Package ‘qmrparser’

October 13, 2022

Type Package
Title Parser Combinator in R
Version 0.1.6
Date 2022-04-10
Author Juan Gea Rosat, Ramon Martínez Coscollà.
Maintainer Juan Gea <juangea@geax.net>
Description Basic functions for building parsers, with an application to PC-AXIS format files.
License GPL (>= 3)
Depends R (>= 3.4.0)
Suggests RUnit
LazyLoad yes
Encoding UTF-8
NeedsCompilation no
Repository CRAN
Date/Publication 2022-04-24 00:00:05 UTC

R topics documented:

qmrparser-package ....................................................... 2
alternation ................................................................. 3
charInSetParser ............................................................ 4
charParser ................................................................. 5
commentParser ............................................................ 7
concatenation .............................................................. 8
dots .......................................................................... 9
empty ....................................................................... 10
eofMark ................................................................. 11
isDigit ............................................................... 12
isHex ................................................................. 13
isLetter ............................................................. 14
isLowercase ............................................................ 14
Description

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

Details

Package: qmrparser
Type: Package
Version: 0.1.6
Date: 2022-04-10
License: GPL (>= 3)
LazyLoad: yes

Collection of functions to build programs to read complex data files formats, with an application to the case of PC-AXIS format.
**alternation**

**Author(s)**

Juan Gea Rosat, Ramon Martínez Coscollà
Maintainer: Juan Gea Rosat <juangea@geax.net>

**References**


Type `RShowDoc("index",package="qmrparser")` at the R command line to open the package vignette.
Type `RShowDoc("qmrparser",package="qmrparser")` to open pdf developer guide.
Source code used in literate programming can be found in folder 'noweb'.

---

**alternation**

**Alternative phrases**

**Description**

Applies parsers until one succeeds or all of them fail.

**Usage**

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

**Arguments**

... list of alternative parsers to be executed
action Function to be executed if recognition succeeds. It takes as input parameters information derived from parsers involved as parameters
error Function to be executed if recognition does not succeed. It takes two parameters:
  * p with position where parser, `streamParser`, starts its recognition, obtained with `streamParserPosition`
  * 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 `error` definition in these parsers.
Details

In case of success, action gets the node from the first parse to succeed. In case of failure, parameter h from error gets a list, with information about failure from all the parsers processed.

Value

Anonymous functions, returning a list.

\[
\text{function(stream) \rightarrow list(status, node, stream)}
\]

From these input parameters, an anonymous function is constructed. This function admits just one parameter, stream, with `streamParser` class, and returns a three-field list:

- status
  "ok" or "fail"
- node
  With action or error function output, depending on the case
- stream
  With information about the input, after success or failure in recognition

Examples

```r
# 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")]
```

---

`charInSetParser`  
**Single character, belonging to a given set, token**

Description

Recognises a single character satisfying a predicate function.

Usage

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

**Arguments**

- **fun**: Function to determine if character belongs to a set. Argument "fun" is a signature function: character -> logical (boolean)

- **action**: Function to be executed if recognition succeeds. Character stream making up the token is passed as parameter to this function

- **error**: Function to be executed if recognition does not succeed. Position of `streamParser` obtained with `streamParserPosition` is passed as parameter to this function

**Value**

Anonymous function, returning a list.

```r
function(stream) -> list(status,node,stream)
```

From input parameters, an anonymous function is defined. This function admits just one parameter, stream, with type `streamParser`, and returns a three-field list:

- **status**: "ok" or "fail"
- **node**: With action or error function output, depending on the case
- **stream**: With information about the input, after success or failure in recognition

**Examples**

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

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

---

**charParser**

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

- **char**: character to be recognised
- **action**: Function to be executed if recognition succeeds. Character stream making up the token is passed as parameter to this function
- **error**: Function to be executed if recognition does not succeed. Position of `streamParser` obtained with `streamParserPosition` is passed as parameter to this function

Value

Anonymous function, returning a list.

```
function(stream) -> list(status,node,stream)
```

From input parameters, an anonymous function is defined. This function admits just one parameter, `stream`, with type `streamParser`, and returns a three-field list:

- **status**: "ok" or "fail"
- **node**: With action or error function output, depending on the case
- **stream**: With information about the input, after success or failure in recognition

See Also

- `keyword`

Examples

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

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

# ok
( charParser("\U0006") (streamParserFromString("\U0006")))[c("status","node")]
```


**Description**

Recognises a comment, a piece of text delimited by two predefined tokens.

**Usage**

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

**Arguments**

- **beginComment**: String indicating comment beginning
- **endComment**: String indicating comment end
- **action**: Function to be executed if recognition succeeds. Character stream making up the token is passed as parameter to this function
- **error**: Function to be executed if recognition does not succeed. Position of streamParser obtained with `streamParserPosition` is passed as parameter to this function

**Details**

Characters preceded by \ are not considered as part of beginning of comment end.

**Value**

Anonymous function, returning a list.

```r
function(stream) -> list(status, node, stream)
```

From input parameters, an anonymous function is defined. This function admits just one parameter, stream, with type `streamParser`, and returns a three-field list:

- **status**: "ok" or "fail"
- **node**: With action or error function output, depending on the case
- **stream**: With information about the input, after success or failure in recognition
concatenation

Examples

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

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

---

concatenation  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

...  list of parsers to be executed
action  Function to be executed if recognition succeeds. It takes as input parameters information derived from parsers involved as parameters
error  Function to be executed if recognition does not succeed. I takes two parameters:
    • p with position where parser, streamParser, starts its recognition, obtained with streamParserPosition
    • 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 error definition in these parsers.

Details

In case of success, parameter s from action gets a list with information about node from all parsers processed.
In case of failure, parameter h from error gets the value returned by the failing parser.
Value
Anonymous functions, returning a list.

\[
\text{function(stream) \rightarrow list(status, node, stream)}
\]

From these input parameters, an anonymous function is constructed. This function admits just one parameter, stream, with \texttt{streamParser} class, and returns a three-field list:

- **status**
  "ok" or "fail"
- **node**
  With action or error function output, depending on the case
- **stream**
  With information about the input, after success or failure in recognition

Examples

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

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

---

**dots**

\textit{Dots sequence token.}

Description
Recognises a sequence of an arbitrary number of dots.

Usage

\[
dots(action = \text{function(s) list(type}="dots","value=s)
\]

```r
e error = \text{function(p) list(type}="dots","pos =p))
```

Arguments

- **action**
  Function to be executed if recognition succeeds. Character stream making up the token is passed as parameter to this function
- **error**
  Function to be executed if recognition does not succeed. Position of \texttt{streamParser} obtained with \texttt{streamParserPosition} is passed as parameter to this function
Anonymous function, returning a list.

```r
function(stream) -> list(status,node,stream)
```

From input parameters, an anonymous function is defined. This function admits just one parameter, `stream`, with type `streamParser`, and returns a three-field list:

- **status**
  - "ok" or "fail"
- **node**
  - With action or error function output, depending on the case
- **stream**
  - With information about the input, after success or failure in recognition

### Examples

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

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

---

**empty**

*Empty token*

#### Description

Recognises a null token. This parser always succeeds.

#### Usage

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

#### Arguments

- **action**
  - Function to be executed if recognition succeeds. Character stream making up the token is passed as parameter to this function
- **error**
  - Function to be executed if recognition does not succeed. Position of `streamParser` obtained with `streamParserPosition` is passed as parameter to this function
eofMark

Details

action s parameter is always "". Error parameters exists for the sake of homogeneity with the rest of functions. It is not used.

Value

Anonymous function, returning a list.

function(stream) → list(status,node,stream)

From input parameters, an anonymous function is defined. This function admits just one parameter, stream, with type `streamParser`, and returns a three-field list:

- status
  "ok" or "fail"
- node
  With action or error function output, depending on the case
- stream
  With information about the input, after success or failure in recognition

Examples

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

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

 eofMark

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**

- **action**: Function to be executed if recognition succeeds. Character stream making up the token is passed as parameter to this function.
- **error**: Function to be executed if recognition does not succeed. Position of `streamParser` obtained with `streamParserPosition` is passed as parameter to this function.

**Details**

When succeeds, parameter `s` takes the value `""`.

**Value**

Anonymous function, returning a list.

```scala
function(stream) -> list(status, node, stream)
```

From input parameters, an anonymous function is defined. This function admits just one parameter, stream, with type `streamParser`, and returns a three-field list:

- **status**: "ok" or "fail"
- **node**: With `action` or `error` function output, depending on the case
- **stream**: With information about the input, after success or failure in recognition.

**Examples**

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

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

---

**isDigit**  
*Is it a digit?*

**Description**

Checks whether a character is a digit: `{ 0 .. 9 }.

**Usage**

```scala
isDigit(ch)
```
isHex

Arguments

ch  character to be checked

Value

TRUE/FALSE, depending on the character being a digit.

Examples

isDigit('9')
isDigit('a')

isHex

Is it an hexadecimal digit?

Description

Checks whether a character is an hexadecimal digit.

Usage

isHex(ch)

Arguments

ch  character to be checked

Value

TRUE/FALSE, depending on character being an hexadecimal digit.

Examples

isHex('+')
isHex('A')
isHex('a')
isHex('9')
isLetter  

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

ch  character to be checked

Value

TRUE/FALSE, depending on the character being a letter.

Examples

isLetter('A')
isLetter('a')
isLetter('9')

isLowercase  

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

ch  character to be checked

Value

TRUE/FALSE, depending on character being a lower case character.
### isNewline

**Examples**

- isLowercase('A')
- isLowercase('a')
- isLowercase('9')

<table>
<thead>
<tr>
<th>isNewline</th>
<th>Is it a new line character?</th>
</tr>
</thead>
</table>

**Description**

Checks whether a character is a new line character.

**Usage**

isNewline(ch)

**Arguments**

- ch character to be checked

**Value**

TRUE/FALSE, depending on character being a newline character

**Examples**

- isNewline(' ')  
- isNewline('\n')

<table>
<thead>
<tr>
<th>isSymbol</th>
<th>Is it a symbol?</th>
</tr>
</thead>
</table>

**Description**

Checks whether a character is a symbol, a special character.

**Usage**

isSymbol(ch)

**Arguments**

- ch character to be checked
Details

These characters are considered as symbols:

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

Value

TRUE/FALSE, depending on character being a symbol.

Examples

```plaintext
isSymbol('+

isSymbol('A'

isSymbol('a'

isSymbol('9')
```

<table>
<thead>
<tr>
<th>isUppercase</th>
<th>Is it an upper case?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description

Checks whether a character is an upper case.

Restricted to ASCII character (does not process ñ, ç, accented vowels...)

Usage

```plaintext
isUppercase(ch)
```

Arguments

ch character to be checked

Value

TRUE/FALSE, depending on character being an upper case character.

Examples

```plaintext
isUppercase('A'

isUppercase('a'

isUppercase('9')
```
isWhitespace

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**

ch character to be checked

**Value**
TRUE/FALSE, depending on character belonging to the specified set.

**Examples**
isWhitespace(' ')
isWhitespace('
')
isWhitespace('a')

**keyword**

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**

word Symbol to be recognised.

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

error Function to be executed if recognition does not succeed. Position of streamParser obtained with streamParserPosition is passed as parameter to this function
Anonymous function, returning a list.

\[
\text{function}(\text{stream}) \rightarrow \text{list}(\text{status}, \text{node}, \text{stream})
\]

From input parameters, an anonymous function is defined. This function admits just one parameter, stream, with type `streamParser`, and returns a three-field list:

- **status**
  - "ok" or "fail"
- **node**
  - With action or error function output, depending on the case
- **stream**
  - With information about the input, after success or failure in recognition

### Examples

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

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

---

**numberFloat**  
*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**

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

**Arguments**

- **action**
  - Function to be executed if recognition succeeds. Character stream making up the token is passed as parameter to this function
- **error**
  - Function to be executed if recognition does not succeed. Position of `streamParser` obtained with `streamParserPosition` is passed as parameter to this function
**Value**

Anonymous function, returning a list.

```r
defunction(stream) -> list(status, node, stream)
```

From input parameters, an anonymous function is defined. This function admits just one parameter, `stream`, with type `streamParser`, and returns a three-field list:

- **status**
  - "ok" or "fail"
- **node**
  - With action or error function output, depending on the case
- **stream**
  - With information about the input, after success or failure in recognition

**Examples**

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

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

---

**numberInteger**

**Integer number token.**

**Description**

Recognises an integer, i.e., a natural number optionally preceded by a + or - sign.

**Usage**

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

**Arguments**

- **action**
  - Function to be executed if recognition succeeds. Character stream making up the token is passed as parameter to this function
- **error**
  - Function to be executed if recognition does not succeed. Position of `streamParser` obtained with `streamParserPosition` is passed as parameter to this function
Value

Anonymous function, returning a list.

\[
\text{function}(\text{stream}) \rightarrow \text{list}(\text{status}, \text{node}, \text{stream})
\]

From input parameters, an anonymous function is defined. This function admits just one parameter, stream, with type \text{streamParser}, and returns a three-field list:

- status
  "ok" or "fail"
- node
  With action or error function output, depending on the case
- stream
  With information about the input, after success or failure in recognition

Examples

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

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

---

numberNatural  \(\text{Natural number token.}\)

Description

A natural number is a sequence of digits.

Usage

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

Arguments

- action  \(\text{Function to be executed if recognition succeeds. Character stream making up the token is passed as parameter to this function}\)
- error  \(\text{Function to be executed if recognition does not succeed. Position of}\ \text{streamParser} \text{obtained with}\ \text{streamParserPosition} \text{is passed as parameter to this function}\)
Anonymous function, returning a list.

function(stream) → list(status, node, stream)

From input parameters, an anonymous function is defined. This function admits just one parameter, stream, with type `streamParser`, and returns a three-field list:

- **status**
  "ok" or "fail"
- **node**
  With action or error function output, depending on the case
- **stream**
  With information about the input, after success or failure in recognition

**Examples**

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

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

---

**numberScientific**  
*Number in scientific notation token.*

**Description**

Recognises a number in scientific notation, i.e., a floating-point number with an (optional) exponential part.

**Usage**

```r
numberScientific(action = function(s) list(type="numberScientific",value=s),
                  error = function(p) list(type="numberScientific",pos=p) )
```

**Arguments**

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

- **error**  
  Function to be executed if recognition does not succeed. Position of `streamParser` obtained with `streamParserPosition` is passed as parameter to this function
Value

Anonymous function, returning a list.

```r
function(stream) -> list(status,node,stream)
```

From input parameters, an anonymous function is defined. This function admits just one parameter, `stream`, with type `streamParser`, and returns a three-field list:

- **status**
  "ok" or "fail"
- **node**
  With `action` or `error` function output, depending on the case
- **stream**
  With information about the input, after success or failure in recognition

Examples

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

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

<table>
<thead>
<tr>
<th>option</th>
<th>Optional parser</th>
</tr>
</thead>
</table>

Description

Applies a parser to the text. If it does not succeed, an empty token is returned.

Optional parser never fails.

Usage

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ap</td>
<td>Optional parser</td>
</tr>
<tr>
<td>action</td>
<td>Function to be executed if recognition succeeds. It takes as input parameters information derived from parsers involved as parameters</td>
</tr>
<tr>
<td>error</td>
<td>Function to be executed if recognition does not succeed. It takes two parameters:</td>
</tr>
<tr>
<td></td>
<td>• p with position where parser, <code>streamParser</code>, starts its recognition, obtained with <code>streamParserPosition</code></td>
</tr>
<tr>
<td></td>
<td>• 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</code> definition in these parsers.</td>
</tr>
</tbody>
</table>

Details

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

Function `error` is never called. It is defined as parameter for the sake of homogeneity with the rest of functions.

Value

Anonymous functions, returning a list.

```r
function(stream) -> list(status,node,stream)
```

From these input parameters, an anonymous function is constructed. This function admits just one parameter, `stream`, with `streamParser` class, and returns a three-field list:

- `status` "ok" or "fail"
- `node` With `action` or `error` function output, depending on the case
- `stream` With information about the input, after success or failure in recognition

Examples

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

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

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
cstream tree returned by the PC-AXIS file syntactical analysis

Value
It returns a list with the following elements:

- `headingLength` Number of variables in "HEADING".
- `StubLength` Number of variables in "STUB".
- `frequency` Data frequency if "TIMEVAL" is present.

pxCube (data.frame)

- `variableName` Variable name.
- `headingOrStud` Indicator, whether the variable appears in "HEADING" or "STUB".
- `codesYesNo` Indicator, whether there is "CODES" associated to the variable.
- `valuesYesNo` Indicator, whether there is "VALUES" associated to the variable.
- `variableOrder` Variable order number in "HEADING" or "STUB"
- `valueLength` Number of different "CODES" and/or "VALUES" associated with the variable.

pxCubeVariable (data.frame)

- `variableName` Variable name.
- `code` Value code when "CODES" is present.
- `value` Value literal when "VALUES" is present.
- `valueOrder` Variable order number in "CODES" and/or "VALUES".
- `eliminationYesNo` Indicator, whether the value for the variables is present in "ELIMINATION".
pxCubeVariableDomain (data.frame)

pxCubeAttrN  data.frame list, one for each different parameters cardinalities appearing in "keyword"

• pxCubeAttrNSA0 (data.frame)

  keyword    Keyword.
  language   Language code o "".
  length     Number of elements of value list.
  value      Associated data, keyword[language] = value.

• pxCubeAttrNSA1 (data.frame)

  keyword    Keyword.
  language   Language code o "".
  arg1       Argument value.
  length     Number of elements of value list.
  value      Associated data, keyword[language](arg) = value.

• pxCubeAttrNSA2 (data.frame)

  keyword    Keyword.
  language   Language code o "".
  arg1       Argument one value.
  arg2       Argument to value.
  length     Value list number of elements.
  value      Associated data, keyword[language](arg1,arg2) = value.

StubLength + headingLength columns , with variables values, ordered according to "STUB" and followed by those appearing in the data associated value.

pxCubeData (data.frame)

Returned value short version is:

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
References

PC-Axis file format.

Examples

```r
## Not run:
## significant time reductions may be achieve by doing:
library("compiler")
enableJIT(level=3)

## End(Not run)

name <- system.file("extdata","datInSFexample6_1.px", package = "qmrparser")
stream <- streamParserFromFileName(name,encoding="UTF-8")
cstream <- pcAxisParser(stream)
if ( cstream$status == "ok" ) {
cube <- pcAxisCubeMake(cstream)

## Variables
print(cube$pxCubeVariable)

## Data
print(cube$pxCubeData)

## Not run:
# # 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 <- "https://www.ine.es/pcaxisdl/t20/e245/p04/a2009/10/00000008.px"
stream <- streamParserFromString( readLines( name ) )
cstream <- pcAxisParser(stream)
if ( cstream$status == 'ok' ) cube <- pcAxisCubeMake(cstream)
# # is caused by files with a non-readable 'encoding'.
# In the case where it could be read, there may also be problems
# with string-handling functions, due to multibyte characters.
# In Windows, according to `Sys.getlocale()
# file may be read but accents, ñ, ... may not be correctly recognised.
```
There are, at least, the following options:
- File conversion to utf-8, from the OS, with
  "iconv - Convert encoding of given files from one encoding to another"
- File conversion in R:
  ```r
  name <- "https://www.ine.es/pcaxisdl/t20/e245/p04/a2009/l0/00000008.px"
  stream <- streamParserFromString(iconv(readLines(name), "IBM850", "UTF-8"))
  cstream <- pcAxisParser(stream)
  if (cstream$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 <- "https://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, ...)
  
  ## End(Not run)
```

---

**pcAxisCubeToCSV**

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**

- **prefix**
  - prefix for files to be created
- **pcAxisCube**
  - PC-AXIS cube
pcAxisParser

Details

Created files names are:

- prefix+"pxCube.csv"
- prefix+"pxCubeVariable.csv"
- prefix+"pxCubeVariableDomain.csv"
- prefix+"pxCubeData.csv"
- prefix+"pxCube"+name+.csv" With name = A0,A1,A2 ...

Value

NULL

Examples

```r
name <- system.file("extdata","datInSFexample6_1.px", package = "qmrparser")
stream <- streamParserFromFileName(name,encoding="UTF-8")
cstream <- pcAxisParser(stream)
if (cstream$status == 'ok') {
  cube <- pcAxisCubeMake(cstream)

  pcAxisCubeToCSV(prefix="datInSFexample6_1",pcAxisCube=cube)

  unlink("datInSFexample6_1*.csv")
}
```

---

**pcAxisParser**  
*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

- `streamParser`  
  stream parse associated to the file/text to be recognised
pcAxisParser

Details

Grammar definition, wider than the strict PC-AXIS definition

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

Value

Returns a list with "status" "node" "stream":

status
"ok" or "fail"

stream
Stream situation after recognition

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

- keyword
  PC-AXIS keyword
- language
  language code or ""
- parameters
  null or string list with parenthesised values associated to keyword
- 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 )

References

PC-Axis file format.


Examples

## Not run:
## significant time reductions may be achieve by doing:
library("compiler")
enableJIT(level=3)

## End(Not run)

name <- system.file("extdata","datInSFexample6_1.px", package = "qmrparser")
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)
```r
## DATA
print(Filter(function(e) e$keyword=="DATA",cstream$node)[[1]] $ruleRight$value)

## Not run:
#
# 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 <- "https://www.ine.es/pcaxisdl/t20/e245/p04/a2009/10/00000008.px"
# stream <- streamParserFromString( readLines( name ) )
# cstream <- pcAxisParser(stream)
# if ( cstream$status == 'ok' ) cube <- pcAxisCubeMake(cstream)
#
# is caused by files with a non-readable 'encoding'.
# In the case where it could be read, there may also be problems
# with string-handling functions, due to multibyte characters.
# In Windows, according to \code{Sys.getlocale}(),
# file may be read but accents, ñ, ... may not be correctly recognised.
#
#
# There are, at least, the following options:
# - File conversion to utf-8, from the OS, with
# "iconv - Convert encoding of given files from one encoding to another"
# - File conversion in R:
# name <- "https://www.ine.es/pcaxisdl/t20/e245/p04/a2009/10/00000008.px"
# stream <- streamParserFromString( iconv( readLines( name ), "IBM850", "UTF-8") )
# cstream <- pcAxisParser(stream)
# if ( cstream$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 <- "https://www.ine.es/pcaxisdl/t20/e245/p04/a2009/10/00000008.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, ...)
```
repetition0N

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

rpa0  parse to be applied iteratively
action Function to be executed if recognition succeeds. It takes as input parameters information derived from parsers involved as parameters
error  Function to be executed if recognition does not succeed. It takes two parameters:
  • p with position where parser, streamParser, starts its recognition, obtained with streamParserPosition
  • 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 error definition in these parsers.

Details

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

Function error is never called. It is defined as parameter for the sake of homogeneity with the rest of functions.
value

Anonymous functions, returning a list.

function(stream) -> list(status,node,stream)

From these input parameters, an anonymous function is constructed. This function admits just one
parameter, stream, with \texttt{streamParser} class, and returns a three-field list:

\begin{itemize}
\item \textbf{status}
  "ok" or "fail"
\item \textbf{node}
  With action or error function output, depending on the case
\item \textbf{stream}
  With information about the input, after success or failure in recognition
\end{itemize}

Examples

\begin{verbatim}
# ok
stream <- streamParserFromString("Hello world")
(repetition1N(symbolic()))(stream)[c("status","node")]

# ok
stream <- streamParserFromString("123 Hello world")
(repetition1N(symbolic()))(stream)[c("status","node")]
\end{verbatim}

---

\textbf{repetition1N} \hspace{1cm} \textit{Repeats a parser, at least once.}

\textbf{Description}

Repeats a parser application indefinitely while it is successful. It must succeed at least once.

\textbf{Usage}

\begin{verbatim}
repetition1N(rpa,
  action = function(s) list(type="repetition1N",value=s ),
  error = function(p,h) list(type="repetition1N",pos=p,h=h))
\end{verbatim}
Arguments

rpa
parse to be applied iteratively

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

error
Function to be executed if recognition does not succeed. It takes two parameters:

• p
with position where parser, streamParser, starts its recognition, obtained with streamParserPosition

• 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 error definition in these parsers.

Details

In case of success, action gets a list with information about the node returned by the applied parser. List length equals the number of successful repetitions.

In case of failure, parameter h from error gets error information returned by the first attempt of parser application.

Value

Anonymous functions, returning a list.

function(stream) -> list(status, node, stream)

From these input parameters, an anonymous function is constructed. This function admits just one parameter, stream, with streamParser class, and returns a three-field list:

• status
"ok" or "fail"

• node
With action or error function output, depending on the case

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

Examples

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

# fail
stream <- streamParserFromString("123 Hello world")
(repetition1N(symbolic()))(stream )[[c("status","node")]]
**Description**

Recognises a white character sequence, with comma or semicolon optionally inserted in the sequence. Empty sequences are not allowed.

**Usage**

```r
separator(action = function(s) list(type="separator",value=s) ,
       error = function(p) list(type="separator",pos =p) )
```

**Arguments**

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

- **error**
  Function to be executed if recognition does not succeed. Position of `streamParser` obtained with `streamParserPosition` is passed as parameter to this function.

**Details**

A character is considered a white character when function `isWhitespace` returns TRUE.

**Value**

Anonymous function, returning a list.

```r
function(stream) -> list(status,node,stream)
```

From input parameters, an anonymous function is defined. This function admits just one parameter, `stream`, with type `streamParser`, and returns a three-field list:

- **status**
  "ok" or "fail"

- **node**
  With `action` or `error` function output, depending on the case.

- **stream**
  With information about the input, after success or failure in recognition.

**Note**

PC-Axis has accepted the delimiters comma, space, semicolon, tabulator.
streamParser

Examples

```r
# 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")]
```

streamParser

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

```r
streamParserNextChar(stream)
streamParserNextCharSeq(stream)
streamParserPosition(stream)
streamParserClose(stream)
```

Arguments

stream  object containing information about the text to be processed and, specifically, about the next character to be read

Details

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

• streamParserPosition
  Returns information about text position being read.

• streamParserClose
  Closes the stream

Value

streamParserNextChar and streamParserNextCharSeq
Three field list:
  • status
    "ok" or "eof"
  • char
    Character read (ok) or "" (eof)
  • stream
    With information about next character to be read or same position if end of file has been reached ("eof")

streamParserPosition
Three field list:
  • fileName File name or "" if the stream is not associated with a file name
  • line
    line number
  • linePos
    character to be read position within its line
  • streamPos
    character to be read position from the text beginning

streamParserClose
NULL

See Also
streamParserFromFileName 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)

---

**streamParserFromFileName**  
*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**

- `fileName` file name
- `encoding` file encoding

**Details**

See `streamParser`

This function implementation uses function `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, `seek` is not possible.

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

```
streamParserFromString(readLines(fileName, encoding=encoding))
```

Alternatively, it can be used:

- `streamParserFromString` with: `streamParserFromString(readLines(fileName))`
- or
- `streamParserFromString(iconv(readLines(fileName), encodingOrigen, encodingDestino))`

Since streamParserFromFileName also uses `readChar`, this last option is the one advised in Linux if encoding is different from Latin-1 or UTF-8. As documentation states, `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."

streamParserFromString

Value

A list of four functions which allow stream manipulation:

streamParserNextChar
  Function which takes a streamParser as argument and returns a list(status, char, stream)

streamParserNextCharSeq
  Function which takes a streamParser as argument and returns list(status, char, stream)

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

streamParserClose
  Closes the stream

Examples

name <- system.file("extdata","datInTest01.txt", package = "qmrparser")
stream <- streamParserFromFileName(name)
cstream <- streamParserNextChar(stream)
while( cstream$status == "ok" ) {
  print(streamParserPosition(cstream$stream))
  print(cstream$char)
  cstream <- streamParserNextCharSeq(cstream$stream)
}
streamParserClose(stream)

streamParserFromString

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

string        string to be recognised
Details

See `streamParser`

Value

A list of four functions which allow stream manipulation:

- `streamParserNextChar` - Functions which takes a `streamParser` as argument ant returns a list(status, char, stream)
- `streamParserNextCharSeq` - Function which takes a `streamParser` as argument and returns a list(status, char, stream)
- `streamParserPosition` - Function which takes a `streamParser` as argument and returns position of next character to be read
- `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)

string

Description

Any character sequence, by default using simple or double quotation marks.

Usage

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

Arguments

isQuote Predicate indicating whether a character begins and ends a string
action Function to be executed if recognition succeeds. Character stream making up the token is passed as parameter to this function
error Function to be executed if recognition does not succeed. Position of streamParser obtained with streamParserPosition is passed as parameter to this function

Details

Characters preceded by \ are not considered as part of string end.

Value

Anonymous function, returning a list.

\begin{verbatim}
function(stream) -> list(status,node,stream)
\end{verbatim}

From input parameters, an anonymous function is defined. This function admits just one parameter, stream, with type streamParser, and returns a three-field list:

- status "ok" or "fail"
- node With action or error function output, depending on the case
- stream With information about the input, after success or failure in recognition

Examples

\begin{verbatim}
# fail
stream <- streamParserFromString("Hello world")
(string(stream))[[c("status","node")]]

# ok
stream <- streamParserFromString("'Hello world'")
(string(stream))[[c("status","node")]]
\end{verbatim}

symbolic 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

charFirst Predicate of valid characters as first symbol character
charRest Predicate of valid characters as the rest of symbol characters
action Function to be executed if recognition succeeds. Character stream making up
    the token is passed as parameter to this function
error Function to be executed if recognition does not succeed. Position of streamParser
    obtained with streamParserPosition is passed as parameter to this function

Value

Anonymous function, returning a list.

function(stream) -> list(status,node,stream)

From input parameters, an anonymous function is defined. This function admits just one parameter,
stream, with type streamParser, and returns a three-field list:

- status
  "ok" or "fail"
- node
  With action or error function output, depending on the case
- stream
  With information about the input, after success or failure in recognition

Examples

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

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

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

action Function to be executed if recognition succeeds. Character stream making up
the token is passed as parameter to this function
error Function to be executed if recognition does not succeed. Position of streamParser
obtained with streamParserPosition is passed as parameter to this function

Details

A character is considered a white character when function isWhitespace returns TRUE

Value

Anonymous function, returning a list.

function(stream) -> list(status,node,stream)

From input parameters, an anonymous function is defined. This function admits just one parameter,
stream, with type streamParser, and returns a three-field list:

• status
  "ok" or "fail"
• node
  With action or error function output, depending on the case
• stream
  With information about the input, after success or failure in recognition

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")]

whitespace
Index

* **PC-AXIS**
  pcAxisCubeMake, 24
  pcAxisCubeToCSV, 27
  pcAxisParser, 28
  qmrparser-package, 2

* **package**
  qmrparser-package, 2

* **parser combinator**
  alternation, 3
  concatenation, 8
  option, 22
  qmrparser-package, 2
  repetition0N, 32
  repetition1N, 33

* **set of character**
  isDigit, 12
  isHex, 13
  isLetter, 14
  isLowercase, 14
  isNewline, 15
  isSymbol, 15
  isUppercase, 16
  isWhitespace, 17

* **streamParser**
  streamParser, 36
  streamParserFromFileName, 38
  streamParserFromString, 39

* **token**
  charInSetParser, 4
  charParser, 5
  commentParser, 7
  dots, 9
  empty, 10
  eofMark, 11
  keyword, 17
  numberFloat, 18
  numberInteger, 19
  numberNatural, 20
  numberScientific, 21
  qmrparser-package, 2
  separator, 35
  string, 40
  symbolic, 41
  whitespace, 43
  alternation, 3
  charInSetParser, 4
  charParser, 5
  commentParser, 7
  concatenation, 8
  dots, 9
  empty, 10, 23, 32
  eofMark, 11
  iconv, 38
  isDigit, 12
  isHex, 13
  isLetter, 14
  isLowercase, 14
  isNewline, 15
  isSymbol, 15
  isUppercase, 16
  isWhitespace, 17, 35, 43
  keyword, 6, 17
  numberFloat, 18
  numberInteger, 19
  numberNatural, 20
  numberScientific, 21
  option, 22
  pcAxisCubeMake, 24
  pcAxisCubeToCSV, 27
  pcAxisParser, 28
  qmrparser (qmrparser-package), 2
qmrparser-package, 2

readChar, 38
readLines, 38
repetition0N, 32
repetition1N, 32, 33

seek, 38
separator, 35
streamParser, 3–12, 17–23, 32–35, 36, 38, 40–43
streamParserClose(streamParser), 36
streamParserFromFileName, 37, 38
streamParserFromString, 37, 38, 39
streamParserNextChar(streamParser), 36
streamParserNextCharSeq(streamParser), 36
streamParserPosition, 3, 5–10, 12, 17–21, 23, 32, 34, 35, 41–43
streamParserPosition(streamParser), 36
string, 40
symbolic, 41

whitespace, 43