Package 'Ramble'

January 20, 2025

Type Package

Title Parser Combinator for R

Version 0.1.1

Date 2016-10-23

Author Chapman Siu

Maintainer Chapman Siu <chpmn.siu@gmail.com>

Description Parser generator for R using combinatory parsers. It is inspired by combinatory parsers developed in Haskell.

License MIT + file LICENSE

Imports methods

Suggests testthat, knitr, rmarkdown

VignetteBuilder knitr

LazyData true

RoxygenNote 5.0.1

URL https://github.com/chappers/Ramble

NeedsCompilation no

Repository CRAN

Date/Publication 2016-10-29 10:48:13

Contents

`Alpha` *Alpha checks for single alphabet character*

Description

Alpha checks for single alphabet character

Usage

```
Alpha(...)
```

Arguments

`...`	additional arguments for the primitives to be parsed

See Also

`Digit`, `Lower`, `Upper`, `AlphaNum`, `SpaceCheck`, `String`, `ident`, `nat`, `space`, `token`, `identifier`, `natural`, `symbol`

Examples

```
Alpha()("abc")
```

AlphaNum *AlphaNum checks for a single alphanumeric character*

Description

AlphaNum checks for a single alphanumeric character

Usage

```
AlphaNum(...)
```

Arguments

...	additional arguments for the primitives to be parsed

See Also

Digit, Lower, Upper, Alpha, SpaceCheck, String, ident, nat, space, token, identifier, natural, symbol

Examples

```
AlphaNum()("123")
AlphaNum()("abc123")
```

---

alt *alt combinator is similar to alternation in BNF. the parser* (alt(p1, p2)) *recognises anything that* p1 *or* p2 *would. The approach taken in this parser follows (Fairbairn86), in which either is interpretted in a sequential (or exclusive) manner, returning the result of the first parser to succeed, and failure if neither does.*

Description

%alt% is the infix notation for the alt function, and it is the preferred way to use the alt operator.

Usage

```
alt(p1, p2)
```

Arguments

p1	the first parser
p2	the second parser

Value

Returns the first parser if it suceeds otherwise the second parser

See Also

then

Examples

```
(item() %alt% succeed("2")) ("abcdef")
```

`Digit` *Digit checks for single digit*

Description

Digit checks for single digit

Usage

```
Digit(...)
```

Arguments

`...`	additional arguments for the primitives to be parsed

See Also

Lower, Upper, Alpha, AlphaNum, SpaceCheck, String, ident, nat, space, token, identifier, natural, symbol

Examples

```
Digit()("123")
```

ident *ident is a parser which matches zero or more alphanumeric characters.*

Description

ident is a parser which matches zero or more alphanumeric characters.

Usage

```
ident()
```

See Also

Digit, Lower, Upper, Alpha, AlphaNum, SpaceCheck, String, nat, space, token, identifier, natural, symbol

Examples

```
ident() ("variable1 = 123")
```

identifier *identifier creates an identifier*

Description

identifier creates an identifier

Usage

```
identifier(...)
```

Arguments

...	takes in token primitives

See Also

Digit, Lower, Upper, Alpha, AlphaNum, SpaceCheck, String, ident, nat, space, token, natural, symbol

---

`item` *item is a parser that consumes the first character of the string and returns the rest. If it cannot consume a single character from the string, it will emit the empty list, indicating the parser has failed.*

Description

`item` is a parser that consumes the first character of the string and returns the rest. If it cannot consume a single character from the string, it will emit the empty list, indicating the parser has failed.

Usage

```
item(...)
```

Arguments

`...`	additional arguments for the parser

Examples

```
item() ("abc")
item() ("")
```

`literal` *literal is a parser for single symbols. It will attempt to match the single symbol with the first character in the string.*

Description

`literal` is a parser for single symbols. It will attempt to match the single symbol with the first character in the string.

Usage

```
literal(char)
```

Arguments

`char`	is the character to be matched

Examples

```
literal("a") ("abc")
```

Lower — *Lower checks for single lower case character*

Description

Lower checks for single lower case character

Usage

```
Lower(...)
```

Arguments

...	additional arguments for the primitives to be parsed

See Also

Digit, Upper, Alpha, AlphaNum, SpaceCheck, String, ident, nat, space, token, identifier, natural, symbol

Examples

```
Lower() ("abc")
```

many — many *matches 0 or more of pattern* p. *In BNF notation, repetition occurs often enough to merit its own abbreviation. When zero or more repetitions of a phrase* p *are admissible, we simply write* p*. *The* many *combinator corresponds directly to this operator, and is defined in much the same way.*

Description

This implementation of many differs from (Hutton92) due to the nature of R's data structures. Since R does not support the concept of a list of tuples, we must revert to using a list rather than a vector, since all values in an R vector must be the same datatype.

Usage

```
many(p)
```

Arguments

p	is the parser to match 0 or more times.

See Also

maybe, some

Examples

```
Digit <- function(...) {satisfy(function(x) {return(!!length(grep("[0-9]", x)))})}
many(Digit()) ("123abc")
many(Digit()) ("abc")
```

---

maybe — maybe *matches 0 or 1 of pattern* p. *In EBNF notation, this corresponds to a question mark ('?').*

Description

maybe matches 0 or 1 of pattern p. In EBNF notation, this corresponds to a question mark ('?').

Usage

```
maybe(p)
```

Arguments

p	is the parser to be matched 0 or 1 times.

See Also

many, some

Examples

```
maybe(Digit())("123abc")
maybe(Digit())("abc123")
```

---

nat — nat *is a parser which matches one or more numeric characters.*

Description

nat is a parser which matches one or more numeric characters.

Usage

```
nat()
```

See Also

Digit, Lower, Upper, Alpha, AlphaNum, SpaceCheck, String, ident, space, token, identifier, natural, symbol

Examples

```
nat() ("123 + 456")
```

---

natural *natural creates a token parser for natural numbers*

Description

natural creates a token parser for natural numbers

Usage

```
natural(...)
```

Arguments

...	additional arguments for the parser

See Also

Digit, Lower, Upper, Alpha, AlphaNum, SpaceCheck, String, ident, nat, space, token, identifier, symbol

---

Ramble *Ramble is a parser generator using combinatory parsers.*

Description

Ramble allows you to write parsers in a functional manner, inspired by Haskell's Parsec library.

satisfy	satisfy *is a function which allows us to make parsers that recognise single symbols.*

Description

satisfy is a function which allows us to make parsers that recognise single symbols.

Usage

```
satisfy(p)
```

Arguments

p	is the predicate to determine if the arbitrary symbol is a member.

some	some *matches 1 or more of pattern* p. *in BNF notation, repetition occurs often enough to merit its own abbreviation. When zero or more repetitions of a phrase* p *are admissible, we simply write* p+. *The* some *combinator corresponds directly to this operator, and is defined in much the same way.*

Description

some matches 1 or more of pattern p. in BNF notation, repetition occurs often enough to merit its own abbreviation. When zero or more repetitions of a phrase p are admissible, we simply write p+. The some combinator corresponds directly to this operator, and is defined in much the same way.

Usage

```
some(p)
```

Arguments

p	is the parser to match 1 or more times.

See Also

maybe, many

Examples

```
Digit <- function(...) {satisfy(function(x) {return(!!length(grep("[0-9]", x)))})}
some(Digit()) ("123abc")
```

space — space *matches zero or more space characters.*

Description

space matches zero or more space characters.

Usage

```
space()
```

See Also

Digit, Lower, Upper, Alpha, AlphaNum, SpaceCheck, String, ident, nat, token, identifier, natural, symbol

Examples

```
space() ("  abc")
```

SpaceCheck — *SpaceCheck checks for a single space character*

Description

SpaceCheck checks for a single space character

Usage

```
SpaceCheck(...)
```

Arguments

...	additional arguments for the primitives to be parsed

See Also

Digit, Lower, Upper, Alpha, AlphaNum, String, ident, nat, space, token, identifier, natural, symbol

Examples

```
SpaceCheck()(" 123")
```

`String` `String` *is a combinator which allows us to build parsers which recognise strings of symbols, rather than just single symbols*

Description

`String` is a combinator which allows us to build parsers which recognise strings of symbols, rather than just single symbols

Usage

```
String(string)
```

Arguments

`string`	is the string to be matched

See Also

`Digit`, `Lower`, `Upper`, `Alpha`, `AlphaNum`, `SpaceCheck`, `ident`, `nat`, `space`, `token`, `identifier`, `natural`, `symbol`

Examples

```
String("123")("123 abc")
```

`succeed` `succeed` *is based on the empty string symbol in the BNF notation The* `succeed` *parser always succeeds, without actually consuming any input string. Since the outcome of succeed does not depend on its input, its result value must be pre-detemined, so it is included as an extra parameter.*

Description

`succeed` is based on the empty string symbol in the BNF notation The `succeed` parser always succeeds, without actually consuming any input string. Since the outcome of succeed does not depend on its input, its result value must be pre-detemined, so it is included as an extra parameter.

Usage

```
succeed(string)
```

Arguments

`string`	the result value of succeed parser

Examples

```
succeed("1") ("abc")
```

`symbol` `symbol` *creates a token for a symbol*

Description

`symbol` creates a token for a symbol

Usage

```
symbol(xs)
```

Arguments

`xs`	takes in a string to create a token

See Also

`Digit`, `Lower`, `Upper`, `Alpha`, `AlphaNum`, `SpaceCheck`, `String`, `ident`, `nat`, `space`, `token`, `identifier`, `natural`

Examples

```
symbol("[") ("  [123]")
```

`then` `then` *combinator corresponds to sequencing in BNF. The parser* `(then(p1, p2))` *recognises anything that* `p1` *and* `p2` *would if placed in succession.*

Description

`%then%` is the infix operator for the then combinator, and it is the preferred way to use the `then` operator.

Usage

```
then(p1, p2)
```

Arguments

`p1`	the first parser
`p2`	the second parser

Value

recognises anything that p1 and p2 would if placed in succession.

See Also

alt, thentree

Examples

```
(item() %then% succeed("123")) ("abc")
```

---

thentree — thentree *keeps the full tree representation of the results of parsing. Otherwise, it is identical to* then.

Description

thentree keeps the full tree representation of the results of parsing. Otherwise, it is identical to then.

Usage

```
thentree(p1, p2)
```

Arguments

p1	the first parser
p2	the second parser

Value

recognises anything that p1 and p2 would if placed in succession.

See Also

alt, thentree

Examples

```
(item() %thentree% succeed("123")) ("abc")
```

`token` *token is a new primitive that ignores any space before and after applying a parser to a token.*

Description

`token` is a new primitive that ignores any space before and after applying a parser to a token.

Usage

```
token(p)
```

Arguments

`p`	is the parser to have spaces stripped.

See Also

`Digit`, `Lower`, `Upper`, `Alpha`, `AlphaNum`, `SpaceCheck`, `String`, `ident`, `nat`, `space`, `identifier`, `natural`, `symbol`

Examples

```
token(ident()) ("   variable1   ")
```

`Unlist` *Unlist is the same as unlist, but doesn't recurse all the way to preserve the type. This function is not well optimised.*

Description

Unlist is the same as unlist, but doesn't recurse all the way to preserve the type. This function is not well optimised.

Usage

```
Unlist(obj)
```

Arguments

`obj`	is a list to be flatten

Upper — *Upper checks for a single upper case character*

Description

Upper checks for a single upper case character

Usage

```
Upper(...)
```

Arguments

...	additional arguments for the primitives to be parsed

See Also

Digit, Lower, Alpha, AlphaNum, SpaceCheck, String, ident, nat, space, token, identifier, natural, symbol

Examples

```
Upper()("Abc")
```

using — *using combinator allows us to manipulate results from a parser, for example building a parse tree. The parser (p %using% f) has the same behaviour as the parser p, except that the function f is applied to each of its result values.*

Description

%using% is the infix operator for using, and it is the preferred way to use the using operator.

Usage

```
using(p, f)
```

Arguments

p	is the parser to be applied
f	is the function to be applied to each result of p.

Value

The parser (p %using% f) has the same behaviour as the parser p, except that the function f is applied to each of its result values.

Examples

```
(item() %using% as.numeric) ("1abc")
```

`%alt%` *%alt% is the infix notation for the* `alt` *function.*

Description

`%alt%` is the infix notation for the `alt` function.

Usage

```
p1 %alt% p2
```

Arguments

`p1`	the first parser
`p2`	the second parser

Value

Returns the first parser if it suceeds otherwise the second parser

Examples

```
(item() %alt% succeed("2")) ("abcdef")
```

`%then%` *%then% is the infix operator for the then combinator.*

Description

`%then%` is the infix operator for the then combinator.

Usage

```
p1 %then% p2
```

Arguments

`p1`	the first parser
`p2`	the second parser

Value

recognises anything that `p1` and `p2` would if placed in succession.

Examples

```
(item() %then% succeed("123")) ("abc")
```

`%thentree%` *`%thentree%` is the infix operator for the then combinator, and it is the preferred way to use the* `thentree` *operator.*

Description

`%thentree%` is the infix operator for the then combinator, and it is the preferred way to use the `thentree` operator.

Usage

```
p1 %thentree% p2
```

Arguments

`p1`	the first parser
`p2`	the second parser

Value

recognises anything that `p1` and `p2` would if placed in succession.

See Also

`alt`, `thentree`

Examples

```
(item() %thentree% succeed("123")) ("abc")
```

`%using%` *`%using%` is the infix operator for using*

Description

`%using%` is the infix operator for using

Usage

```
p %using% f
```

Arguments

p	is the parser to be applied
f	is the function to be applied to each result of p.

Examples

```
(item() %using% as.numeric) ("1abc")
```

Index