Package 'nc'

May 2, 2023

Maintainer Toby Dylan Hocking <toby.hocking@r-project.org>

Author Toby Dylan Hocking

Version 2023.5.1

License GPL-3

Title Named Capture to Data Tables

Description User-friendly functions for extracting a data table (row for each match, column for each group) from non-tabular text data using regular expressions, and for melting columns that match a regular expression. Patterns are defined using a readable syntax that makes it easy to build complex patterns in terms of simpler, re-usable sub-patterns. Named R arguments are translated to column names in the output; capture groups without names are used internally in order to provide a standard interface to three regular expression C libraries (PCRE, RE2, ICU). Output can also include numeric columns via user-specified type conversion functions.

Depends R (>= 2.14)

Imports data.table (>= 1.14.8)

Suggests testthat, re2, stringi, ggplot2, tidyr (>= 1.0.0), cdata, reshape2, knitr, markdown, R.utils, covr, arrow

VignetteBuilder knitr

URL https://github.com/tdhock/nc

BugReports https://github.com/tdhock/nc/issues

NeedsCompilation no

Repository CRAN

Date/Publication 2023-05-01 23:10:03 UTC

R topics documented:

`alternatives` *alternatives*

Description

Make a pattern that matches one of the specified alternatives. The `altlist` function can be helpful for defining named sub-patterns that are used in several alternatives.

Usage

```
alternatives(...)
```

Arguments

`...`	Each argument is a different alternative pattern.

Value

Pattern list.

Author(s)

Toby Dylan Hocking

Examples

```
## simple example.
subject <- c("foooo1", "barrr2")
str(foo.or.bar <- nc::alternatives(bar="bar+", foo="fo+"))
str(foo.or.bar <- list(bar="bar+", "|", foo="fo+"))#same
nc::capture_first_vec(subject, foo.or.bar, number="[12]")

## More complicated regular expression for matching the JobID column
## of SLURM sacct output.
JobID <- c(
  "13937810_25", "13937810_25.batch",
  "13937810_25.extern", "14022192_[1-3]", "14022204_[4]")
int.pattern <- list("[0-9]+", as.integer)
## Match the whole range inside square brackets.
range.pattern <- list(
  "[[]",
  task.start=int.pattern,
  nc::quantifier("-", task.end=int.pattern, "?"),
  "[]]")
nc::capture_first_vec(JobID, range.pattern, nomatch.error=FALSE)

## Match either a single task ID or a range, after an underscore.
task.pattern <- list(job="[0-9]+", "_", nc::alternatives(
  task.id=int.pattern,
  range.pattern))
nc::capture_first_vec(JobID, task.pattern)
```

`alternatives_with_shared_groups` *alternatives with shared groups*

Description

Create a pattern which matches `alternatives` with common sub-pattern groups.

Usage

```
alternatives_with_shared_groups(...)
```

Arguments

...	named arguments are sub-pattern groups, un-named arguments are alternative patterns which can refer to argument names.

Value

Pattern created by first using `altlist` on named arguments to create a list of sub-patterns, then using `alternatives` on un-named arguments (evaluated using the names defined in the sub-pattern list).

Author(s)

Toby Dylan Hocking

Examples

```
## Example 1: matching dates in different formats, but always same
## type in each alternative.
subject.vec <- c("mar 17, 1983", "26 sep 2017", "17 mar 1984")
## Another way is with alternative regular expressions.
pattern <- nc::alternatives_with_shared_groups(
  month="[a-z]{3}",
  day="[0-9]{2}",
  year="[0-9]{4}",
  list(month, " ", day, ", ", year),
  list(day, " ", month, " ", year))
match.dt <- nc::capture_first_vec(subject.vec, pattern)
print(match.dt, class=TRUE)
match.dt[, data.table::as.IDate(paste0(month, day, year), format="%b%d%Y")]

## Example 2: matching dates in different formats, but with
## different types in different alternatives.
subject.vec <- c("3/17/1983", "26 sep 2017")
month2int <- c(#this approach is locale-indepdendent.
  jan=1L, feb=2L, mar=3L, apr=4L,  may=5L,  jun=6L,
  jul=7L, aug=8L, sep=9L, oct=10L, nov=11L, dec=12L)
pattern <- nc::alternatives_with_shared_groups(
  day=list("[0-9]{2}", as.integer),
  year=list("[0-9]{4}", as.integer),
  list(month="[0-9]", as.integer, "/", day, "/", year),
  list(day, " ", month="[a-z]{3}", function(m)month2int[m], " ", year))
match.dt <- nc::capture_first_vec(subject.vec, pattern)
print(match.dt, class=TRUE)

## Example 3: three alternatives with four groups each.
subject.vec <- c(
  "EUR 5.00 Theft in delivery inserted in wire transfer 11/02/2021",
  "EUR 50.00 - Refund for theft in delivery - 30/07/2020",
  "EUR68.50 - Refund for theft in delivery 02/07/2020",
  "45.00 EUR 29/10/2020 Refund for theft in delivery",
  "53.00\u20ac Refund for theft in delivery 24/09/2020")
```

```
sep <- function(x, y, ...){
  if(missing(y)){
    list("^", x, "$")
  }else{
    sep(list(x, list(" - | |"), y), ...)
  }
}
pattern <- nc::alternatives_with_shared_groups(
  currency="EUR|\u20ac",
  amount=list("[0-9.]+", as.numeric),
  reason="[A-Za-z ]+?",
  date=list(
    "[0-9]{2}/[0-9]{2}/[0-9]{4}",
    function(d)data.table::as.IDate(d, format="%d/%m/%Y")),
  sep(currency, amount, reason, date),
  sep(amount, currency, date, reason),
  sep(amount, currency, reason, date))
match.dt <- nc::capture_first_vec(subject.vec, pattern)
print(match.dt, class=TRUE)
```

`altlist` *altlist*

Description

Create a named list containing named patterns, useful for creating a named list of named sub-pattern groups to be used with `alternatives`. Instead of using this function directly, most users should try `alternatives_with_shared_groups` which is more user-friendly and covers the most common use cases. You should only have to use altlist directly if your `alternatives` also have names (see examples).

Usage

```
altlist(...)
```

Arguments

`...`	Named patterns which will be used as sub-pattern groups in `alternatives`.

Value

Named list of patterns to be used for constructing `alternatives` using `with`, see examples.

Author(s)

Toby Dylan Hocking

Examples

```
## Example 1: matching dates in different formats, but always same
## type in each alternative.
subject.vec <- c("mar 17, 1983", "26 sep 2017", "17 mar 1984")
pat.list <- nc::altlist(month="[a-z]{3}", day="[0-9]{2}", year="[0-9]{4}")
pattern <- with(pat.list, nc::alternatives(
  american=list(month, " ", day, ", ", year),
  european=list(day, " ", month, " ", year)))
match.dt <- nc::capture_first_vec(subject.vec, pattern)
print(match.dt, class=TRUE)
match.dt[, data.table::as.IDate(paste0(month, day, year), format="%b%d%Y")]

## Example 2: matching dates in different formats, but with
## different types in different alternatives.
subject.vec <- c("3/17/1983", "26 sep 2017")
month2int <- c(#this approach is locale-indepdendent.
  jan=1L, feb=2L, mar=3L, apr=4L,  may=5L,  jun=6L,
  jul=7L, aug=8L, sep=9L, oct=10L, nov=11L, dec=12L)
pat.list <- nc::altlist(
  day=list("[0-9]{2}", as.integer),
  year=list("[0-9]{4}", as.integer))
pattern <- with(pat.list, nc::alternatives(
  american=list(month="[0-9]", as.integer, "/", day, "/", year),
  european=list(
    day, " ", month="[a-z]{3}", function(m)month2int[m], " ", year)))
match.dt <- nc::capture_first_vec(subject.vec, pattern)
print(match.dt, class=TRUE)
```

`apply_type_funs` *apply type funs*

Description

Convert columns of `match.mat` using corresponding functions from `fun.list`, then handle any duplicate capture `group` names.

Usage

```
apply_type_funs(match.mat,
    fun.list)
```

Arguments

`match.mat`	Character matrix (matches X groups).
`fun.list`	Named list of functions to apply to captured groups. If there are any duplicate names, they must be in `alternatives` (only one match per unique `group` name, otherwise error).

Value

data.table with columns defined by calling the functions in `fun.list` on the corresponding column of `match.mat`. Even if `fun.list` has duplicated names, the output data.table will have unique column names (identically named capture groups in `alternatives` will be combined into a single output column).

Author(s)

Toby Dylan Hocking

---

capture_all_str *Capture all matches in a single subject string*

Description

Capture each match of a regex pattern from one multi-line subject string or text file. It can be used to convert any regular text file (web page, log, etc) to a data table, see examples.

Usage

```
capture_all_str(...,
    engine = getOption("nc.engine",
        "PCRE"), collapse = "\n")
```

Arguments

`...`	subject, name1=pattern1, fun1, etc. The first argument must be a subject character vector (or file name which is read via `readLines` to get a subject). After removing missing values, we use `paste` to `collapse` the subject (by default using newline) and treat it as single character string to search. Arguments after the first specify the regex/conversion and must be string/function/list, as documented in `capture_first_vec`.
`engine`	character string, one of PCRE, ICU, RE2
`collapse`	separator string for combining elements of subject into a single string, used as `collapse` argument of `paste`.

Value

data.table with one row for each match, and one column for each capture `group`.

Author(s)

Toby Dylan Hocking

Examples

```
chr.pos.vec <- c(
  "chr10:213,054,000-213,055,000",
  "chrM:111,000-222,000",
  "this will not match",
  NA, # neither will this.
  "chr1:110-111 chr2:220-222") # two possible matches.
keep.digits <- function(x)as.integer(gsub("[^0-9]", "", x))
## By default elements of subject are treated as separate lines (and
## NAs are removed). Named arguments are used to create capture
## groups, and conversion functions such as keep.digits are used to
## convert the previously named group.
int.pattern <- list("[0-9,]+", keep.digits)
(match.dt <- nc::capture_all_str(
  chr.pos.vec,
  chrom="chr.*?",
  ":",
  chromStart=int.pattern,
  "-",
  chromEnd=int.pattern))
str(match.dt)

## Extract all fields from each alignment block, using two regex
## patterns, then dcast.
info.txt.gz <- system.file(
  "extdata", "SweeD_Info.txt.gz", package="nc")
info.vec <- readLines(info.txt.gz)
info.vec[24:40]
info.dt <- nc::capture_all_str(
  sub("Alignment ", "//", info.vec),
  "//",
  alignment="[0-9]+",
  fields="[^/]+")
(fields.dt <- info.dt[, nc::capture_all_str(
  fields,
  "\t+",
  variable="[^:]+",
  ":\t*",
  value=".*"),
  by=alignment])
(fields.wide <- data.table::dcast(fields.dt, alignment ~ variable))

## Capture all csv tables in report -- the file name can be given as
## the subject to nc::capture_all_str, which calls readLines to get
## data to parse.
(report.txt.gz <- system.file(
  "extdata", "SweeD_Report.txt.gz", package="nc"))
(report.dt <- nc::capture_all_str(
  report.txt.gz,
  "//",
  alignment="[0-9]+",
```

```
  "\n",
  csv="[^/]+"
)[, {
  data.table::fread(text=csv)
}, by=alignment])

## Join report with info fields.
report.dt[fields.wide, on=.(alignment)]

## parsing nbib citation file.
(pmc.nbib <- system.file(
  "extdata", "PMC3045577.nbib", package="nc"))
blank <- "\n      "
pmc.dt <- nc::capture_all_str(
  pmc.nbib,
  Abbreviation="[A-Z]+",
  " *- ",
  value=list(
    ".*",
    list(blank, ".*"), "*"),
  function(x)sub(blank, "", x))
str(pmc.dt)

## What do the variable fields mean? It is explained on
## https://www.nlm.nih.gov/bsd/mms/medlineelements.html which has a
## local copy in this package (downloaded 18 Sep 2019).
fields.html <- system.file(
  "extdata", "MEDLINE_Fields.html", package="nc")
if(interactive())browseURL(fields.html)
fields.vec <- readLines(fields.html)

## It is pretty easy to capture fields and abbreviations if gsub
## used to remove some tags first.
no.strong <- gsub("</?strong>", "", fields.vec)
no.comments <- gsub("<!--.*?-->", "", no.strong)
## grep then capture_first_vec can be used if each desired row in
## the output comes from a single line of the input file.
(h3.vec <- grep("<h3", no.comments, value=TRUE))
h3.pattern <- list(
  nc::field("name", '="', '[^"]+'),
  '"></a>',
  fields.abbrevs="[^<]+")
first.fields.dt <- nc::capture_first_vec(
  h3.vec, h3.pattern)
field.abbrev.pattern <- list(
  Field=".*?",
  " \\(",
  Abbreviation="[^)]+",
  "\\)",
  "(?: and |$)?")
(first.each.field <- first.fields.dt[, nc::capture_all_str(
  fields.abbrevs, field.abbrev.pattern),
  by=fields.abbrevs])
```

```
## If we want to capture the information after the initial h3 line
## of the input, e.g. the rest column below which contains a
## description/example for each field, then capture_all_str can be
## used on the full input file.
h3.fields.dt <- nc::capture_all_str(
  no.comments,
  h3.pattern,
  '</h3>\n',
  rest="(?:.*\n)+?", #exercise: get the examples.
  "<hr />\n")
(h3.each.field <- h3.fields.dt[, nc::capture_all_str(
  fields.abbrevs, field.abbrev.pattern),
  by=fields.abbrevs])

## Either method of capturing abbreviations gives the same result.
identical(first.each.field, h3.each.field)

## but the capture_all_str method returns the additional rest column
## which contains data after the initial h3 line.
names(first.fields.dt)
names(h3.fields.dt)
cat(h3.fields.dt[fields.abbrevs=="Volume (VI)", rest])

## There are 66 Field rows across three tables.
a.href <- list('<a href=[^>]+>')
(td.vec <- fields.vec[240:280])
fields.pattern <- list(
  "<td.*?>",
  a.href,
  Fields="[^()<]+",
  "</a></td>\n")
(td.only.Fields <- nc::capture_all_str(fields.vec, fields.pattern))

## Extract Fields and Abbreviations. Careful: most fields have one
## abbreviation, but one field has none, and two fields have two.
(td.fields.dt <- nc::capture_all_str(
  fields.vec,
  fields.pattern,
  "<td[^>]*>",
  "(?:\n<div>)?",
  a.href, "?",
  abbrevs=".*?",
  "</"))

## Get each individual abbreviation from the previously captured td
## data.
td.each.field <- td.fields.dt[, {
  f <- nc::capture_all_str(
    Fields,
    Field=".*?",
    "(?:$| and )")
  a <- nc::capture_all_str(
```

```
    abbrevs,
    "\\(",
    Abbreviation="[^)]+",
    "\\)")
  if(nrow(a)==0)list() else cbind(f, a)
}, by=Fields]
str(td.each.field)
td.each.field[td.fields.dt, .(
  count=.N
), on=.(Fields), by=.EACHI][order(count)]

## There is a typo in the data captured from the h3 headings.
td.each.field[!Field %in% h3.each.field$Field]
h3.each.field[!Field %in% td.each.field$Field]

## Abbreviations are consistent.
td.each.field[!Abbreviation %in% h3.each.field$Abbreviation]
h3.each.field[!Abbreviation %in% td.each.field$Abbreviation]

## There is a a table that provides a description of each comment
## type.
(comment.vec <- fields.vec[840:860])
comment.dt <- nc::capture_all_str(
  fields.vec,
  "<td><strong>",
  Field="[^<]+",
  "</strong></td>\n",
  "<td><strong>\\(",
  Abbreviation="[^)]+",
  "\\)</strong></td>\n",
  "<td>",
  description=".*",
  "</td>\n")
str(comment.dt)

## Join to original PMC citation file in order to see what the
## abbreviations used in that file mean.
all.abbrevs <- rbind(
  td.each.field[, .(Field, Abbreviation)],
  comment.dt[, .(Field, Abbreviation)])
all.abbrevs[pmc.dt, .(
  Abbreviation,
  Field,
  value=substr(value, 1, 20)
), on=.(Abbreviation)]

## There is a listing of examples for each comment type.
(comment.ex.dt <- nc::capture_all_str(
  fields.vec[938],
  "br />\\s*",
  Abbreviation="[A-Z]+",
  "\\s*-\\s*",
  citation="[^<]+?",
```

```
  list(
    "[.] ",
    nc::field("PMID", ": ", "[0-9]+")
  ), "?",
  "<"))

## Join abbreviations to see what kind of comments.
all.abbrevs[comment.ex.dt, on=.(Abbreviation)]

## parsing bibtex file.
refs.bib <- system.file(
  "extdata", "namedCapture-refs.bib", package="nc")
refs.vec <- readLines(refs.bib)
at.lines <- grep("@", refs.vec, value=TRUE)
str(at.lines)
refs.dt <- nc::capture_all_str(
  refs.vec,
  "@",
  type="[^{]+",
  "[{]",
  ref="[^,]+",
  ",\n",
  fields="(?:.*\n)+?.*",
  "[}]\\s*(?:$|\n)")
str(refs.dt)

## parsing each field of each entry.
eq.lines <- grep("=", refs.vec, value=TRUE)
str(eq.lines)
strip <- function(x)sub("^\\s*\\{*", "", sub("\\}*,?$", "", x))
refs.fields <- refs.dt[, nc::capture_all_str(
  fields,
  "\\s+",
  variable="\\S+",
  "\\s+=",
  value=".*", strip),
  by=.(type, ref)]
str(refs.fields)
with(refs.fields[ref=="HockingUseR2011"], structure(
  as.list(value), names=variable))
## the URL of my talk is now
## https://user2011.r-project.org/TalkSlides/Lightening/2-StatisticsAndProg_3-Hocking.pdf

if(!grepl("solaris", R.version$platform)){#To avoid CRAN check error on solaris
  ## Parsing wikimedia tables: each begins with {| and ends with |}.
  emoji.txt.gz <- system.file(
    "extdata", "wikipedia-emoji-text.txt.gz", package="nc")
  tables <- nc::capture_all_str(
    emoji.txt.gz,
    "\n[{][|]",
    first=".*",
    '\n[|][+] style="',
    nc::field("font-size", ":", '.*?'),
```

```
    '" [|] ',
    title=".*",
    lines="(?:\n.*)*?",
    "\n[|][}]")
  str(tables)
  ## Rows are separated by |-
  rows.dt <- tables[, {
    row.vec <- strsplit(lines, "|-", fixed=TRUE)[[1]][-1]
    .(row.i=seq_along(row.vec), row=row.vec)
  }, by=title]
  str(rows.dt)
  ## Try to parse columns from each row. Doesn't work for second table
  ## https://en.wikipedia.org/w/index.php?title=Emoji&oldid=920745513#Skin_color
  ## because some entries have rowspan=2.
  contents.dt <- rows.dt[, nc::capture_all_str(
    row,
    "[|] ",
    content=".*?",
    "(?: [|]|\n|$)"),
    by=.(title, row.i)]
  contents.dt[, .(cols=.N), by=.(title, row.i)]
  ## Make data table from
 ## https://en.wikipedia.org/w/index.php?title=Emoji&oldid=920745513#Emoji_versus_text_presentation
  contents.dt[, col.i := 1:.N, by=.(title, row.i)]
  data.table::dcast(
    contents.dt[title=="Sample emoji variation sequences"],
    row.i ~ col.i,
    value.var="content")
}

## Simple way to extract code chunks from Rmd.
vignette.Rmd <- system.file(
  "extdata", "vignette.Rmd", package="nc")
non.greedy.lines <- list(
  list(".*\n"), "*?")
optional.name <- list(
  list(" ", name="[^,}]+"), "?")
Rmd.dt <- nc::capture_all_str(
  vignette.Rmd,
  before=non.greedy.lines,
  "```\\{r",
  optional.name,
  parameters=".*",
  "\\}\n",
  code=non.greedy.lines,
  "```")
Rmd.dt[, chunk := 1:.N]
Rmd.dt[, .(chunk, name, parameters, some.code=substr(code, 1, 20))]

## Extract individual parameter names and values.
Rmd.dt[, nc::capture_all_str(
  parameters,
  ", *",
```

```
  variable="[^= ]+",
  " *= *",
  value="[^ ,]+"),
  by=chunk]

## Simple way to extract code chunks from Rnw.
vignette.Rnw <- system.file(
  "extdata", "vignette.Rnw", package="nc")
Rnw.dt <- nc::capture_all_str(
  vignette.Rnw,
  before=non.greedy.lines,
  "<<",
  name="[^,>]*",
  parameters=".*",
  ">>=\n",
  code=non.greedy.lines,
  "@")
Rnw.dt[, .(name, parameters, some.code=substr(code, 1, 20))]

## The next example involves timing some compression programs that
## were run on a 159 megabyte input/uncompressed text file. Here is
## how to get a data table from the time command line output.
times.out <- system.file(
  "extdata", "compress-times.out", package="nc", mustWork=TRUE)
times.dt <- nc::capture_all_str(
  times.out,
  "coverage.bedGraph ",
  program=".*?",
  " coverage.bedGraph.",
  suffix=".*",
  "\n\nreal\t",
  minutes.only="[0-9]+", as.numeric,
  "m",
  seconds.only="[0-9.]+", as.numeric)
times.dt[, seconds := minutes.only*60+seconds.only]
times.dt

## join with output from du command line program.
sizes.out <- system.file(
  "extdata", "compress-sizes.out", package="nc", mustWork=TRUE)
sizes.dt <- data.table::fread(
  file=sizes.out,
  col.names=c("megabytes", "file"))
sizes.dt[, suffix := sub("coverage.bedGraph.?", "", file)]
join.dt <- times.dt[sizes.dt, on="suffix"][order(megabytes)]
join.dt[file=="coverage.bedGraph", seconds := 0]
join.dt

## visualize with ggplot2.
if(require(ggplot2)){
  ggplot(join.dt, aes(
    seconds, megabytes, label=suffix))+
    geom_text(vjust=-0.5)+
```

```
    geom_point()+
    scale_x_log10()+
    scale_y_log10()
}
```

---

capture_first_df *Capture first match in columns of a data frame*

Description

Capture first matching text from one or more character columns of a data frame, using a different regular expression for each column.

Usage

```
capture_first_df(...,
    nomatch.error = getOption("nc.nomatch.error",
        TRUE), existing.error = getOption("nc.existing.error",
        TRUE), engine = getOption("nc.engine",
        "PCRE"))
```

Arguments

...	subject data frame, colName1=list(groupName1=pattern1, fun1, etc), colName2=list(etc), etc. First argument must be a data frame with one or more character columns of subjects for matching. If the first argument is a data table then it will be modified using set (for memory efficiency, to avoid copying the whole data table); otherwise the input data frame will be copied to a new data table. Each other argument must be named using a column name of the subject data frame, e.g. colName1, colName2. Each other argument value must be a list that specifies the regex/conversion to use (in string/function/list format as documented in capture_first_vec, which is used on each named column), and possibly a column-specific engine to use.
nomatch.error	if TRUE (default), stop with an error if any subject does not match; otherwise subjects that do not match are reported as missing/NA rows of the result.
existing.error	if TRUE (default to avoid data loss), stop with an error if any capture groups have the same name as an existing column of subject.
engine	character string, one of PCRE, ICU, RE2. This engine will be used for each column, unless another engine is specified for that column in ...

Value

data.table with same number of rows as subject, with an additional column for each named capture group specified in ...

Author(s)

Toby Dylan Hocking

Examples

```
## The JobID column can be match with a complicated regular
## expression, that we will build up from small sub-pattern list
## variables that are easy to understand independently.
(sacct.df <- data.frame(
  JobID = c(
    "13937810_25", "13937810_25.batch",
    "13937810_25.extern", "14022192_[1-3]", "14022204_[4]"),
  Elapsed = c(
    "07:04:42", "07:04:42", "07:04:49",
    "00:00:00", "00:00:00"),
  stringsAsFactors=FALSE))

## Just match the end of the range.
int.pattern <- list("[0-9]+", as.integer)
end.pattern <- list(
  "-",
  task.end=int.pattern)
nc::capture_first_df(sacct.df, JobID=list(
  end.pattern, nomatch.error=FALSE))

## Match the whole range inside square brackets.
range.pattern <- list(
  "[[]",
  task.start=int.pattern,
  end.pattern, "?", #end is optional.
  "[]]")
nc::capture_first_df(sacct.df, JobID=list(
  range.pattern, nomatch.error=FALSE))

## Match either a single task ID or a range, after an underscore.
task.pattern <- list(
  "_",
  list(
    task.id=int.pattern,
    "|",#either one task(above) or range(below)
    range.pattern))
nc::capture_first_df(sacct.df, JobID=task.pattern)

## Match type suffix alone.
type.pattern <- list(
  "[.]",
  type=".*")
nc::capture_first_df(sacct.df, JobID=list(
  type.pattern, nomatch.error=FALSE))

## Match task and optional type suffix.
```

```
task.type.pattern <- list(
  task.pattern,
  type.pattern, "?")
nc::capture_first_df(sacct.df, JobID=task.type.pattern)

## Match full JobID and Elapsed columns.
nc::capture_first_df(
  sacct.df,
  JobID=list(
    job=int.pattern,
    task.type.pattern),
  Elapsed=list(
    hours=int.pattern,
    ":",
    minutes=int.pattern,
    ":",
    seconds=int.pattern))

## If input is data table then it is modified for memory efficiency,
## to avoid copying entire table.
sacct.DT <- data.table::as.data.table(sacct.df)
nc::capture_first_df(sacct.df, JobID=task.pattern)
sacct.df #not modified.
nc::capture_first_df(sacct.DT, JobID=task.pattern)
sacct.DT #modified!
```

`capture_first_glob` *capture first glob*

Description

Glob files, then use `capture_first_vec` to get meta-data from each file name, and combine with contents of each file.

Usage

```
capture_first_glob(glob,
    ..., READ = fread)
```

Arguments

`glob`	string: `glob` specifying files to read.
`...`	pattern passed to `capture_first_vec`, used to get meta-data from file names.
`READ`	function of one argument (file name) which returns a data table, default `fread`.

Value

Data table with columns of meta-data specified by pattern, plus contents of all files specified by `glob`.

Author(s)

Toby Dylan Hocking

Examples

```
## Example 1: simple pattern.
db <- system.file("extdata/chip-seq-chunk-db", package="nc", mustWork=TRUE)
glob <- paste0(db, "/*/*/counts/*")
read.bedGraph <- function(f)data.table::fread(
  f, skip=1, col.names = c("chrom","start", "end", "count"))
data.chunk.pattern <- list(
  data="H.*?",
  "/",
  chunk="[0-9]+", as.integer)
(data.chunk.dt <- nc::capture_first_glob(glob, data.chunk.pattern, READ=read.bedGraph))

## Write same data set in Hive partition, then re-read.
if(requireNamespace("arrow")){
  path <- tempfile()
  arrow::write_dataset(
    dataset=data.chunk.dt,
    path=path,
    format="csv",
    partitioning=c("data","chunk"),
    max_rows_per_file=1000)
  hive.glob <- file.path(path, "*", "*", "*.csv")
  hive.pattern <- list(
    nc::field("data","=",".*?"),
    "/",
    nc::field("chunk","=",".*?", as.integer),
    "/",
    nc::field("part","-","[0-9]+", as.integer))
  hive.dt <- nc::capture_first_glob(hive.glob, hive.pattern)
  hive.dt[, .(rows=.N), by=.(data,chunk,part)]
}

## Example 2: more complex pattern.
count.dt <- nc::capture_first_glob(
  glob,
  data.chunk.pattern,
  "/counts/",
  name=list("McGill", id="[0-9]+", as.integer),
  ".bedGraph.gz",
  READ=read.bedGraph)
count.dt[, .(count=.N), by=.(data, chunk, name, chrom)]

if(require(ggplot2)){
  ggplot()+
    facet_wrap(~data+chunk+name+chrom, labeller=label_both, scales="free")+
    geom_step(aes(
      start, count),
```

```
      data=count.dt)
}

## Example 3: parsing non-CSV data.
vignettes <- system.file("extdata/vignettes", package="nc", mustWork=TRUE)
non.greedy.lines <- list(
  list(".*\n"), "*?")
optional.name <- list(
  list(" ", chunk_name="[^,}]+"), "?")
chunk.pattern <- list(
  before=non.greedy.lines,
  "```\\{r",
  optional.name,
  parameters=".*",
  "\\}\n",
  code=non.greedy.lines,
  "```")
chunk.dt <- nc::capture_first_glob(
  paste0(vignettes, "/*.Rmd"),
  "/v",
  vignette_number="[0-9]", as.integer,
  "-",
  vignette_name=".*?",
  ".Rmd",
  READ=function(f)nc::capture_all_str(f, chunk.pattern))
chunk.dt[, chunk_number := seq_along(chunk_name), by=vignette_number]
chunk.dt[, .(
  vignette_number, vignette_name, chunk_number, chunk_name,
  lines=nchar(code))]
cat(chunk.dt$code[2])
```

`capture_first_vec` *Capture first match in each character vector element*

Description

Use a regular expression (regex) with capture groups to extract the first matching text from each of several subject strings. For all matches in one multi-line text file or string use `capture_all_str`. For the first match in every row of a data.frame, using a different regex for each column, use `capture_first_df`. For reading regularly named files, use `capture_first_glob`. For matching column names in a wide data frame and then melting/reshaping those columns to a taller/longer data frame, see `capture_melt_single` and `capture_melt_multiple`. To simplify the definition of the regex you can use `field`, `quantifier`, and `alternatives`.

Usage

```
capture_first_vec(...,
    nomatch.error = getOption("nc.nomatch.error",
        TRUE), engine = getOption("nc.engine",
        "PCRE"))
```

Arguments

...	subject, name1=pattern1, fun1, etc. The first argument must be a character vector of length>0 (subject strings to parse with a regex). Arguments after the first specify the regex/conversion and must be string/list/function. All character strings are pasted together to obtain the final regex used for matching. Each string/list with a named argument in R becomes a capture group in the regex, and the name is used for the corresponding column of the output data table. Each function must be un-named, and is used to convert the previous capture group. Each un-named list becomes a non-capturing group. Elements in each list are parsed recursively using these rules.
nomatch.error	if TRUE (default), stop with an error if any subject does not match; otherwise subjects that do not match are reported as missing/NA rows of the result.
engine	character string, one of PCRE, ICU, RE2

Value

data.table with one row for each subject, and one column for each capture group.

Author(s)

Toby Dylan Hocking

Examples

```
chr.pos.vec <- c(
  "chr10:213,054,000-213,055,000",
  "chrM:111,000",
  "chr1:110-111 chr2:220-222") # two possible matches.
## Find the first match in each element of the subject character
## vector. Named argument values are used to create capture groups
## in the generated regex, and argument names become column names in
## the result.
(dt.chr.cols <- nc::capture_first_vec(
  chr.pos.vec,
  chrom="chr.*?",
  ":",
  chromStart="[0-9,]+"))

## Even when no type conversion functions are specified, the result
## is always a data.table:
str(dt.chr.cols)

## Conversion functions are used to convert the previously named
## group, and patterns may be saved in lists for re-use.
keep.digits <- function(x)as.integer(gsub("[^0-9]", "", x))
int.pattern <- list("[0-9,]+", keep.digits)
range.pattern <- list(
  chrom="chr.*?",
  ":",
```

```
  chromStart=int.pattern,
  list( # un-named list becomes non-capturing group.
    "-",
    chromEnd=int.pattern
  ), "?") # chromEnd is optional.
(dt.int.cols <- nc::capture_first_vec(
  chr.pos.vec, range.pattern))

## Conversion functions used to create non-char columns.
str(dt.int.cols)

## NA used to indicate no match or missing subject.
na.vec <- c(
  "this will not match",
  NA, # neither will this.
  chr.pos.vec)
nc::capture_first_vec(na.vec, range.pattern, nomatch.error=FALSE)
```

`capture_longer_spec` *capture longer spec*

Description

Create a spec `data` table for input to `pivot_longer_spec`.

Usage

```
capture_longer_spec(data,
    ..., values_to = "value")
```

Arguments

`data`	Data table to reshape (actually the `data` are ignored, and only the column names are used).
`...`	Regex and conversion as described in `capture_first_vec`. This is processed by `measure` so if "column" is used as an argument name then there will be multiple output columns in the reshaped `data` (analogous to names_to=".value" in `pivot_longer`).
`values_to`	string to use for name of value column in output (only used if there is a single output column in the reshaped `data`).

Value

`data` table describing a reshape longer operation.

Author(s)

Toby Dylan Hocking

Examples

```
if("measure" %in% ls(asNamespace("data.table"))){
  (one.iris <- iris[1,])
 (single.spec <- nc::capture_longer_spec(iris, part=".*", "[.]", dim=".*", values_to="cm"))
  (multiple.spec <- nc::capture_longer_spec(iris, part=".*", "[.]", column=".*"))
  if(requireNamespace("tidyr")){
    tidyr::pivot_longer_spec(one.iris, single.spec)
    tidyr::pivot_longer_spec(one.iris, multiple.spec)
  }
}
```

---

capture_melt_multiple *Capture and melt into multiple columns*

Description

Match a regex to column names of a wide data frame (many columns/few rows), then melt/reshape the matching columns into multiple result columns in a taller/longer data table (fewer columns/more rows). Input should be a data frame with four or more regularly named columns of possibly different types to reshape, and output is a data table with at least two columns of reshaped data. For melting into a single result column, see capture_melt_single.

Usage

```
capture_melt_multiple(...,
    fill = FALSE, na.rm = FALSE,
    verbose = getOption("datatable.verbose"))
```

Arguments

...	First argument must be a data frame to melt/reshape; column names of this data frame will be used as the subjects for regex matching. Other arguments (regex/conversion/engine) are passed to capture_first_vec along with no-match.error=FALSE. The regex must define a group named "column" – each unique value captured in this group becomes a column name for the reshaped data in the output. There must also be at least one other group, and the output will contain a column for each other group – see examples.
fill	If TRUE, fill missing input reshape columns with runs of rows with missing values in the output reshape columns. Otherwise stop with an error (default).
na.rm	Remove missing values from melted data? (passed to melt.data.table)
verbose	Print verbose output messages? (passed to melt.data.table)

Value

Data table of reshaped/melted/tall/long data, with a new column for each unique value of the capture group named "column", and a new column for each other capture group.

Author(s)

Toby Dylan Hocking

Examples

```
## Example 1: melt iris columns to compare Sepal and Petal dims, as
## in cdata package, https://winvector.github.io/cdata/
(iris.part.cols <- nc::capture_melt_multiple(
  iris,
  column=".*?",
  "[.]",
  dim=".*"))
iris.part.cols[Sepal<Petal] #Sepals are never smaller than Petals.
if(require("ggplot2")){
  ggplot()+
    theme_bw()+
    theme(panel.spacing=grid::unit(0, "lines"))+
    facet_grid(dim ~ Species)+
    coord_equal()+
    geom_abline(slope=1, intercept=0, color="grey")+
    geom_point(aes(
      Petal, Sepal),
      shape=1,
      data=iris.part.cols)
}

## Example 2. melt iris to Length and Width columns.
(iris.dim.cols <- nc::capture_melt_multiple(
  iris,
  part=".*?",
  "[.]",
  column=".*"))
iris.dim.cols[Length<Width] #Length is never less than Width.

## Example 3. Lots of column types, from example(melt.data.table).
set.seed(1)
DT <- data.table::data.table(
  i_1 = c(1:5, NA),
  i_2 = c(NA,6:10),
  f_1 = factor(sample(c(letters[1:3], NA), 6, TRUE)),
  f_2 = factor(c("z", "a", "x", "c", "x", "x"), ordered=TRUE),
  c_1 = sample(c(letters[1:3], NA), 6, TRUE),
  l_2 = list(NULL, NA, c(NA,NA), logical(), 1:2, TRUE),
  d_1 = as.Date(c(1:3,NA,4:5), origin="2013-09-01"),
  d_2 = as.Date(6:1, origin="2012-01-01"))
## nc syntax melts to three output columns of different types using
## a single regex (na.rm=FALSE by default in order to avoid losing
## information).
nc::capture_melt_multiple(
  DT,
  column="[dfi]",
```

```
  "_",
  number="[12]", as.integer)

## fill=TRUE means to output NA in positions that correspond to
## missing input columns (in this case, there is no l_1 nor c_2).
nc::capture_melt_multiple(
  DT,
  column=".*",
  "_",
  number="[12]", as.integer,
  fill=TRUE)

## Example 4, three children, one family per row, from data.table
## vignette.
family.dt <- data.table::fread(text="
family_id age_mother dob_child1 dob_child2 dob_child3 gender_child1 gender_child2 gender_child3
1         30 1998-11-26 2000-01-29         NA             1              2             NA
2         27 1996-06-22         NA         NA             2             NA             NA
3         26 2002-07-11 2004-04-05 2007-09-02             2              2              1
4         32 2004-10-10 2009-08-27 2012-07-21             1              1              1
5         29 2000-12-05 2005-02-28         NA             2              1             NA")
## nc::field can be used to define group name and pattern at the
## same time, to avoid repetitive code.
(children.nc <- nc::capture_melt_multiple(
  family.dt,
  column=".+",
  "_",
  nc::field("child", "", "[1-3]", as.integer),
  na.rm=TRUE))

## Example 5: wide data CSV with 100 possible peaks per row, each
## peak has three attributes (Allele, Height, Size) from
## https://lftdi.camden.rutgers.edu/repository/PROVEDIt_1-5-Person%20CSVs%20Filtered.zip
PROVEDIt.csv <- system.file(
  "extdata", "RD12-0002_PP16HS_5sec_GM_F_1P.csv.gz",
  package="nc", mustWork=TRUE)
PROVEDIt.wide <- data.table::fread(PROVEDIt.csv)
names(PROVEDIt.wide)
PROVEDIt.tall <- nc::capture_melt_multiple(
  PROVEDIt.wide,
  column=".*",
  " ",
  peak="[0-9]+", as.integer,
  na.rm=TRUE)
head(PROVEDIt.tall)

## plot number of peaks per row.
peaks.per.sample.marker <- PROVEDIt.tall[, .(
  peaks=.N
), by=.(`Sample File`, Marker)][order(peaks)]
if(require(ggplot2)){
  ggplot()+
    geom_histogram(aes(
```

```
      peaks),
      data=peaks.per.sample.marker,
      binwidth=1)
}

## which row has the most peaks?
(most <- PROVEDIt.tall[which.max(peak), .(`Sample File`, Marker, Dye)])
PROVEDIt.tall[most, on=names(most)]
PROVEDIt.wide[most, on=names(most)]
```

`capture_melt_single` *Capture and melt into a single column*

Description

Match a regex to column names of a wide data frame (many columns/few rows), then melt/reshape the matching columns into a single result column in a taller/longer data table (fewer columns/more rows). It is for the common case of melting several columns of the same type in a "wide" input data table which has several distinct pieces of information encoded in each column name. For melting into several result columns of possibly different types, see `capture_melt_multiple`.

Usage

```
capture_melt_single(...,
    value.name = "value",
    na.rm = TRUE, verbose = getOption("datatable.verbose"))
```

Arguments

`...`	First argument must be a data frame to melt/reshape; column names of this data frame will be used as the subjects for regex matching. Other arguments (regex/conversion/engine) are passed to `capture_first_vec` along with no-match.error=FALSE.
`value.name`	Name of the column in output which has values taken from melted/reshaped column values of input (passed to `melt.data.table`).
`na.rm`	remove missing values from melted data? (passed to `melt.data.table`)
`verbose`	Print `verbose` output messages? (passed to `melt.data.table`)

Value

Data table of reshaped/melted/tall/long data, with a new column for each named argument in the pattern, and additionally variable/value columns.

Author(s)

Toby Dylan Hocking

Examples

```
## Example 1: melt iris data and barplot for each numeric variable.
(iris.tall <- nc::capture_melt_single(
  iris,
  part=".*",
  "[.]",
  dim=".*",
  value.name="cm"))
## Histogram of cm for each variable.
if(require("ggplot2")){
  ggplot()+
    theme_bw()+
    theme(panel.spacing=grid::unit(0, "lines"))+
    facet_grid(part ~ dim)+
    geom_bar(aes(cm), data=iris.tall)
}

## Example 2: melt who data and use type conversion functions for
## year limits (e.g. for censored regression).
if(requireNamespace("tidyr")){
  data(who, package="tidyr", envir=environment())
  ##2.1 just extract diagnosis and gender to chr columns.
  new.diag.gender <- list(#save pattern as list for re-use later.
    "new_?",
    diagnosis=".*",
    "_",
    gender=".")
  who.tall.chr <- nc::capture_melt_single(who, new.diag.gender, na.rm=TRUE)
  print(head(who.tall.chr))
  str(who.tall.chr)
  ##2.2 also extract ages and convert to numeric output columns.
  who.tall.num <- nc::capture_melt_single(
    who,
    new.diag.gender,#previous pattern for matching diagnosis and gender.
    ages=list(#new pattern for matching age range.
      min.years="0|[0-9]{2}", as.numeric,#in-line type conversion functions.
      max.years="[0-9]{0,2}", function(x)ifelse(x=="", Inf, as.numeric(x))),
    value.name="count",
    na.rm=TRUE)
  print(head(who.tall.num))
  str(who.tall.num)
  ##2.3 compute total count for each age range then display the
  ##subset with max.years lower than a threshold.
  who.age.counts <- who.tall.num[, .(
    total=sum(count)
  ), by=.(min.years, max.years)]
  print(who.age.counts[max.years < 50])
}

## Example 3: pepseq data.
if(requireNamespace("R.utils")){#for reading gz files with data.table
```

```
  pepseq.dt <- data.table::fread(
    system.file("extdata", "pepseq.txt.gz", package="nc", mustWork=TRUE))
  u.pepseq <- pepseq.dt[, unique(names(pepseq.dt)), with=FALSE]
  nc::capture_melt_single(
    u.pepseq,
    "^",
    prefix=".*?",
    nc::field("D", "", ".*?"),
    "[.]",
    middle=".*?",
    "[.]",
    "[0-9]+",
    suffix=".*",
    "$")
}
```

`check_df_names` *check df names*

Description

Check that first argument is a data frame and then call `check_names` on its names.

Usage

```
check_df_names(...)
```

Arguments

```
...
```

Author(s)

Toby Dylan Hocking

`check_names` *check names*

Description

Check that `subject` is a vector of unique names and then call `capture_first_vec`.

Usage

```
check_names(subject,
    var.args)
```

Arguments

subject

var.args

Author(s)

Toby Dylan Hocking

---

collapse_some *collapse some*

Description

Create character string with some or all items.

Usage

```
collapse_some(all.vec,
    max.first.last = 5,
    collapse = ",")
```

Arguments

all.vec	Vector of all items.
max.first.last	Max number of items to show.
collapse	Passed to paste.

Value

Character string formed by paste with collapse on some items of all.vec (first/last few items used if length is greater than max.first.last*2, otherwise all items).

Author(s)

Toby Dylan Hocking

`field` *Capture a field*

Description

Capture a field with a pattern of the form list("field.name", `between.pattern`, field.name=list(...)) – see examples.

Usage

```
field(field.name, between.pattern,
    ...)
```

Arguments

`field.name`	Field name, used as a pattern and as a capture `group` (output column) name.
`between.pattern`	Pattern to match after `field.name` but before the field value.
`...`	Pattern(s) for matching field value.

Value

Pattern list which can be used in `capture_first_vec`, `capture_first_df`, or `capture_all_str`.

Author(s)

Toby Dylan Hocking

Examples

```
## Two ways to create the same pattern.
str(list("Alignment ", Alignment="[0-9]+"))
## To avoid typing Alignment twice use:
str(nc::field("Alignment", " ", "[0-9]+"))

## An example with lots of different fields.
info.txt.gz <- system.file(
  "extdata", "SweeD_Info.txt.gz", package="nc")
info.vec <- readLines(info.txt.gz)
info.vec[24:40]
## For each Alignment there are many fields which have a similar
## pattern, and occur in the same order. One way to capture these
## fields is by coding a pattern that says to look for all of those
## fields in that order. Each field is coded using this helper
## function.
g <- function(name, fun=identity, suffix=list()){
  list(
    "\t+",
```

```
    nc::field(name, ":\t+", ".*"),
    fun,
    suffix,
    "\n+")
}
nc::capture_all_str(
  info.vec,
  nc::field("Alignment", " ", "[0-9]+"),
  "\n+",
  g("Chromosome"),
  g("Sequences", as.integer),
  g("Sites", as.integer),
  g("Discarded sites", as.integer),
  g("Processing", as.integer, " seconds"),
  g("Position", as.integer),
  g("Likelihood", as.numeric),
  g("Alpha", as.numeric))

## Another example where field is useful.
trackDb.txt.gz <- system.file(
  "extdata", "trackDb.txt.gz", package="nc")
trackDb.vec <- readLines(trackDb.txt.gz)
cat(trackDb.vec[101:115], sep="\n")
int.pattern <- list("[0-9]+", as.integer)
 cell.sample.type <- list(
  cellType="[^ ]*?",
  "_",
  sampleName=list(
    "McGill",
    sampleID=int.pattern),
  dataType="Coverage|Peaks")
## Each block in the trackDb file begins with track, followed by a
## space, followed by the track name. That pattern is coded below,
## using field:
track.pattern <- nc::field(
  "track",
  " ",
  cell.sample.type,
  "|",
  "[^\n]+")
nc::capture_all_str(trackDb.vec, track.pattern)

## Each line in a block has the same structure (field name, space,
## field value). Below we use the field function to extract the
## color line, along with columns for each of the three channels
## (red, green, blue).
any.lines.pattern <- "(?:\n[^\n]+)*"
nc::capture_all_str(
  trackDb.vec,
  track.pattern,
  any.lines.pattern,
  "\\s+",
  nc::field(
```

```
  "color", " ",
  red=int.pattern, ",",
  green=int.pattern, ",",
  blue=int.pattern))
```

---

group *Capture group*

Description

Create a capture group (named column in output). In the vast majority of patterns R arguments can/should be used to specify names, e.g. list(name=pattern). This is a helper function which is useful for programmatically creating group names (see example for a typical use case).

Usage

```
group(name, ...)
```

Arguments

name	Column name in output.
...	Regex pattern(s).

Value

Named list.

Author(s)

Toby Dylan Hocking

Examples

```
## Three ways to create a group named data which matches zero or
## more non-newline characters.
str(list(data=".*"))
str(nc::group("data", ".*"))
g.name <- "data"
str(nc::group(g.name, ".*"))

## Data downloaded from
## https://en.wikipedia.org/wiki/Hindu%E2%80%93Arabic_numeral_system
numerals <- system.file(
  "extdata", "Hindu-Arabic-numerals.txt.gz", package="nc")

## Use engine="ICU" for unicode character classes
## http://userguide.icu-project.org/strings/regexp e.g. match any
```

```
## character with a numeric value of 2 (including japanese etc).
if(requireNamespace("stringi"))
  nc::capture_all_str(
  numerals,
  " ",
  two="[\\p{numeric_value=2}]",
  " ",
  engine="ICU")

## Create a table of numerals with script names.
digits.pattern <- list()
for(digit in 0:9){
  digits.pattern[[length(digits.pattern)+1]] <- list(
    "[|]",
    nc::group(paste(digit), "[^{|]+"),
    "[|]")
}
nc::capture_all_str(
  numerals,
  "\n",
  digits.pattern,
  "[|]",
  " *",
  "\\[\\[",
  name="[^\\]|]+")
```

`measure` *measure*

Description

Computes a value to be used as measure.vars argument to `melt.data.table`. NOTE: only works on newer versions of data.table that include the measure function.

Usage

```
measure(..., cols)
```

Arguments

`...`	Regular expression pattern list, passed to `capture_first_vec` with `cols` as subject.
`cols`	Character vector, column names to match with regex.

Details

`measure_multiple` is called if there is a capture `group` named "column" and `measure_single` is called otherwise.

Value

List or vector to use as measure.vars argument to melt.data.table.

Author(s)

Toby Dylan Hocking

Examples

```
if("measure" %in% ls(asNamespace("data.table"))){
  library(data.table)
  iris.dt <- data.table(datasets::iris[c(1,150),])
  melt(iris.dt, measure=nc::measure(part  =".*", "[.]", dim   =".*"))
  melt(iris.dt, measure=nc::measure(column=".*", "[.]", dim   =".*"))
  melt(iris.dt, measure=nc::measure(part  =".*", "[.]", column=".*"))
}
```

---

measure_multiple *measure multiple*

Description

Compute a measure.vars list (indicating multiple output columns) with variable_table attribute to pass to melt.data.table.

Usage

```
measure_multiple(subject.names,
    match.dt, no.match,
    fill = TRUE)
```

Arguments

subject.names

match.dt

no.match

fill

Author(s)

Toby Dylan Hocking

measure_single *measure single*

Description

Compute a measure.vars vector (indicating a single output column) with variable_table attribute to pass to melt.data.table.

Usage

```
measure_single(subject.names,
    match.dt, no.match,
    value.name = NULL)
```

Arguments

subject.names

match.dt

no.match

value.name

Author(s)

Toby Dylan Hocking

melt_list *melt list*

Description

Compute a list of arguments to pass to melt.data.table.

Usage

```
melt_list(measure.fun,
    dot.args, ...)
```

Arguments

measure.fun

dot.args

...

Author(s)

Toby Dylan Hocking

only_captures *only captures*

Description

Extract capture group columns from match.mat and assign optional groups to "".

Usage

```
only_captures(match.mat,
    stop.fun)
```

Arguments

match.mat

stop.fun

Author(s)

Toby Dylan Hocking

quantifier *quantifier*

Description

Create a group with a quantifier.

Usage

```
quantifier(...)
```

Arguments

...	Pattern(s) to be enclosed in a group, and a quantifier (last argument). A quantifier is character string: "?" for zero or one, "*?" for non-greedy zero or more, "+" for greedy one or more, etc.

Value

A pattern list.

Author(s)

Toby Dylan Hocking

Examples

```
## No need to use nc::quantifier when the pattern to be quantified
## is just a string literal.
digits <- "[0-9]+"

## nc::quantifier is useful when there is a sequence of patterns to
## be quantified, here an optional group with a dash (not captured)
## followed by some digits (captured in the chromEnd group).
str(optional.end <- nc::quantifier("-", chromEnd=digits, "?"))
str(optional.end <- list(list("-", chromEnd=digits), "?"))#same

## Use it as a sub-pattern for capturing genomic coordinates.
chr.pos.vec <- c(
  "chr10:213054000-213055000",
  "chrM:111000",
  "chr1:110-111 chr2:220-222") # two possible matches.
nc::capture_first_vec(
  chr.pos.vec,
  chrom="chr.*?",
  ":",
  chromStart=digits,
  optional.end)

## Another example which uses quantifier twice, for extracting code
## chunks from Rmd files.
vignette.Rmd <- system.file(
  "extdata", "vignette.Rmd", package="nc")
non.greedy.lines <- nc::quantifier(".*\n", "*?")
optional.name <- nc::quantifier(" ", name="[^,}]+", "?")
Rmd.dt <- nc::capture_all_str(
  vignette.Rmd,
  before=non.greedy.lines,
  "```\\{r",
  optional.name,
  parameters=".*",
  "\\}\n",
  code=non.greedy.lines,
  "```")
Rmd.dt[, chunk := 1:.N]
Rmd.dt[, .(chunk, name, parameters, some.code=substr(code, 1, 20))]
```

`stop_for_capture_same_as_id` *stop for capture same as id*

Description

Error if capture names same as `id.vars`.

Usage

```
stop_for_capture_same_as_id(capture.vars,
    id.vars)
```

Arguments

`capture.vars`

`id.vars`

Author(s)

Toby Dylan Hocking

---

stop_for_engine *stop for engine*

Description

Stop if specified `engine` is not available.

Usage

```
stop_for_engine(engine)
```

Arguments

`engine`	character string: PCRE, RE2, or ICU.

Value

character string.

Author(s)

Toby Dylan Hocking

stop_for_subject *stop for subject*

Description

Error if `subject` or `pattern` incorrect type.

Usage

```
stop_for_subject(subject,
    pattern)
```

Arguments

`subject`

`pattern`

Author(s)

Toby Dylan Hocking

subject_var_args *subject var args*

Description

Parse the complete argument list including subject.

Usage

```
subject_var_args(...)
```

Arguments

`...`	subject, regex/conversion.

Value

Result of `var_args_list` plus subject.

Author(s)

Toby Dylan Hocking

try_or_stop_print_pattern *try or stop print pattern*

Description

Try to run a capture function. If it fails we wrap the error message with a more informative message that also includes the generated pattern.

Usage

```
try_or_stop_print_pattern(expr,
    pat, engine)
```

Arguments

`expr`

`pat`

`engine`

Author(s)

Toby Dylan Hocking

var_args_list *var args list*

Description

Parse the variable-length argument list used in `capture_first_vec`, `capture_first_df`, and `capture_all_str`. This function is mostly intended for internal use, but is useful if you want to see the regex pattern generated by the variable argument syntax.

Usage

```
var_args_list(...)
```

Arguments

`...`	character vectors (for regex patterns) or functions (which specify how to convert extracted character vectors to other types). All patterns must be character vectors of length 1. If the pattern is a named argument in R, it becomes a capture `group` in the regex pattern. All patterns are pasted together to obtain the final pattern used for matching. Each named pattern may be followed by at most one function which is used to convert the previous named pattern. Patterns may also be lists, which are parsed recursively for convenience.

Value

a list with two named elements

`fun.list`	list of conversion functions with names corresponding to capture group(s)
`pattern`	regular expression string with capture group(s)

Author(s)

Toby Dylan Hocking

Examples

```
pos.pattern <- list("[0-9]+", as.integer)
nc::var_args_list(
  chrom="chr.*?",
  ":",
  chromStart=pos.pattern,
  list(
    "-",
    chromEnd=pos.pattern
  ), "?")
```

Index