.
TIMEVAL(”Time”)=TLIST(M1,”199605”−”199704”);
TIMEVAL(”Time”)=TLIST(M1)
,”199605”,”199606”,”199607”,”199608”,”199609”,”199610”,”199611”,”199612”,”199701”,

The following TLIST specifications are used:
A1 in annual statistics in form CCYY (CC century, YY
year)
H1 quarterly in form CCYYH, where H is 1 or 2
Q1 quarterly in form CCYYQ, where Q is 1−4.
M1 in monthly statistics in form CCYYMM, where MM is
01−12
W1 in weekly information in form CCYYWW, where WW
is 01−52.

3.3.2 man

⟨*pcAxisCubeMake.Rd*⟩≡

```
\name{pcAxisCubeMake}
\alias{pcAxisCubeMake}
\title{
        Creates PC−AXIS cube
}

\description{
  From the constructed syntactical tree, structures in R are
  generated. These structures contain the PC−AXIS cube
  information.
}

\usage{
pcAxisCubeMake(cstream)
}
\arguments{
\item{cstream}{tree returned by the PC−AXIS file
  syntactical analysis }
}

%\details{}

\value{
  It returns a list with the following elements:
 \item{pxCube (data.frame)}{
    \tabular{ll}{
      headingLength \tab Number of variables in "
  HEADING".\cr
      StubLength \tab Number of variables in "STUB".\cr
      frequency \tab Data frequency if "TIMEVAL" is
  present. \cr
 }}
 \item{pxCubeVariable (data.frame)}{
    \tabular{ll}{
        variableName \tab Variable name.\cr
        headingOrStud \tab Indicator, whether the variable
  appears in "HEADING" or "STUB". \cr
        codesYesNo \tab Indicator, whether there is "
  CODES" associated to the variable.\cr
        valuesYesNo \tab Indicator, whether there is "
  VALUES" associated to the variable.\cr
        variableOrder \tab Variable order number in "
  HEADING" or "STUB"\cr
        valueLength \tab Number of different "CODES"
```

```
 and/or "VALUES" associated with the variable.\cr
}}
\item{pxCubeVariableDomain (data.frame)}{
   \tabular{ll}{
      variableName \tab Variable name.\cr
      code \tab Value code when "CODES" is present.\cr
      value \tab Value literal when "VALUES" is present.\
 cr
      valueOrder \tab Variable order number in "CODES"
 and/or "VALUES".\cr
      eliminationYesNo \tab Indicator, whether the value
 for the variables is present in "ELIMINATION".\cr
}}
\item{pxCubeAttrN}{data.frame list, one for each
 different parameters cardinalities appearing in "keyword"
\itemize{
\item{pxCubeAttrN$A0 (data.frame)}{
 \tabular{ll}{
       keyword \tab Keyword.\cr
       language \tab Language code o "".\cr
       length \tab Number of elements of value list.\cr
       value \tab Associated data, keyword[language] =
 value.\cr
  }}
\item{pxCubeAttrN$A1 (data.frame)}{
 \tabular{ll}{
       keyword \tab Keyword.\cr
       language \tab Language code o "".\cr
       arg1 \tab Argument value.\cr
       length \tab Number of elements of value list.\cr
       value \tab Associated data , keyword[language](arg)
  = value.\cr
 }}
\item{pxCubeAttrN$A2 (data.frame)}{
 \tabular{ll}{
       keyword \tab Keyword.\cr
       language \tab Language code o "".\cr
       arg1 \tab Argument one value.\cr
       arg2 \tab Argument to value.\cr
       length \tab Value list number of elements.\cr
       value \tab Associated data , keyword[language](arg
 1,arg2) = value.\cr
}}
}}
```

```
 \item{pxCubeData (data.frame)}{
  \tabular{ll}{
        StubLength + headingLength columns \tab, with
  variables values, ordered according to "STUB" and
  followed by those appearing in "HEADING".

        Variables names correspond to variable names.\cr
        data \tab associated value.\cr
  }}

  Returned value short version is:
 \preformatted{
Value:
 pxCube (headingLength, StubLength)
 pxCubeVariable (variableName , headingOrStud,
  codesYesNo, valuesYesNo, variableOrder, valueLength)
 pxCubeVariableDomain(variableName , code, value,
  valueOrder, eliminationYesNo)
 pxCubeAttr -> list pxCubeAttrN(key, {variableName} ,
  value)
 pxCubeData ({variableName}+, data) varia signatura

}

}
\examples{

  \dontrun{
    ## significant time reductions may be achieve by doing:
    library("compiler")
    enableJIT(level=3)
  }

  name <- system.file("extdata","datInSFexample6_1.px",
   package = "< < PACKAGE > >")

  stream <- streamParserFromFileName(name,encoding="
  UTF-8")

  cstream <- pcAxisParser(stream)
  if ( cstream$status == 'ok' ) {
    cube <- pcAxisCubeMake(cstream)
```

```
    ## Variables
    print(cube$pxCubeVariable)

    ## Data
    print(cube$pxCubeData)

  }

  \dontrun{
    < < encodingProblem > >
    }

}
\references{

  PC−Axis file format.

  \url{http://www.scb.se/Pages/List_____314011.aspx}


  PC−Axis file format manual. Statistics of Finland.

  \url{http://tilastokeskus.fi/tup/pcaxis/tiedostomuoto
  2006_laaja_en.pdf}

}

\keyword{PC−AXIS}
```

3.3.3 code

⟨*pcAxisCubeMake.R*⟩≡

```
  pcAxisCubeMake <- function(cstream) {
##
    stringsAsFactorsOld <- options("stringsAsFactors")
    on.exit(options(stringsAsFactors=stringsAsFactorsOld
  $stringsAsFactors
))
    options(stringsAsFactors=FALSE)
##
## Creates a vector with values associated with a keyword,
  taking into account data type
    avecMapValue <- function(e) {
      mapValue <- e[[4]]
      if (length(mapValue) == 0 ) return(c())
      switch(mapValue$type,
             stringstring=as.character(mapValue$value),
             liststring =as.character(mapValue$value),
             symbol =as.character(mapValue$value),
             list =as.character(mapValue$value),
             tlist =c(as.character(unlist(mapValue$value
  [[1]]$value)),as.character(mapValue$value[[2]]$type),as.
  character(unlist(mapValue$value[[2]]$value))),
             as.character(mapValue$value))
    }
##
## Flattens data associated with a keyword
## "rownum" "keyword" "variable" "variableValue" "
  mapValueType" "mapValueLength" "mapValue"

    keywordsAplana <- function(e)
      list( keyword = e[[1]][[1]],
            language= if(! is.null(e[[2]]) && length(e[[2]])
  > 0 ) e[[2]][[1]] else list(as.character("")),
            arity =length(e[[3]]),
            args = e[[3]],
            type = e[[4]]$type,
            length = length(e[[4]]$value),
            value = avecMapValue(e) )
    ##
    ##
    productoCartersiano <- function(a,b) unlist(lapply(
  a,function(e1) lapply(b,function(e2) c(list(),e1,e2))),
  recursive=FALSE,use.names = FALSE)
    ##
    combinaValores <- function(listVar) {
```

```
    listVar <- Filter(function(x) length(x) > 0, listVar
)
    if( length(listVar) <= 1 ) as.list(listVar[[1]])
    else {
      expa <- listVar[[1]]
      for( i in 2:length(listVar) ) expa <-
productoCartersiano(expa,listVar[[i]])
      return(expa)
    }
  }
  proyect <- function(listlist,n) lapply(listlist,function
(e) e[[n]])
  ##
  ##
  keywords <- lapply(cstream[[2]],function(x)
keywordsAplana(x))

  selectClauses <- function(data,filter,fields)
    if( missing(filter) ) { if ( missing(fields) ) data else
 lapply( data ,function(e) e[fields]) } else
  { if ( missing(fields) ) Filter(filter,data) else lapply(
Filter(filter,data),function(e) e[fields]) }

  getColumn <- function(data,field) as.vector(unlist(
lapply(data,function(e) e[[field]])))
  ##
  STUB <- selectClauses(keywords, function(e) e$
language=="" && e$arity == 0 && e$keyword == "
STUB" )[[1]]
  HEADING <- selectClauses(keywords, function(e) e$
language=="" && e$arity == 0 && e$keyword == "
HEADING")[[1]]
  DATA <- selectClauses(keywords, function(e) e$
language=="" && e$arity == 0 && e$keyword == "
DATA" )[[1]]
  TIMEVAL <- selectClauses(keywords, function(e) e$
language=="" && e$arity == 1 && e$keyword == "
TIMEVAL")
  KEYS <- selectClauses(keywords, function(e) e$
language=="" && e$arity == 1 && e$keyword == "
KEYS" )
  ##
  frequency <- if ( length(TIMEVAL) > 0 )
TIMEVAL[[1]]$value[[1]] else as.character(NA)
```

```
  keysFlag <- length(KEYS) > 0
  ##
  ## pxCube:
  ## (headingLength, StubLength, frequency)
  pxCube <-data.frame(headingLength=HEADING$
length,StubLength=STUB$length,frequency=
frequency)
  ##
  ## pxCubeAttrN
  aritys <- unique(as.vector(lapply(keywords,
function(e) e$arity)))
  pxCubeAttrN <- list()
  for ( a in aritys ) {
    ##print(a)
    ks <- Filter(function(e) e$arity == a && e$
keyword != "DATA" , keywords )
    ksap <- lapply(ks, function(e) c(e$keyword,e$
language,e$args,e$length,value=paste('"',e$value,'"',sep
="",collapse=" ")[[1]]))
    ksdf <- data.frame(lapply(do.call(function(...)
rbind.data.frame(..., deparse.level=0),ksap),unlist))
    names(ksdf) <- c("keyword","language",if(a>0)
paste("arg",1:a,sep="") else NULL,"length","value")
    pxCubeAttrN[[paste("A",as.character(a),sep="")]]
<- ksdf
  }
  ##
  ##
  ##
  ## CODES or VALUES registries
  codesvalues <- selectClauses(keywords,
                               function(e) e$language
=="" && e$arity == 1 && e$keyword %in% c("CODES
","VALUES"),
                               c("keyword","args","
length","value"))
  ## Value number, by variables
  codesValuesLength <- aggregate(getColumn(
codesvalues,"length"),list(getColumn(codesvalues,"args")),
max)
  rownames(codesValuesLength) <- codesValuesLength
$Group.1
  colnames(codesValuesLength) <- c("variable","
valueLength")
```

```
  ## Whether use "codes" or "values" in the numerical
value key/pk
  codesOValues <- aggregate(getColumn(codesvalues,"
keyword"),list(getColumn(codesvalues,"args")),function(
x) min(as.character(x)))
  rownames(codesOValues ) <- codesOValues$Group.1
  colnames(codesOValues ) <- c("variable","keyword")

  ## Information union: Number of values, by variables
and whether use "codes" or "values" in the numerical
value key/pk
  variableMeta <- merge(codesOValues,
codesValuesLength)
  rownames(variableMeta) <- codesOValues$variable
  ##

  valueselimination <- selectClauses(keywords,function(
e) e$language=="" && e$arity == 1 && e$keyword ==
 "ELIMINATION")


  ## pxCubeVariable:
  ## (variableName, headingOrStud, codesYesNo,
valuesYesNo, variableOrder, valueLength)
  variableName <- unlist(c(HEADING$value,STUB$
value))

  headingOrStud <- c(rep("HEADING",HEADING$
length),rep("STUB" ,STUB$length))

  codesYesNo <- sapply(variableName,function(v)
length(selectClauses(codesvalues,function(e) e$keyword
== "CODES" && e$args[[1]] == v)) >= 1 )

  valuesYesNo <- sapply(variableName,function(v)
length(selectClauses(codesvalues,function(e) e$keyword
== "VALUES" && e$args[[1]] == v)) >= 1 )

  variableOrder <- c(1:HEADING$length,1:STUB$
length)

  valueLength <- unlist(variableMeta[variableName,"
valueLength"])
```

```
  pxCubeVariable<- cbind.data.frame(variableName,
headingOrStud, codesYesNo, valuesYesNo, variableOrder,
valueLength)

  colnames(pxCubeVariable) <- c("variableName","
headingOrStud","codesYesNo","valuesYesNo","
variableOrder","valueLength")
  ##
  ## pxCubeVariableDomain:
  ## (variableName, code, value, valueOrder,
eliminationYesNo)
  variableName <- unlist(sapply(1:length(
pxCubeVariable$variableName),function(i) rep(
pxCubeVariable$variableName[i],pxCubeVariable$
valueLength[i])))

  code <- unlist(sapply(1:length(pxCubeVariable$
variableName),function(i) if(pxCubeVariable$codesYesNo
[i]) selectClauses(codesvalues,function(e) e$keyword ==
"CODES" && e$args[[1]] == pxCubeVariable$
variableName[i],"value") else rep(as.character(NA),
pxCubeVariable$valueLength[i])))

  value <- unlist(sapply(1:length(pxCubeVariable$
variableName),function(i) if(pxCubeVariable$
valuesYesNo[i]) selectClauses(codesvalues,function(e) e$
keyword == "VALUES" && e$args[[1]] ==
pxCubeVariable$variableName[i],"value") else rep(as.
character(NA),pxCubeVariable$valueLength[i])))

  valueOrder <- unlist(sapply(1:length(
pxCubeVariable$variableName),function(i) 1:
pxCubeVariable$valueLength[i]))

  eliminationYesNo <- unlist(sapply(1:length(value),
function(i) length(selectClauses(valueselimination,
function(e) e$keyword == "ELIMINATION" && e$
args[[1]] == variableName[i] && e$value == value[i])) >
 0))

  if ( length(code) != length(value) ) stop("Error:
CODE and VALUE non-consistent")

  pxCubeVariableDomain <- cbind.data.frame(
```

```
variableName,code,value,valueOrder,eliminationYesNo)
      names(pxCubeVariableDomain) <- c("
variableName","code","value","valueOrder","
eliminationYesNo")
  ##
  ## pxCubeData:
  ## ({variableName}+, data)
  if ( ! keysFlag ) {
    STUDValues <- combinaValores(lapply(STUB$
value ,function(v) unlist(selectClauses(codesvalues,
function(e) e$keyword == variableMeta[v,"keyword"] &
& e$args[[1]] == v,"value"))))

    HEADINGValues <- combinaValores(lapply(
HEADING$value,function(v) unlist(selectClauses(
codesvalues,function(e) e$keyword == variableMeta[v,"
keyword"] && e$args[[1]] == v,"value"))))

    pxCubeData <- combinaValores(list(STUDValues,
HEADINGValues))
    ##
    ## data.frame construction, from the values
combinations list
    e1 <- pxCubeData[[1]]
    pxCubeData <- do.call("cbind.data.frame",
                          lapply(1:length(e1),function(i
) as.character(proyect(pxCubeData,i))))
    colnames(pxCubeData) <- pxCubeVariable$
variableName
    ##
    ##
    if( length(DATA$value) < nrow(pxCubeData) )
stop("Error: variables and data inconsistency"," ",length
(DATA$value) ," ", nrow(pxCubeData) )

    if( length(DATA$value) > nrow(pxCubeData) )
warning("Warnings: variables and data inconsistency"," "
,length(DATA$value) ," ", nrow(pxCubeData) )

    pxCubeData$data <- DATA$value[1:nrow(
pxCubeData)]
    names(pxCubeData) <- c(STUB$value,HEADING$
value,"data")
    rownames(pxCubeData) <- 1:nrow(pxCubeData)
```

```
  } else {
    HEADINGValues <- combinaValores(lapply(
HEADING$value,function(v) unlist(selectClauses(
codesvalues,function(e) e$keyword == variableMeta[v,"
keyword"] && e$args[[1]] == v,"value"))))

    numFields <- pxCube$StubLength + length(
HEADINGValues)
    keysdata <- DATA$value
    ## print(paste(pxCube$StubLength," - ",length(
HEADINGValues)))
    ## print(paste( length(keysdata)," - ",numFields))
    numreg <- length(keysdata)/numFields

    keys <- sapply(1:numreg,function(i) as.list(
keysdata[((i-1)*numFields+1):((i-1)*numFields+pxCube
$StubLength)]),simplify = FALSE)

    pxCubeData <- combinaValores(list(keys,
HEADINGValues))
    ##
    ## data.frame construction, from the values
combinations list
    e1 <- pxCubeData[[1]]
    pxCubeData <- do.call("cbind.data.frame",
                          lapply(1:length(e1),function(i
) as.character(proyect(pxCubeData,i))))
    colnames(pxCubeData) <- pxCubeVariable$
variableName
    ##
    ##
    data <- as.vector(sapply(1:numreg,function(i)
keysdata[((i-1)*numFields+1+pxCube$StubLength):((i
-1)*numFields+pxCube$StubLength+length(
HEADINGValues))],simplify = TRUE))

    if( length(data) != nrow(pxCubeData) ) stop("
Error: variables and data inconsistency"," ",length(data),
 " ",nrow(pxCubeData))

    pxCubeData$data <- as.numeric(data)
    names(pxCubeData) <- c(STUB$value,HEADING$
value,"data")
```

```
    rownames(pxCubeData) <- 1:nrow(pxCubeData)

  }
  ##
  return(list(pxCube=pxCube,pxCubeAttrN=
 pxCubeAttrN,pxCubeVariable=pxCubeVariable,
 pxCubeVariableDomain=pxCubeVariableDomain,
 pxCubeData=pxCubeData))

}
```

3.4 CSV file creation from PC-AXIS cube

3.4.1 man

⟨*pcAxisCubeToCSV.Rd*⟩≡

```
\name{pcAxisCubeToCSV}
\alias{pcAxisCubeToCSV}
\title{
  Exports a PC-AXIS cube into CSV in several files.
}
\description{
  It generates four csv files, plus four more depending on "
  keyword" parameters in PC-AXIS file.
}
\usage{
pcAxisCubeToCSV(prefix,pcAxisCube)
}
\arguments{
\item{prefix}{prefix for files to be created}
\item{pcAxisCube}{PC-AXIS cube}
}
\details{
  Created files names are:
\itemize{
\item{prefix+"pxCube.csv"}{}
\item{prefix+"pxCubeVariable.csv"}{}
\item{prefix+"pxCubeVariableDomain.csv"}{}
\item{prefix+"pxCubeData.csv"}{}
\item{prefix+"pxCube"+name+".csv"}{ With name = A0,
  A1,A2 ...}
}
}
\value{
NULL
}
\examples{

  name <- system.file("extdata","datInSFexample6__1.px",
   package = "< < PACKAGE > >")
  stream <- streamParserFromFileName(name,encoding="
  UTF-8")
  cstream <- pcAxisParser(stream)
  if ( cstream$status == 'ok' ) {
    cube <- pcAxisCubeMake(cstream)

    pcAxisCubeToCSV(prefix="datInSFexample6__1",
  pcAxisCube=cube)

  }
```

}
\keyword{PC−AXIS}

3.4.2 code

⟨*pcAxisCubeToCSV.R*⟩≡

```
pcAxisCubeToCSV <− function(prefix,pcAxisCube) {

  write.csv(pcAxisCube$pxCube , file = paste(prefix,"
  pxCube.csv" ,sep=""),row.names = FALSE)

  write.csv(pcAxisCube$pxCubeVariable , file = paste(
  prefix,"pxCubeVariable.csv" ,sep=""),row.names =
  FALSE)

  write.csv(pcAxisCube$pxCubeVariableDomain, file =
  paste(prefix,"pxCubeVariableDomain.csv",sep=""),row.
  names = FALSE)

  write.csv(pcAxisCube$pxCubeData , file = paste(prefix,
  "pxCubeData.csv ",sep=""),row.names = FALSE)

  for( name in names(pcAxisCube$pxCubeAttrN) )
    write.csv(pcAxisCube$pxCubeAttrN[[name]], file =
  paste(prefix,"pxCube",name,".csv ",sep=""),row.names
  = FALSE)
}
```

3.5 Combined tests for functions handling PC-AXIS files

⟨*runit.03PCAXIS.RUnit*⟩≡

```
<<testFuncionesUtiles>>
  <<runit.PCAXIS>>

test.filesPcAxis01 <- function() {

  auxfun00 <- function(stream) {
    cstream <- pcAxisParser(stream)
    streamParserClose(cstream$stream)
    print(names(cstream))
    checkEquals("ok",cstream$status,name)

    if( cstream$status != "ok" ) {
      printCStream(cstream)
    } else {
      cube <- pcAxisCubeMake(cstream)
      print(names(cube))
    }
  }

  auxfun01 <- function(pcaxisFiles,encoding= getOption
("encoding")) {
    for ( name in pcaxisFiles) {

      print("");print("");
      print(paste("File:",name))

      name <- system.file("extdata", name, package = "
<<PACKAGE>>")

      stream <- streamParserFromFileName(name,
  encoding=encoding)

  # print(stream)
      auxfun00(stream)
    }
  }

  auxfun02 <- function(pcaxisFiles,encoding) {
    for ( name in pcaxisFiles) {

      print("");print("");
      print(paste("File:",name))

      name <- system.file("extdata", name, package = "
```

```
  < < PACKAGE > >")

      stream <- streamParserFromString(iconv(readLines(
  name,encoding),"UTF-8"))
  # print(stream)
      auxfun00(stream)
    }
  }

  ## from sample files:
  pcaxisFilesExamples <- list(
                          "datInSFexample6_1.px", "
  datInSFexample6_2.px" ,
                          "datInSFexample6_3.px", "
  datInSFexample6_4.px" ,
                          "datInSFexampleA_5.px", "
  datInSFexample6_5.px" )
  auxfun01(pcaxisFilesExamples,"UTF-8")



} # end function
```

Chapter 4

Package creation

4.1 Instructions

Instructions for generating the literate documentation and the R package.

⟨*README*⟩≡

In order to generate package and the pdf file associated to this document:

0. Required software
noweb. http://www.cs.tufts.edu/~nr/noweb/ http://en.wikipedia.org/wiki/Noweb
"notangle" is used and in makefile variable ("MARKUP"), it is identified the "markup" program associated to noweb.

bash. http://www.gnu.org/s/bash/ http://en.wikipedia.org/wiki/Bash_%28Unix_shell%29

xelatex http://www.tug.org/xetex/ http://en.wikipedia.org/wiki/XeTeX

ps2pdf http://pages.cs.wisc.edu/~ghost/ http://en.wikipedia.org/wiki/Ghostscript

make http://www.gnu.org/software/make/ http://en.wikipedia.org/wiki/Make_%28software%29

sed http://www.gnu.org/s/sed/ http://en.wikipedia.org/wiki/Sed

awk http://www.gnu.org/s/gawk/ http://en.wikipedia.org/wiki/Awk

1. Create a working directory and copy these files:
<< NAMEPKGLP >>
<< PACKAGE >>−vignette.Rnw

2. Execute:
notangle −t8 << NAMEPKGLP >>> Makefile ; make

The following directories are created:
<< PACKAGE >> : package source
<< PACKAGE >>.Rcheck : directory created by "R CMD check"

Moreover, these files are also created:
< < PACKAGE > >_< < VERSION > >.tar.gz :
package file
< < PACKAGE > >.pdf : pdf version of this document

4.2 Makefile

The following objectives are available, within Makefile:

* all: generates literate documentation, recreates package directory structure (along with its content), checks the package and creates tar file for its installation. Default option.
* doc: generates only literate documentation pdf file.
* vignette: generates vignette and R associate, during development phase.
* pkgbuild: recreates packages directory structure (along with its content), checks the package and creates tar file for its installation.
  Package is checked twice: the first time, it is checked the package directory. It generates a warning and the vignette file. The second time, tar.gz is checked, it should generate no warning in order to upload the package to CRAN.
* pkgcheck: recreates package directory structure (along with its content) and checks the package.
* pkg: recreates package directory structure (along with its content).

Package configuration variables:
⟨ *⟩≡

```
#
SHELL:=/bin/bash
#
PKGDIR=./< < PACKAGE > >
#
RDIR=$(PKGDIR)/R
#
MANDIR=$(PKGDIR)/man
#
TESTDIR=$(PKGDIR)/tests
RUNITDIR=$(PKGDIR)/tests/runit
#
INSTDOCDIR=$(PKGDIR)/inst/doc
INSTDATDIR=$(PKGDIR)/inst/extdata
#
NOWEB=noweb
INSTNWDIR=$(PKGDIR)/inst/$(NOWEB)
#
#
LIBR=-l /usrs/lib/R/x86_64-pc-linux-gnu-library/
  2.13
#
NAMEPKG=< < PACKAGE > >
#
NAMEPKGLP=< < NAMEPKGLP > >
#
NAMEVIGNETTE=< < PACKAGE > >-vignette
#
```

Variables for noweb, so we can achieve the desired presentation in latex:

⟨ *⟩+≡

```
# Program markup de noweb
MARKUP=/usr/lib/noweb/markup
#
# Modified back-end totex
TOTEX=./totex
```

The following instruction generates automatically the names for functions to be processed as R files and their Rd associated documentation, etc. Depending on the chunk sufix, its destination directory varies.

Source code for functions in R package	.R	$(RDIR)
Package functions documentation	.Rd	$(MANDIR)
R functions for testing the package	.tstR	$(TESTDIR)
Main function in R, for tests control	.RUnit	$(RUNITDIR)
Data file for tests	.DUnit	$(INSTDATDIR)

⟨*⟩+≡

```
#
RFUN=$(shell sed -rn -e 's/^[<][<](.*)[.]R[>][>]=/\1/p'
  $(NAMEPKGLP))
RMAN=$(shell sed -rn -e 's/^[<][<](.*)[.]Rd[>][>]=/\1/
  p' $(NAMEPKGLP))
TEST=$(shell sed -rn -e 's/^[<][<](.*)[.]tstR[>][>]=/\1/
  p' $(NAMEPKGLP))
RUNI=$(shell sed -rn -e 's/^[<][<](.*)[.]RUnit[>][>]=/\1
  /p' $(NAMEPKGLP))
DUNI=$(shell sed -rn -e 's/^[<][<](.*)[.]DUnit[>][>]=/\1
  /p' $(NAMEPKGLP))
#
```

Objectives available when executing make.

⟨*⟩+≡

```
all: pkgcheck

doc: $(NAMEPKG).pdf

pkgcheck: pkgbuild check

pkgbuild: pkgprecheck build

pkgprecheck: pkg precheck

pkg: $(RFUN:%=$(RDIR)/%.R) \
        $(RMAN:%=$(MANDIR)/%.Rd) \
        $(MANDIR)/$(NAMEPKG)-package.Rd \
        $(TEST:%=$(TESTDIR)/%.R) \
        $(RUNI:%=$(RUNITDIR)/%.R) \
        $(DUNI:%=$(INSTDATDIR)/%) \
        $(PKGDIR)/DESCRIPTION \
        $(PKGDIR)/NAMESPACE \
        pkgdoc \
        pkgvignette

pkgdoc: $(INSTDOCDIR)/$(NAMEPKG).pdf \
        $(INSTNWDIR)/README \
        $(INSTNWDIR)/$(NAMEPKGLP)

pkgvignette: $(INSTDOCDIR)/$(NAMEVIGNETTE).Rnw


INSTALL:
        R CMD INSTALL $(LIBR) --html << PACKAGE >>_<< VERSION >>.tar.gz

REMOVE:
        R CMD REMOVE $(LIBR) << PACKAGE >>

# literate documentation
# installation pdf in package
$(INSTDOCDIR)/$(NAMEPKG).pdf: $(NAMEPKG).pdf | $(INSTDOCDIR)
        cp $(NAMEPKG).pdf $(INSTDOCDIR)/$(NAMEPKG).pdf

# literate documentation
# pdf generation
```

```
$(NAMEPKG).pdf: $(NAMEPKG).tex
	xelatex $(NAMEPKG).tex
	xelatex $(NAMEPKG).tex
	ps2pdf -dAutoRotatePages=/None $(NAMEPKG).
  pdf
	mv qmrparser.pdf.pdf qmrparser.pdf

# literate documentation
# tex generation
$(NAMEPKG).tex: $(NAMEPKGLP) $(TOTEX)
	$(MARKUP) $(NAMEPKGLP) | $(TOTEX) -
  delay | cpif $(NAMEPKG).tex
# noweave -delay $(NAMEPKGLP) > $(NAMEPKG).tex

# Required, so latex/pdf documentation can be generated
$(TOTEX): $(NAMEPKGLP)
	notangle -R`basename $@` $(NAMEPKGLP) | cpif
  $@
	chmod +x $(TOTEX)

# For developing phase, vignette generation
vignette: $(NAMEVIGNETTE).pdf \
	  $(NAMEVIGNETTE).R

# literate documentation
# nw installation in package
# NOWEB PKG
$(INSTNWDIR)/$(NAMEPKGLP): $(NAMEPKGLP) | $(
  INSTNWDIR)
	cp $(NAMEPKGLP) $@


# VIGNETTE PKG doc
$(INSTDOCDIR)/$(NAMEVIGNETTE).Rnw: $(
  NAMEVIGNETTE).Rnw | $(INSTDOCDIR)
	cat $(NAMEVIGNETTE).Rnw | cpif $@

# Package creation
# README
$(INSTNWDIR)/README: $(NAMEPKGLP) | $(
  INSTNWDIR)
	notangle -R`basename $@` $(NAMEPKGLP) | cpif
  $@
```

```
# DESCRIPTION PKG
$(PKGDIR)/DESCRIPTION: $(NAMEPKGLP) | $(
  PKGDIR)
        notangle -R‘basename $@‘ $(NAMEPKGLP) | cpif
  $@

# NAMESPACE PKG
$(PKGDIR)/NAMESPACE: $(NAMEPKGLP) | $(
  PKGDIR)
        notangle -R‘basename $@‘ $(NAMEPKGLP) | cpif
  $@

# R PKG
$(RDIR)/%.R: $(NAMEPKGLP) | $(RDIR)
        cat <( echo "#do not edit, edit $(NOWEB)/$(
  NAMEPKGLP)") \
              <( notangle -R‘basename $@‘ $(
  NAMEPKGLP) ) | cpif $@
#
# man PKG
$(MANDIR)/%.Rd: $(NAMEPKGLP) | $(MANDIR)
        cat <( echo "%do not edit, edit $(NOWEB)/$(
  NAMEPKGLP)") \
              <( notangle -R‘basename $@‘ $(
  NAMEPKGLP) ) | cpif $@


# VIGNETTE for development phase, in local directory
$(NAMEVIGNETTE).pdf: $(NAMEVIGNETTE).tex
        pdflatex $(NAMEVIGNETTE).tex
        pdflatex $(NAMEVIGNETTE).tex

$(NAMEVIGNETTE).tex: $(NAMEVIGNETTE).Rnw
        R CMD Sweave $(NAMEVIGNETTE).Rnw

$(NAMEVIGNETTE).R: $(NAMEVIGNETTE).Rnw
        R CMD Stangle $(NAMEVIGNETTE).Rnw
```

Data files, for examples and tests

⟨ *⟩+≡

```
$(INSTDATDIR)/%: $(NAMEPKGLP) | $(INSTDATDIR)
        notangle -R`basename $@ `.DUnit $(NAMEPKGLP
  ) | cpif $@
```

Rules to create R test code, in test and tests/runit directories. As directories are nested, rule order is relevant.

⟨*⟩+≡

```
$(RUNITDIR)/%.R: $(NAMEPKGLP) | $(RUNITDIR)
        notangle -R`basename $@ R`RUnit $(
  NAMEPKGLP) | cpif $@

$(TESTDIR)/%.R: $(NAMEPKGLP) | $(TESTDIR)
        notangle -R`basename $@ R`tstR $(NAMEPKGLP
  ) | cpif $@
```

Rules to check and construct package

⟨*⟩+≡

```
# checking package source directory
precheck: $(PKGDIR)
        R CMD check $(PKGDIR)

# tar.gz creation
build: <<PACKAGE>>_<<VERSION>>.tar.gz

<<PACKAGE>>_<<VERSION>>.tar.gz: $(
  PKGDIR) precheck
        R CMD build $(PKGDIR)

# checking tar.gz
check: <<PACKAGE>>_<<VERSION>>.tar.gz
        R CMD check <<PACKAGE>>_<<
  VERSION>>.tar.gz
```

Rules to create source package directories

⟨*⟩+≡

```
# package directory
$(PKGDIR):
        mkdir -p $(PKGDIR)

# R directory
$(RDIR):
        mkdir -p $(RDIR)

# man directory
$(MANDIR):
        mkdir -p $(MANDIR)

# inst/doc directory. In installed package, doc directory (
  with no inst)
$(INSTDOCDIR):
        mkdir -p $(INSTDOCDIR)

# inst/extdata directory. In installed package, extdata
  directory (with no inst)
$(INSTDATDIR):
        mkdir -p $(INSTDATDIR)

# inst/noweb directory. In installed package, noweb
  directory (with no inst)
$(INSTNWDIR):
        mkdir -p $(INSTNWDIR)

# test directory, not present in installed package
$(TESTDIR):
        mkdir -p $(TESTDIR)

# tests/runit directory, not present in installed package
$(RUNITDIR):
        mkdir -p $(RUNITDIR)
```

If no tabulators are inserted or -t8 is used, make errors are generated:

```
Makefile:: *** missing separator (did you mean TAB instead
   of 8 spaces?). Stop.
Makefile:: *** missing separator. Stop.
```

Chapter 5

Data files used

5.1 Manual test files

5.1.1 datInTest01.txt

Files for testing recognising tokens and combination rules functions.

⟨*datInTest01.txt.DUnit*⟩≡

```
"aaa4567"
−121.01e−222

121
1211e33
−123e−222
+127
0123.123
1233.
.123E22
+0123.123e+11
−1233.e−66
+.123E600
abd
"aaas45"
```

5.2 PC-AXIS test functions

Test files obtained:
PC-Axis file format manual. Statistics of Finland.
`http://tilastokeskus.fi/tup/pcaxis/tiedostomuoto2006_laaja_en.pdf`

5.2.1 datInSFexample6__1.px

⟨*datInSFexample6_1.px.DUnit*⟩≡

```
CHARSET="ANSI";
AXIS-VERSION="2000";
DECIMALS=0;
MATRIX="vaerak";
SUBJECT-CODE="VRM";
SUBJECT-AREA="Väestö";
TITLE="Väestö 31.12. muuttujina Sukupuoli, Kunta, Vuosi
   ja Siviilisääty";
CONTENTS="Väestö 31.12.";
UNITS="Henkilöä";
STUB="Sukupuoli","Kunta";
HEADING="Vuosi","Siviilisääty";
VALUES("Sukupuoli")="Sukupuoli yhteensä","Miehet","
  Naiset";
VALUES("Kunta")="Espoo","Helsinki","Vantaa";
VALUES("Vuosi")="2001","2002";
VALUES("Siviilisääty")="Siviilisääty yhteensä","Naimaton
  ","Naimisissa";
DATA=
57516 43030 100546 57516 43030 100546
144564 85339 229903 144564 85339 229903
47131 33688 80819 47131 33688 80819
53821 43375 97196 53821 43375 97196
151536 86184 237720 151536 86184 237720
44071 33935 78006 44071 33935 78006
111337 86405 197742 111337 86405 197742
296100 171523 467623 296100 171523 467623
91202 67623 158825 91202 67623 158825;
```

5.2.2 datInSFexample6__2.px

⟨*datInSFexample6_2.px.DUnit*⟩≡

```
CHARSET="ANSI";
AXIS-VERSION="2000";
LANGUAGE="fi";
CREATION-DATE="20001212 11:00";
DECIMALS=0;
SHOWDECIMALS=0;
MATRIX="vaerak";
COPYRIGHT=YES;
SUBJECT-CODE="VRM";
SUBJECT-AREA="Väestö";
DESCRIPTION="Väestö 31.12.2003";
TITLE="Väestö 31.12.2003 muuttujina Sukupuoli, Kunta ja
   Siviilisääty";
CONTENTS="Väestö 31.12.2003";
UNITS="Henkilöä";
STUB="Sukupuoli","Kunta";
HEADING="Siviilisääty";
VALUES("Sukupuoli")="Sukupuoli yhteensä","Miehet","
  Naiset";
VALUES("Kunta")="Espoo","Helsinki","Vantaa";
VALUES("Siviilisääty")="Siviilisääty yhteensä","Naimaton
  ","Naimisissa";
CODES("Sukupuoli")="S","1","2";
CODES("Kunta")="049","091","092";
CODES("Siviilisääty")="S","1","2";
LAST-UPDATED="20040319 09:00";
CONTACT="Tilastokeskus, Väestötilastopalvelu#
  Postiosoite:"
"Väestötilastopalvelu, #PL 4A, 00022 Tilastokeskus#
  Puhelin: (09) 1734 3590"
"#Faksi: (09) 1734 #3251#Yhteyshenkilö: Nicola Brun#
  Sähköposti:"
"#vaestotilasto.palvelu@tilastokeskus.fi#<A HREF=http:/
  /tilastokeskus.fi/ "
"TARGET=_blank>Linkki tilastokeskuksen kotisivulle</A
  >'";
SOURCE="Tilastokeskus";
INFOFILE="vaerak";
NOTE="Vuodesta 2002 lähtien on siviilisäätytietoihin
  lisätty rekisteröidyt"
"parisuhteet...";
DATA=
197742 111337 86405
467623 296100 171523
```

158825 91202 67623
100546 57516 43030
229903 144564 85339
80819 47131 33688
97196 53821 43375
237720 151536 86184
78006 44071 33935;

5.2.3 datInSFexample6_3.px

⟨*datInSFexample6_3.px.DUnit*⟩≡

```
CHARSET="ANSI";
AXIS-VERSION="2000";
LANGUAGE="fi";
CREATION-DATE="19930401 12:10";
DECIMALS=0;
MATRIX="vaerak";
COPYRIGHT=YES;
SUBJECT-CODE="VRM";
SUBJECT-AREA="Väestö";
DESCRIPTION="Väestö 1990-1992";
TITLE="Väestö muuttujina siviilisääty, sukupuoli, alue ja
  aika";
CONTENTS="Väestö";
UNITS="henkilöiden lukumäärä";
STUB="siviilisääty","sukupuoli";
HEADING="alue","aika";
VALUES("siviilisääty")="naimisissa","naimaton","eronnut
  ","leski";
VALUES("sukupuoli")="mies","nainen";
VALUES("alue")="Koko maa","Helsinki","Espoo";
VALUES("aika")="1990","1991","1992";
TIMEVAL("aika")=TLIST(A1),"1990","1991","1992";
CODES("siviilisääty")="1","2","3","4";
CODES("sukupuoli")="1","2";
CODES("alue")="SSS","091","049";
ELIMINATION("alue")="Koko maa";
ELIMINATION("sukupuoli")=YES;
LAST-UPDATED="19950209 13:00";
SEASADJ=NO;
CONTACT="Tilastokeskus#PC-Axis-koulutus#fax 09
  1734 1234#s-posti pcaxis@stat.fi";
DATABASE="TKDB";
SOURCE="Tilastokeskus";
REFPERIOD="Viiteajankohta on 31. joulukuuta joka vuosi
  ";
INFOFILE="vaerak";
NOTE("siviilisääty")="Siviilisääty on riippuvainen
  väestölaskennan "
"rekistereistä. Naimisissa olevat yhdessä asuvat merkitään
  naimisissa "
"oleviksi. Muut yhdessä asuvat kuuluvat naimattomiin";
VALUENOTE("alue","Espoo")="Tähän selitystä Espoon
  datasta#ja selitys "
"jaetaan näytössä usealle riville.#Tästä alkaakin jo kolmas
```

selitysrivi";
DATA=
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42

```
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72

;
```

5.2.4 datInSFexample6_4.px

⟨*datInSFexample6_4.px.DUnit*⟩≡

```
CHARSET="ANSI";
AXIS-VERSION="2000";
LANGUAGE="fi";
DECIMALS=0;
SHOWDECIMALS=0;
MATRIX="tyti";
COPYRIGHT=YES;
SUBJECT-CODE="TYM";
SUBJECT-AREA="Työmarkkinat";
DESCRIPTION="T01E Väestö 31.12.2000";
TITLE="T01E Väestö 31.12.2000 muuttujina Osa-alue,"
"Ikä ja Pääasiallinen toiminta/ammattiasema.";
CONTENTS="T01E Väestö 31.12.2000";
UNITS="Henkilö";
STUB="Osa-alue","Ikä";
HEADING="Pääasiallinen toiminta/ammattiasema";
VALUES("Osa-alue")="049 Espoo","078 Hanko-Hangö
  ","091 Helsinki",
"092 Vantaa","106 Hyvinkää";
VALUES("Ikä")="0-6","7-14","30-34";
VALUES("Pääasiallinen toiminta/ammattiasema")="
  Työvoima",
"Työlliset","Palkansaajat";
CODES("Osa-alue")="049","078","091","092","106";
DOMAIN("Osa-alue")="OSAL_01 2002";
INFOFILE="tyti";
KEYS("Osa-alue")=CODES;
KEYS("Ikä")=VALUES;
DATA=
"049","30-34",16064 15324 14660
"078","30-34",560 490 470
"091","30-34",42100 39027 37423
"092","30-34",13215 12358 11806
"106","30-34",2646 2423 2270;
```

5.2.5 datInSFexampleA_5.px

⟨*datInSFexampleA_5.px.DUnit*⟩≡

CHARSET=”ANSI”;
AXIS−VERSION=”2005”;
LANGUAGE=”en”;
LANGUAGES=”en”,”sv”,”fi”;
CREATION−DATE=”20050217 18:34”;
DECIMALS=0;
SHOWDECIMALS=0;
MATRIX=”BE0101F1”;
SUBJECT−CODE=”BE”;
SUBJECT−AREA=”Population”;
SUBJECT−AREA[sv]=”Befolkning”;
SUBJECT−AREA[fi]=”Väestö”;
DESCRIPTION=”Migration 2003”;
DESCRIPTION[sv]=”Flyttningar 2003”;
DESCRIPTION[fi]=”Väestönmuutokset 2003”;
TITLE=”Migration by region, age, period, type and sex”;
TITLE[sv]=”Flyttningar efter region, ålder, tid, typ och kön
”;
TITLE[fi]=”Väestönmuutokset muuttujina kunta, ikä, aika,
tyyppi ja sukupuoli”;
CONTENTS=”Migration”;
CONTENTS[sv]=”Flyttningar”;
CONTENTS[fi]=”Väestönmuutokset”;
UNITS=”number”;
UNITS[sv]=”antal”;
UNITS[fi]=”henkilöä”;
STUB=”region”,”age”;
STUB[sv]=”region”,”ålder”;
STUB[fi]=”kunta”,”ikä”;
HEADING=”period”,”type”,”sex”;
HEADING[sv]=”tid”,”typ”,”kön”;
HEADING[fi]=”aika”,”tyyppi”,”sukupuoli”;
CONTVARIABLE=”type”;
CONTVARIABLE[sv]=”typ”;
CONTVARIABLE[fi]=”tyyppi”;
VALUES(”region”)=”Sweden”,”Stockholm county”,”
Upplands Väsby”,”Vallentuna”;
VALUES[sv](”region”)=”Riket”,”Stockholms län”,”Upplands
Väsby”,”Vallentuna”;
VALUES[fi](”kunta”)=”Ruotsi”,”Tukholman lääni”,”
Upplands Väsby”,”Vallentuna”;
VALUES(”age”)=”20”,”21”,”22”;
VALUES[sv](”ålder”)=”20”,”21”,”22”;
VALUES[fi](”ikä”)=”20”,”21”,”22”;

VALUES(”period”)=”2003”;
VALUES[sv](”tid”)=”2003”;
VALUES[fi](”aika”)=”2003”;
VALUES(”type”)=”Inmigrated”,”Outmigrated”;
VALUES[sv](”typ”)=”Inflyttade”,”Utflyttade”;
VALUES[fi](”tyyppi”)=”Tulomuutto”,”Lähtömuutto”;
VALUES(”sex”)=”men”,”women”;
VALUES[sv](”kön”)=”män”,”kvinnor”;
VALUES[fi](”sukupuoli”)=”miehet”,”naiset”;
CODES(”region”)=”00”,”01”,”0114”,”0115”;
CODES[sv](”region”)=”00”,”01”,”0114”,”0115”;
CODES[fi](”kunta”)=”00”,”01”,”0114”,”0115”;
CODES(”age”)=”20”,”21”,”22”;
CODES[sv](”ålder”)=”20”,”21”,”22”;
CODES[fi](”ikä”)=”20”,”21”,”22”;
CODES(”type”)=”BE0101F1”,”BE0101F2”;
CODES[sv](”typ”)=”BE0101F1”,”BE0101F2”;
CODES[fi](”tyyppi”)=”BE0101F1”,”BE0101F2”;
CODES(”sex”)=”1”,”2”;
CODES[sv](”kön”)=”1”,”2”;
CODES[fi](”sukupuoli”)=”1”,”2”;
DOMAIN(”age”)=”Age”;
DOMAIN[sv](”ålder”)=”Ålder”;
DOMAIN[fi](”ikä”)=”ikä”;
ELIMINATION(”region”)=”Sweden”;
ELIMINATION[sv](”region”)=”Riket”;
ELIMINATION[fi](”kunta”)=”Ruotsi”;
LAST−UPDATED(”Inmigrated”)=”20040212 16:28”;
LAST−UPDATED[sv](”Inflyttade”)=”20040212 16:28”;
LAST−UPDATED[fi](”Tulomuutto”)=”20040212 16:28”;
UNITS(”Inmigrated”)=”number”;
UNITS[sv](”Inflyttade”)=”antal”;
UNITS[fi](”Tulomuutto”)=”henkilö”;
UNITS(”Outmigrated”)=”number”;
UNITS[sv](”Utflyttade”)=”antal”;
UNITS[fi](”Lähtömuutto”)=”henkilö”;
SOURCE=”Statistics Sweden”;
SOURCE[sv]=”Statistiska centralbyrån (SCB)”;
SOURCE[fi]=”SCB”;
CONTACT(”Inmigrated”)=”Ewa Eriksson, SCB#Tel: +46 19−17 67 43”;
CONTACT[sv](”Inflyttade”)=”Ewa Eriksson, SCB#Tel: 019−17 67 43”;

```
CONTACT[fi](”Tulomuutto”)=”Ewa Eriksson, SCB#Tel:
  019−17 67 43”;
DATABASE=”SDB”;
DATABASE[sv]=”SDB”;
DATABASE[fi]=”SDB”;
INFOFILE=”BE0101en”;
INFOFILE[sv]=”BE0101sv”;
INFOFILE[fi]=”BE0101fi”;
NOTEX=”Mandatory English note”;
NOTEX[sv]=”Obligatorisk svensk not”;
NOTEX[fi]=”Pakollinen suomalainen alaviite”;
NOTEX(”region”)=”Mandatory footnote for region”;
NOTEX[sv](”region”)=”Obligatorisk fotnot för region”;
NOTEX[fi](”kunta”)=”Pakollinen suomalainen alaviite
  muuttujalle kunta”;
NOTE(”age”)=”Age refers to age attained by the end of the
   year, ”
”i.e. in principal, an account for the year of birth.”;
NOTE[sv](”ålder”)=”Med ålder avses uppnådd ålder vid
  årets slut, ”
”d.v.s. i princip en redovisning efter födelseår.”;
NOTE[fi](”ikä”)=”Sama suomeksi”;
VALUENOTEX(”type”,”Inmigrated”)=”Migration cannot
  be added with regard to ”
”regions. When, for instance, municipalities are added to
  metropolitan areas, ”
”migration to the region is added to migrations within the
  region.#With ”
”calcutating of age−specific rates per 1 000 of the mean
  population you shall ”
”use the mean population per year of birth. And some more
  ”;
VALUENOTEX[sv](”typ”,”Inflyttade”)=”Flyttningar kan
  inte summeras vad avser ”
”region. Vid t ex summering av kommuner till
  storstadsområde, summeras ”
”flyttningar till regionen med flyttningar inom regionen.#
  Vid beräkning av ”
”åldersdifferentierade tal per 1 000 av medelfolkmängden
  skall medelfolkmängden ”
”per födelseår användas.”;
VALUENOTEX[fi](”tyyppi”,”Tulomuutto”)=”
  Väestönmuutoksia ei saa laskea yhteen”;
VALUENOTE(”type”,”Inmigrated”)=”Request value note
```

for inmigrated";
VALUENOTE[sv]("typ","Inflyttade")="Frivillig fotnot för inflyttade.";
VALUENOTE[fi]("tyyppi","Tulomuutto")="Suomalainen alaviite, tulomuutto.";
CELLNOTE("*","20","*","Inmigrated","*")="Cellnote inmigrated 20 years";
CELLNOTE[sv]("*","20","*","Inflyttade","*")="Cellnot inflyttade 20 år.";
CELLNOTE[fi]("*","20","*","Tulomuutto","*")="Soluviite, tulomuutto, 20.";
DATA=
456 938 223 327
613 948 305 464
835 968 325 511
590 1017 480 741
771 1145 668 993
924 1273 672 906
14 38 22 43
21 36 34 57
26 36 34 34
17 20 22 30
13 21 29 42
20 23 33 35;

5.2.6 datInSFexample6_5.px

⟨*datInSFexample6_5.px.DUnit*⟩≡

```
CHARSET="ANSI";
AXIS-VERSION="2000";
LANGUAGE="sv";
CREATION-DATE="20030425 18:21";
DECIMALS=0;
SHOWDECIMALS=0;
MATRIX="BE0101F1";
SUBJECT-CODE="BE";
SUBJECT-AREA="Befolkning";
DESCRIPTION="Flyttningar 2000";
TITLE="Flyttningar med region, ålder, kön, tid och typ";
CONTENTS="Flyttningar";
UNITS="antal";
STUB="region","ålder","kön";
HEADING="tid","typ";
CONTVARIABLE="typ";
VALUES("region")="00 Riket";
VALUES("ålder")="2";
VALUES("kön")="män","kvinnor";
VALUES("tid")="2000";
VALUES("typ")="Inflyttade","Flyttningsöverskott","in3";
TIMEVAL("tid")=TLIST(A1),"2000";
CODES("region")="00";
CODES("ålder")="2";
CODES("kön")="1","2";
CODES("typ")="typ1","typ2","in3";
DOMAIN("ålder")="Ålder";
LAST-UPDATED("Inflyttade")="20030212 16:49";
LAST-UPDATED("Flyttningsöverskott")="20030212
  16:49";
LAST-UPDATED("in3")="20030212 16:49";
UNITS("Inflyttade")="antal";
UNITS("Flyttningsöverskott")="antal";
UNITS("in3")="ton";
CONTACT("Inflyttade")="Ewa Eriksson, SCB#Tel:
  019-17 67 43";
REFPERIOD("in3")="2000";
DATABASE="Sveriges Statistiska Databaser";
SOURCE="Statistiska centralbyrån (SCB)";
INFOFILE="BE0101";
NOTE="Det här är också bra att veta";
DATA=
0 0 0
```

0 0 0;

Chapter 6

Appendix

6.1 Literate programming

Package has been developed using literate programming (`http://en.wikipedia.org/wiki/Literate_programming`) and noweb programm (`http://www.cs.tufts.edu/~nr/noweb/` and `http://en.wikipedia.org/wiki/Noweb`).

In order to format in a nice way code coming from different languages (R, Rd, make, etc.), listing latex package has been used (`http://www.ctan.org/tex-archive/macros/latex/contrib/listings/` and `http://en.wikibooks.org/wiki/LaTeX/Packages/Listings`). Therefore, totex noweb program must be slightly modified (modification is included next).

Listing package solves 'Overfull
hbox' massive problem, allowing carriage return control control.

6.1.1 totex

⟨*totex*⟩≡

```
#!/bin/sh
# Copyright 1991 by Norman Ramsey. All rights reserved.
# See file COPYRIGHT for more information.
# Don't try to understand this file! Look at lib/totex.nw in
  the noweb source!
delay=0 noindex=0
for i do
  case $i in
    -delay) delay=1 ;;
    -noindex) noindex=1 ;;
    *) echo "This can't happen -- $i passed to totex" 1>
  &2 ; exit 1 ;;
  esac
done
nawk 'BEGIN { code=0 ; quoting=0 ; text=1; defns[0] = 0
                                                    ulist
  [0] = 0 }
      #/^@begin code/ { code=1 ; printf "\\nwbegincode{%
  s}", substr($0, 13) }
      #/^@end code/ { code=0 ; printf "\\nwendcode{}";
  lastdefnlabel = "" }
      /^@begin code/ { code=1 ; printf "\\nwbegincode{%
  s} ", substr($0, 13) }
      /^@end code/ { code=0 ; printf "\\end{lstlisting}";
  lastdefnlabel = "" }
      /^@begin docs 0$/ { if (delay) next }
      /^@end docs 0$/ { if (delay) {
                          printf "\\nwfilename{%s}",
  filename; delay=0; next
                        } }
      /^@begin docs/ { text=0 ; printf "\\nwbegindocs{%
  s}", substr($0, 13) }
      /^@end docs/ { printf "\\nwenddocs{}" }
      /^@text / { line = substr($0, 7) ; text += length -
   6
                  if (code) printf "%s", line ##
  escape_brace_bslash(line)
                  else if (quoting) printf "%s",
  TeXliteral(line)
                  else printf "%s", line
                }
      /^@nl$/ { if (!code) {if (text==0) printf "\\
  nwdocspar"
                                        text=1}
```

```
                        if (quoting) printf "\\nwnewline
"
                        printf "\n"
                    }
    /^@defn / { name = substr($0, 7); if (lastxreflabel !
= "") {
                                                printf "
\\sublabel{%s}", lastxreflabel
                                                printf "
\\nwmargintag{%s}", label2tag(lastxreflabel)
                                            }
                                            printf "\\
moddef{%s%s}\\%sendmoddef\\nwendcode{}\\begin{
lstlisting}", escape__brace__bslash( convquotes(name)) , (
lastxrefref != "" ? ("~" label2tag(lastxrefref)) : ""),defns[
name]
                                            lastdefnlabel
 = lastxreflabel
                                            lastxreflabel
 = lastxrefref = ""
                                            defns[
name] = "plus" }
    /^@use / { printf "< < %s%s > >", ##printf "\\
LA{}%s%s\\RA{}",
                            convquotes(substr($0, 6)), (
lastxrefref != "" ? ("~" label2tag(lastxrefref)) : "")
                    }
    /^@quote$/ { quoting = 1 ; printf "{\\tt{}" }
    /^@endquote$/ { quoting = 0 ; printf "}" }
    /^@file / { filename = substr($0, 7); lastxreflabel =
lastxrefref = ""
                        if (!delay) printf "\\nwfilename
{%s}", filename
                    }
    /^@literal / { printf "%s", substr($0, 10) }
    /^@header latex / { printf "\\documentclass{article
}\\usepackage{noweb}\\pagestyle{noweb}\\noweboptions
{%s}%s",
                            substr($0, 15), "\\begin
{document}" }
    /^@header tex / { printf "\\input nwmac " }
    /^@trailer latex$/ { print "\\end{document}" }
    /^@trailer tex$/ { print "\\bye" }
    /^@xref label / { lastxreflabel = substr($0, 13) }
```

```
    /^@xref ref / { lastxrefref = substr($0, 11) }
    /^@xref begindefs$/ { printf "\\nwalsodefined{" }
    /^@xref defitem / { printf "\\\\{%s}", substr($0,
15) }
    /^@xref enddefs$/ { printf "}" }
    /^@xref beginuses$/ { printf "\\nwused{" }
    /^@xref useitem / { printf "\\\\{%s}", substr($0,
15) }
    /^@xref enduses$/ { printf "}" }
    /^@xref notused / { printf "\\nwnotused{%s}",
TeXliteral(substr($0, 15)) }
    /^@xref nextdef / { }
    /^@xref prevdef / { }
    /^@xref beginchunks$/ { }
    /^@xref chunkbegin / { label = $3; name = substr($
0, 19 + length(label))
                                        printf "\\
nwixlogsorted{c}{{%s}{%s}{",
                                          convquotes(
name), label
                                    }
    /^@xref chunkuse / { printf "\\nwixu{%s}", substr(
$0, 16) }
    /^@xref chunkdefn / { printf "\\nwixd{%s}", substr
($0, 17) }
    /^@xref chunkend$/ { print "}}%" }
    /^@xref endchunks$/ { }
    /^@index nl$/ { print (code ? "\\eatline" : "%") }
    /^@index defn / {
            if (!noindex) { arg = substr($0, 13); if (
lastxreflabel != "") printf "\\nosublabel{%s}",
lastxreflabel
                                                  if (
lastxrefref != "")
                                                    printf
 "\\nwindexdefn{%s}{%s}{%s}", TeXliteral(arg),
indexlabel(arg), lastxrefref
                                                  lastxreflabel
 = lastxrefref = "" } }
    /^@index localdefn / {
            if (!noindex) { arg = substr($0, 18); if (
lastxreflabel != "") printf "\\nosublabel{%s}",
lastxreflabel
                                                  if (
```

```
lastxrefref != "")
                                                    printf
"\\nwindexdefn{%s}{%s}{%s}", TeXliteral(arg),
indexlabel(arg), lastxrefref
                                                lastxreflabel
= lastxrefref = "" } }
    /^@index use / {
            if (!noindex) { arg = substr($0, 12); if (!
code) {
                                                    if (
lastxreflabel != "") printf "\\protect\\nosublabel{%s}",
lastxreflabel
                                                    if (
lastxrefref != "")
                                                      printf
"\\protect\\nwindexuse{%s}{%s}{%s}",
                                                              TeXliteral
(arg), indexlabel(arg), lastxrefref
                                                }
                                                lastxreflabel
= lastxrefref = "" } }
    /^@index begindefs$/ { if (!noindex) { printf "\\
nwidentdefs{" } }
    /^@index isused / { if (!noindex) { } } # handled by
latex
    /^@index defitem / { if (!noindex) { i = substr($
0,16); printf "\\\\{{%s}{%s}}", TeXliteral(i), indexlabel(
i) } }
    /^@index enddefs$/ { if (!noindex) { printf "}" } }
    /^@index beginuses$/ { if (!noindex) { printf "\\
nwidentuses{"; ucount = 0 } }
    /^@index isdefined / { if (!noindex) { } } # latex
finds the definitions
    /^@index useitem / { if (!noindex) { i = substr($0,
16); printf "\\\\{{%s}{%s}}", TeXliteral(i), indexlabel(i)
                                        ulist[ucount++] =
i
                                    } }
    /^@index enduses$/ { if (!noindex) { printf "}"; if (
lastdefnlabel != "") {
                                                for
(j = 0; j < ucount; j++)
                                                  printf
"\\nwindexuse{%s}{%s}{%s}",
```

```
(ulist[j]), indexlabel(ulist[j]), lastdefnlabel
                                                    }
} }
    /^@index beginindex$/ { if (!noindex) { } }
    /^@index entrybegin / { if (!noindex) { label = $3;
name = substr($0, 20 + length(label))
                                          printf "\\
nwixlogsorted{i}{{%s}{%s}}%%\n",
                                            TeXliteral(
name), indexlabel(name)
                                        } }
    /^@index entryuse / { if (!noindex) { } } # handled
by latex
    /^@index entrydefn / { if (!noindex) { } } # handled
by latex
    /^@index entryend$/ { if (!noindex) { } }
    /^@index endindex$/ { if (!noindex) { } }

    END { printf "\n" }
    function label2tag(label) {
      return "{\\nwtagstyle{}\\subpageref{" label "}}"
    }
    function escape_brace_bslash(line) {
      gsub(/[\\{}]/, "\n&", line)
      gsub(/\n/, "\\", line)
      gsub(/__/ , "\\__", line)
      return line
    }
    function convquotes(s, r, i) {
      r = ""
      while (i = index(s, "[[")) {
        r = r substr(s, 1, i-1) "\\code{}"
        s = substr(s, i+2)
        if (i = match(s, "\\]\\]+")) {
          r = r TeXliteral(substr(s, 1, i-1+RLENGTH
-2)) "\\edoc{}"
          s = substr(s, i+RLENGTH)
        } else {
          r = r s "\\edoc{}"
          s = ""
        }
      }
      return r s
```

```
}
function indexlabel(ident, l) {
  l = ident
  gsub(/:/, ":col", l) # must be first (colon)
  gsub(/ /, ":sp", l) # space
  gsub(/#/, ":has", l) # hash
  gsub(/\$/, ":do", l) # dollar
  gsub(/%/, ":pe", l) # percent
  gsub(/&/, ":am", l) # ampersand
  gsub(/,/, ":com", l) # commad
  gsub(/\\/, ":bs", l) # backslash
  gsub(/\^/, ":hat", l) # hat
  gsub(/__/, ":un", l) # underscore
  gsub(/{/, ":lb", l) # left brace
  gsub(/}/, ":rb", l) # right brace
  gsub(/~/, ":ti", l) # tilde
  return l
}
function TeXliteral(arg) {
  gsub(/\\/, "<\\char92>", arg)
  gsub(/}/, "<\\char125}", arg)
  gsub(/{/, "{\\char123}", arg)
  gsub(/<\\char/, "{\\char", arg)
  gsub(/{\\char92>/, "{\\char92}", arg)
  gsub(/\$/, "{\\char36}", arg)
  gsub(/&/, "{\\char38}", arg)
  gsub(/#/, "{|char35}", arg)
  gsub(/\^/, "{\\char94}", arg)
  gsub(/__/, "{\\char95}", arg)
  gsub(/%/, "{\\char37}", arg)
  gsub(/~/, "{\\char126}", arg)
  gsub(/ /, "\\ ", arg)
  return arg
}' delay=$delay noindex=$noindex
```

And a copy of COPYRIGHT noweb file, included to meet license requirements.