Package 'parcr'

May 9, 2026

Title Construct Parsers for Structured Text Files

Version 0.6.1

Description Construct parser combinator functions, higher order functions that parse input. Construction of such parsers is transparent and easy. Their main application is the parsing of structured text files like those generated by laboratory instruments. Based on a paper by Hutton (1992) <doi:10.1017/S0956796800000411>.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

Suggests knitr, rmarkdown, stringr, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

Depends R (>= 4.1)

LazyData true

URL https://github.com/SystemsBioinformatics/parcr

BugReports https://github.com/SystemsBioinformatics/parcr/issues

NeedsCompilation no

Author Douwe Molenaar [aut, cre, cph] (ORCID: <https://orcid.org/0000-0001-7108-4545>)

Maintainer Douwe Molenaar <d.molenaar@vu.nl>

Repository CRAN

Date/Publication 2026-02-17 09:40:08 UTC

Contents

by_split 2
by_symbol 3
clear_store 4

EmptyLine 4
eof 5
failed 6
fastafile 7
finished 7
has_stored 8
Ignore 8
list_stored 9
literal 9
match_s 10
named 11
print.marker 12
reporter 13
satisfy 14
store 15
stringparser 16
succeed 17
zero_or_more 18
%or% 20
%ret% 21
%then% 22
%using% 23
%xthen% 23

Index **25**

`by_split` *Applying a parser to a split string*

Description

Splits a string by using a split pattern and then applies the parser `p` to the resulting character vector. If `finish = TRUE` then the parser should completely consume its input, otherwise the parser fails. If `finish = FALSE` then any remaining part of the string is discarded.

Usage

```
by_split(p, split, finish = TRUE, fixed = FALSE, perl = FALSE)
```

Arguments

`p`	A parser.
`split`	a string (or object which can be coerced to such) containing regular expression(s) (unless fixed = TRUE) to use for splitting. If empty matches occur, in particular if split has length 0, x is split into single characters.
`finish`	logical. Should the parser completely consume the string? Defaults to `TRUE`.

`fixed`	logical. If `TRUE` match `split` exactly, otherwise use regular expressions. Has priority over `perl`.
`perl`	logical. Should Perl-compatible regexps be used?

Details

The function `base::strsplit()` is used to perform the splitting. The parameters `split`, `fixed` and `perl` are passed on to that function.

Value

A parser.

See Also

`base::strsplit()`, `by_symbol()`

Examples

```
by_split((literal("a") %then% literal("b")),"\\t") ("a\tb") # success
by_split((literal("a") %then% literal("b")),"\\t") ("a\tb\tc") # failure
by_split((literal("a") %then% literal("b")),"\\t", finish=FALSE) ("a\tb\tc") # success
```

---

by_symbol *Applying a parser to individual symbols of a string*

Description

Splits a string to individual symbols and then applies the parser `p` to the resulting character vector, otherwise the parser fails. If `finish = TRUE` then the parser should completely consume its input. If `finish = FALSE` then any remaining part of the string is discarded.

This function is identical to `by_split(p, "", finish)`.

Usage

```
by_symbol(p, finish = TRUE)
```

Arguments

`p`	A parser.
`finish`	logical. Should the parser completely consume the string? Defaults to `TRUE`.

Value

A parser.

See Also

by_split()

Examples

```
by_symbol(exactly(3,literal("a"))) (c("aaa", "bb")) # success
by_symbol(exactly(3,literal("a"))) (c("aaaa", "bb")) # failure
```

---

clear_store *Clear all stored values*

Description

Removes all variables stored using `store()`. Use this to clean up between independent parser runs or in test teardown.

Usage

```
clear_store()
```

EmptyLine *Recognize empty lines*

Description

An empty line is a line that consists entirely of space-like characters. `EmptyLine` is a parser that recognizes one empty line, `Spacer` recognizes one or more empty lines and `MaybeEmpty` recognizes zero or more empty lines. `EmptyLine` returns the empty line but `Spacer` and `MaybeEmpty` discard these.

Usage

```
EmptyLine()

Spacer()

MaybeEmpty()
```

Value

A parser.

Pseudocode

```
space_like_eraser(x):
    d = x in which all "\\s+" are replaced by ""
    if d=="" TRUE else FALSE

Emptyline: satisfy(space_like_eraser)

Spacer: one_or_more(EmptyLine()) %ret% null

MaybeEmpty: zero_or_more(EmptyLine()) %ret% null
```

where `null` is the empty vector.

Examples

```
EmptyLine()(" \t  ") # success
EmptyLine()("    .") # failure
EmptyLine()("") # success
Spacer()(c("   \t  ", "    ", "abc"))
Spacer()(c("            ", "    ", "Important text"))
Spacer()(c("Important text")) # failure, missing empty line
MaybeEmpty()(c("            ", "    ", "Important text")) # success, just as Spacer()
MaybeEmpty()(c("Important text")) # success, in contrast to Spacer()
```

`eof` *Detect end of input*

Description

Tests whether the end of the input character vector has been reached, which boils down to detection of `character(0)` in the R-element (see `succeed()`). Since the intended application of this parser is parsing of text files the function has been called after the end of file (EOF) signal. To indicate that an end of file has been detected, the R-element side of the parser output will be converted to an empty list.

Usage

```
eof()
```

Value

A parser.

Pseudocode

```
eof()(x):
  if x==null then succeed(x)(list())
  else fail()(x)
```

Examples

```
(literal("a") %then% eof())("a") # success
# Notice the difference on the R-side with
literal("a")("a")
eof()(character(0)) # success
eof()("a") # failure
```

`failed` *Testing for parser failure*

Description

Use this function to test whether your parser failed, for example in unit testing of your parsers when writing a package.

Usage

```
failed(o)
```

Arguments

`o`	Output from a parser.

Value

A logical value.

See Also

`print.marker()`

Examples

```
d <- (literal("A") %then% literal("B"))(c("A", "A"))
d
failed(d)
```

fastafile *Example nucleotide fasta file*

Description

An example fasta-formatted file with a mixture of nucleotide and protein sequences. It is used in the vignette to demonstrate parsing with the tools from the package. It is not clear to me whether mixing of sequence types is allowed in a fasta file, but we demonstrate in the vignette that is is easy to parse them from a single file. The sequences used are truncated for the sake of the example.

Usage

```
fastafile
```

Format

`fastafile`**:**

A character vector

Source

Modified from https://bioinformatics.org/annhyb/examples/seq_fasta.html and https://en.wikipedia.org/wiki/FASTA_format

finished *Test whether the parser has completely consumed the input*

Description

Checks if the parser result indicates that the entire input was consumed. A parser has completely consumed its input when the input has satisfied `eof()`.

Usage

```
finished(o)
```

Arguments

o	Output from a parser.

Value

A logical value.

Examples

```
finished((literal("A") %then% eof())("A")) # TRUE
finished((literal("A"))("A")) # FALSE
finished((literal("A") %then% eof())(c("A", "C"))) # FALSE
```

---

has_stored *Check if a variable is stored*

Description

Check if a variable is stored

Usage

```
has_stored(name)
```

Arguments

name	Variable name to check

Value

Logical

Ignore *Ignore all until the end*

Description

Sometimes, a parser has found the end of the text that needs to be parsed and all following lines can be ignored. `Ignore` will read and discard all following lines until the end of the file, empty or not. So, this parser will consume all elements until the end of the input vector.

Usage

```
Ignore()
```

Details

This parser is identical to

```
zero_or_more(satisfy(function(x) TRUE))
```

Value

A parser.

Examples

```
starts_with_a <- function(x) grepl("^a", x)
p <- function() {
  one_or_more(satisfy(starts_with_a)) %then%
    (literal("~End") %ret% NULL) %then%
    Ignore() %then%
    eof()
}
p()(c("ab", "abc", "~End", "boring stuff", "more stuff"))
```

---

list_stored *List all stored variable names*

Description

Returns the names of all variables currently stored in the parser environment.

Usage

```
list_stored()
```

Value

Character vector of variable names

---

literal *Matching parser input with a literal string*

Description

`literal` tests whether a supplied string literally equals a desired value.

Usage

```
literal(string)
```

Arguments

`string`	string, a single-element character vector, or an object that can be coerced to a character vector.

Value

A parser.

Pseudocode

```
literal(a)(x): satisfy(F(y): y==a)(x)
```

where F is equivalent to the function declarator in R. So, we have an anonymous function in the argument of satisfy.

Examples

```
literal("ab")(c("ab", "cdef")) # success
literal("ab")(c("abc", "cdef")) # failure
```

---

match_s *Identifying and processing a string and producing custom output*

Description

match_s matches a string using a function and returns a desired object type.

Usage

```
match_s(s)
```

Arguments

s	a string-parsing function.

Details

This parser short-cuts the pattern satisfy(b) %using% f. With match_s you do not have to write separate predicate and processing functions b and f when identification and parsing can be done with a single string parsing function s.

The function s will be given a non-empty single-element character vector as its argument, so you don't have to test for empty input, like character(0). These two facts also often simplify further processing with the string functions like grep, regmatches and those from the stringr package. The function s can return any R-object when succeeding, but to signal failure to the parser it must return the empty list(). Note that list() output from s will be turned into a marker object, the internal object to mark failure, by match_s(), see failed().

Value

A parser.

Pseudocode

```
match_s(s)(x):
  if x==null then fail()(x)
  else if s(x[1]) then succeed(s(x[1]))(x[-1]) else fail()(x)
```

Examples

```
expect_integers <- function(x) {
  m <- gregexpr("[[:digit:]]+", x)
  matches <- regmatches(x, m)[[1]]
  if (length(matches) == 0) {
    # this means failure to detect numbers where we expected them
    return(list())
  } else {
    return(as.numeric(matches))
  }
}

match_s(expect_integers)("12 15 16 # some comment") # success
match_s(expect_integers)("some text") # failure
```

---

named *Add a semantic name to a parser for better error messages*

Description

named() wraps a parser and provides a meaningful name that will be shown in error messages when the parser fails. This is particularly useful for user-defined parsers to make error messages more informative.

Usage

```
named(p, name)
```

Arguments

p	a parser.
name	a character string describing what the parser expects.

Value

A parser.

Examples

```
# Define a parser with a semantic name
Nucleotide <- function() {
  named(
    satisfy(function(x) grepl("^[GATC]+$", x)),
    "nucleotide sequence"
  )
}

# When this parser fails, the error will say "Expected: nucleotide sequence"
```

```
try(reporter(Nucleotide() %then% eof())(c("GATCxTC")))

# Combine named parsers with %or% to get helpful "Expected one of:" messages
Nucleotide_or_Protein <- function() {
  named(satisfy(function(x) grepl("^[GATC]+$", x)), "nucleotide") %or%
    named(satisfy(function(x) grepl("^[ARNDCQEGHILKMFPSTWYV]+$", x)), "protein")
}

try(reporter(Nucleotide_or_Protein() %then% eof())(c("Some text")))
```

`print.marker` *Print method for an object of class* `marker`

Description

An object of class `marker` is an empty list created by the function `fail()`. To indicate that this object differs from simply `list()` its print method prints `[]`.

Usage

```
## S3 method for class 'marker'
print(x, ...)
```

Arguments

`x`	an object used to select a method.
`...`	further arguments passed to or from other methods.

Details

The `marker` class is used internally to mark the largest index number of the element (i.e. line) of the input character vector at which the parser failed. This number is stored in the attribute `n` of a marker and only correctly corresponds to that index number if the parser is wrapped in a `reporter()` call.

Value

The printed `marker` object is returned invisibly.

See Also

`failed()`

Examples

```
d <- (literal("A") %then% literal("B"))(c("A", "A"))
# prints the icon [] for failed parsing
d
# Reveal the modest content of the marker object
unclass(d)
```

`reporter` *Turn a parser into an error reporting parser*

Description

Turns a parser into an error reporting parser, and when the parser is successful returns only the `L`-element of the parser output, the successfully parsed part of the input (see `succeed()`).

Usage

```
reporter(p, context_size = 5)
```

Arguments

`p`	a parser.
`context_size`	number of lines of context to show around the line where failure occurred. Default is 5.

Details

The error object that this function returns is a list containing the elements `linenr` and `linecontent`, corresponding to the line in which the parser failed and its content. The user of this package can catch this object to create custom error messages instead of the message generated by this function.

A warning is issued when the parser did not completely consume the input. Complete consumption of input is only explicitly made when the parser ends with `eof()`. Therefore, even though all elements were parsed, a zero-length character vector will remain in the `R` element if the parser does not end with `eof()`.

Value

The `L`-part of a successful parser result or an error message about the line where the parser failed. A warning is thrown when the parser did not completely consume the input.

Examples

```
at <- function() literal("a") %then% literal("t")
atat <- rep(c("a", "t"), 2)
# Yields an error message about parser failing on line 5
try(
  reporter(match_n(3, at()) %then% eof())(c(atat, "t", "t"))
)
# No error, but parser result
reporter(match_n(2, at()) %then% eof())(atat)
# warning: the input is not completely consumed
try(
  reporter(match_n(2, at()))(atat)
)
```

satisfy *Matching input using a logical function*

Description

`satisfy()` turns a logical function into a parser that recognizes strings.

Usage

```
satisfy(b, expected = "matching input")
```

Arguments

b	a boolean function to determine if the string is accepted.
expected	character string describing what is expected (for error messages).

Details

Notice (see pseudocode) that `satisfy` fails when presented with empty input, so it is futile to write predicate functions that would recognize such input.

Value

A parser.

Pseudocode

```
satisfy(b)(x):
  if x==null then fail()(x)
  else if b(x[1]) then succeed(x[1])(x[-1]) else fail()(x)
```

where `x[1]` is the first element of `x`, `x[-1]` all subsequent elements (or `null` if it only has one element). `null` is the empty vector, equivalent to `character(0)` in R.

Examples

```
# define a predicate function that tests whether the next element starts
# with an 'a'
starts_with_a <- function(x) grepl("^a", x)
# Use it in the satisfy parser
satisfy(starts_with_a)(c("abc", "def")) # success
satisfy(starts_with_a)(c("bca", "def")) # failure
# Using an anonymous function
satisfy(function(x) {
  as.numeric(x) > 10
})("15") # success
```

store *Store and retrieve objects*

Description

Sometimes you want to use a parsed object to modify a later parser operation, as in the example below. The `store()` and `retrieve()` functions provide the tools to create such a parser.

Usage

```
store(name, value)

retrieve(name)
```

Arguments

`name`	a string used as the name of the stored object.
`value`	object to be stored.

Value

Nothing for `store()` and the stored object for `retrieve()`.

Examples

```
parse_nr <- function(line) {
  m <- stringr::str_extract(line, "number=(\\d+)", group = 1)
  if (is.na(m)) {
    list()
  } else {
    store("nr", as.numeric(m))
  }
}

p <- function() {
  match_s(parse_nr) %then%
    exactly(retrieve("nr"), literal("A"))
}

p()(c("number=3", "A", "A", "A")) # success
p()(c("number=2", "A", "A", "A")) # failure
```

`stringparser` *String parser constructor*

Description

Produce a string parser based on `stringr::str_match()`, to be used with `match_s()`.

Usage

```
stringparser(match_pattern, reshape = identity)
```

Arguments

`match_pattern`	A regular expression that matches the string.
`reshape`	A function that takes the character vector of captured strings and modifies it to a desired output. By default this is the `identity()` function.

Details

This function uses `regexec()` with parameter `perl = TRUE` to produce a string parser for perl compatible regular expressions. Parsers created with this constructor return the failure signal `list()` when a string does not match the `match_pattern`. If the pattern contains captured groups then these groups will be returned as a character vector upon matching. If there is no capture group then the function will return silently upon matching the pattern. You can provide a function to the `reshape` argument to change the output upon matching.

You always have to wrap the parsers made with this constructor in `match_s()` to create a parser combinator. I decided not to include this standard pattern in the `stringparser` constructor itself because it complicates testing of these parsers.

See Also

`match_s()`, `regexec()`

Examples

```
# single capture group
parse_header <- stringparser("^>(\\w+)")
parse_header(">correct_header") # returns "correct_header"
parse_header("> incorrect_header") # returns list()

# multiple capture groups
parse_keyvalue <- stringparser("(\\w+):\\s?(\\w+)")
parse_keyvalue("key1: value1") # returns c("key1", "value1")

# modify output
parse_keyvalue_df <- stringparser(
  "(\\w+):\\s?(\\w+)",
  function(x) data.frame(key = x[1], value = x[2])
```

```
)
parse_keyvalue_df("key1: value1") # returns a data frame
```

---

succeed *The most basic parsers*

Description

These are the most basic constructors of a parser, but they are important cogs of the parser machinery. The succeed parser always succeeds, without consuming any input, whereas the `fail` parser always fails.

Usage

```
succeed(left)

fail(lnr = LNR(), expected = NULL)
```

Arguments

`left`	any R-object constructed from a parsed vector.
`lnr`	integer. The line number (element number) at which the fail occurs
`expected`	character vector. Optional descriptions of what was expected.

Details

The succeed parser constructs a `list` object with a 'left' or `L`-element that contains the parser result of the consumed part of the input vector and the 'right' or `R`-element that contains the unconsumed part of the vector. Since the outcome of succeed does not depend on its input, its result value must be pre-determined, so it is included as a parameter.

While succeed never fails, `fail` always does, regardless of the input vector. It returns the empty list `list()` to signal this fact.

Value

A list. `succeed()` returns a list with two elements named `L` and `R`. `fail()` returns a `marker` object which is basically an empty list with a line number `n` as attribute. It is printed as the icon `[]`, see `print.marker()`. Note that `n` will only correctly represent the line number of failure when a parser is wrapped in the `reporter()` function.

Pseudocode

```
succeed(y)(x): [L=[y],R=[x]]
fail()(x):     []
```

where `[L=[y],R=[x]]` is a named list with lists `[y]` and `[x]` as elements and `[]` is an empty list.

Note

It is very unlikely that you will ever have to use these functions when constructing parsers.

Examples

```
succeed("A")("abc")
succeed(data.frame(title = "Keisri hull", author = "Jaan Kross"))(c("Unconsumed", "text"))

fail()("abc")
```

`zero_or_more` *Repeated application of a parser*

Description

Often, we want to assess whether a given structure can be successfully parsed through repetitive application of a parser `p`. This could involve testing the parser applied multiple times in succession or determining its capability to be applied zero or more times.

The subsequent functions are designed to address and evaluate these scenarios.

Usage

```
zero_or_more(p)

one_or_more(p)

exactly(n, p)

zero_or_one(p)

match_n(n, p)
```

Arguments

`p`	a parser.
`n`	a positive integer, including 0.

Details

All these parsers with the exception of `match_n` exhibit greedy behavior striving to apply `p` as many times as possible. If the resulting count doesn't match the expected quantity, such as in the case of `exactly(n,p)` where `p` successfully parses more than `n` times, then the parser fails. In contrast, `match_n(n,p)` strictly applies `p` exactly `n` times, preventing any further application of `p` even if `p` could potentially be applied more often. Clearly, both functions will fail if `p` fails after less than `n` repetitions.

Value

A parser.

Pseudocode

```
zero_or_more(p):
  (p %then% zero_or_more(p)) %or% succeed(null)

one_or_more(p):
  p %then% zero_or_more(p)

exactly(n,p):
  count = 0
  r = zero_or_more(p %using% F(x): count = count + 1; x)(x)
  if count == n then
    count = 0
    r
  else fail()(x)

zero_or_one:
  exactly(1,p) %or% exactly(0,p)

match_n(n,p):
  if n==0 then F(x): succeed(list())(x)
  else
    if n==1 then p else (p %then% match_n(n-1, p))
```

where `null` is the empty vector.

Examples

```
zero_or_more(literal("A"))(c("A", LETTERS[1:5]))
zero_or_more(literal("A"))(LETTERS[2:5])

one_or_more(literal("A"))(c("A", LETTERS[1:5])) # success
one_or_more(literal("A"))(LETTERS[2:5]) # failure

exactly(2, literal("A"))(c("A", LETTERS[1:5])) # success
exactly(2, literal("A"))(c(rep("A", 2), LETTERS[1:5])) # failure: too many "A"

zero_or_one(literal("A"))(LETTERS[2:5]) # success
zero_or_one(literal("A"))(LETTERS[1:5]) # success
zero_or_one(literal("A"))(c("A", LETTERS[1:5])) # failure

match_n(2, literal("A"))(c("A", LETTERS[1:5])) # success
match_n(2, literal("A"))(c(rep("A", 2), LETTERS[1:5])) # success
```

`%or%` *Applying alternative parsers*

Description

The `%or%` combinator (`p1 %or% p2`) returns the result of `p1` if `p1` is successful or, if `p1` fails that of `p2` if `p2` parses successfully, otherwise it returns a `fail`.

Usage

```
p1 %or% p2
```

Arguments

`p1`, `p2`	two parsers.

Value

A parser.

Pseudocode

```
(p1 %or% p2)(x):
  if p1(x)==[] then
    if p2(x)==[] then fail()(x) else p2(x)
  else p1(x)
```

where `[]` is the empty list.

Examples

```
(literal("A") %or% literal("a"))(LETTERS[1:5]) # success on first parser
(literal("A") %or% literal("a"))(letters[1:5]) # success on second parser
(literal("A") %or% literal("a"))(LETTERS[2:6]) # failure
starts_with_a <- function(x) grepl("^a", x[1])
# success on both parsers, but returns result of p1 only
(literal("a") %or% satisfy(starts_with_a))(letters[1:5])
```

`%ret%` *Return a fixed value instead of the result of a parser*

Description

Sometimes we are not interested in the result from a parser, only that the parser succeeds. It may be convenient to return some short representation or nothing even rather than the string itself. The `%ret%` combinator is useful in such cases. The parser `(p %ret% c)` has the same behavior as `p`, except that it returns the value `c` if successful.

Usage

```
p %ret% c
```

Arguments

`p`	a parser.
`c`	string, *i.e.* a single-element character vector. `NULL` is coerced to `character(0)`.

Value

A parser.

Pseudocode

```
(p %xret% c)(x):
  if p(x)==[] then fail()(x)
  else succeed(c)(x[-1])
```

See Also

%using%

Examples

```
(literal("A") %ret% "We have an A!")(LETTERS[1:5])
(literal("A") %ret% NULL)(LETTERS[1:5])
```

`%then%` *Applying parsers in sequence*

Description

`(p1 %then% p2)` recognizes anything that `p1` and `p2` would if applied in succession.

Usage

```
p1 %then% p2
```

Arguments

`p1, p2`	two parsers.

Value

A parser.

Pseudocode

```
(p1 %then% p2)(x):
  if p1(x)==[] or x==null then fail()(x)
  else
    if p2(x[-1])==[] then fail()(x)
    else succeed([p1(x)$L, p2(x[-1])$L])(x[-2])
```

where `null` is the empty vector, `x[-1]` and `x[-2]` are the vector `x` without the first element and without the first two elements, respectively.

See Also

The discarding versions %xthen% and %thenx%

Examples

```
starts_with_a <- function(x) grepl("^a", x[1])
starts_with_b <- function(x) grepl("^b", x[1])
(satisfy(starts_with_a) %then% satisfy(starts_with_b))(c("ab", "bc", "de")) # success
(satisfy(starts_with_a) %then% satisfy(starts_with_b))(c("bb", "bc", "de")) # failure
(satisfy(starts_with_a) %then% satisfy(starts_with_b))(c("ab", "ac", "de")) # failure
```

%using% *Applying a function to the result of a parser*

Description

The %using% combinator allows us to manipulate results from a parser. The parser (p %using% f) has the same behavior as the parser p, except that the function f is applied to its result value.

Usage

```
p %using% f
```

Arguments

p	a parser.
f	a function to be applied to the result of a successful p.

Value

A parser.

Pseudocode

```
(p %using% f)(x):
  if p1(x)==[] then fail()(x)
  else succeed(f(p1(x)$L))(x[-1])
```

Examples

```
(literal("ab") %using% toupper)(c("ab", "cdef")) # success
(literal("ab") %using% toupper)(c("bb", "cdef")) # failure
```

%xthen% *Keeping only first or second result from a* %then% *sequence*

Description

Two parsers composed in sequence produce a pair of results. Sometimes we are only interested in one component of the pair. For example in the case of reserved words such as 'begin' and 'end'. In such cases, two special versions of the %then% combinator are useful, which keep either the first or second result, as reflected by the position of the letter 'x' in their names.

Usage

```
p1 %xthen% p2

p1 %thenx% p2
```

Arguments

p1, p2	two parsers.

Value

A parser.

Pseudocode

```
(p1 %xthen% p2)(x):
  if p1(x)==[] or x==null then fail()(x)
  else
    if p2(x[-1])==[] then fail()(x)
    else succeed(p1(x)$L)(x[-2])

(p1 %thenx% p2)(x):
  if p1(x)==[] or x==null then fail()(x)
  else
    if p2(x[-1])==[] then fail()(x)
    else succeed(p2(x[-1])$L)(x[-2])
```

where `null` is the empty vector, `x[-1]` and `x[-2]` are the vector `x` without the first element and without the first two elements, respectively.

See Also

%then%

Examples

```
is_number <- function(x) grepl("\\d+", x[1])
# Numbers are preceded by ">" symbols, but we only want the number
(literal(">") %thenx% satisfy(is_number))(c(">", "12"))
# Temperatures are followed by the unit 'C', but we only want the number
(satisfy(is_number) %xthen% literal("C"))(c("21", "C"))
```

Index

∗ **datasets**
    fastafile, 7
%thenx% *(%xthen%)*, 23
%or%, 20
%ret%, 21
%then%, 22, *24*
%thenx%, *22*
%using%, *21*, 23
%xthen%, *22*, 23

base::strsplit(), *3*
by_split, 2
by_split(), *4*
by_symbol, 3
by_symbol(), *3*

clear_store, 4

EmptyLine, 4
eof, 5
eof(), *7*, *13*
exactly *(*zero_or_more*)*, 18

fail *(*succeed*)*, 17
failed, 6
failed(), *10*, *12*
fastafile, 7
finished, 7

has_stored, 8

identity(), *16*
Ignore, 8

list_stored, 9
literal, 9

match_n *(*zero_or_more*)*, 18
match_s, 10
match_s(), *16*
MaybeEmpty *(*EmptyLine*)*, 4

named, 11

one_or_more *(*zero_or_more*)*, 18

print.marker, 12
print.marker(), *6*, *17*

regexec(), *16*
regular expression, *2*
reporter, 13
reporter(), *12*, *17*
retrieve *(*store*)*, 15

satisfy, 14
Spacer *(*EmptyLine*)*, 4
store, 15
stringparser, 16
stringr::str_match(), *16*
succeed, 17
succeed(), *5*, *13*

zero_or_more, 18
zero_or_one *(*zero_or_more*)*, 18