

Package ‘gsubfn’

July 2, 2009

Version 0.5-0

Date 2009-07-01

Title Utilities for strings and function arguments.

Author G. Grothendieck

Maintainer G. Grothendieck <ggrothendieck@gmail.com>

Description gsubfn is like gsub but can take a replacement function or certain other objects instead of the replacement string. Matches and back references are input to the replacement function and replaced by the function output. gsubfn can be used to split strings based on content rather than delimiters and for quasi-perl-style string interpolation. The package also has facilities for translating formulas to functions and allowing such formulas in function calls instead of functions. This can be used with R functions such as apply, sapply, lapply, optim, integrate, xyplot, Filter and any other function that expects another function as an input argument or functions like cat or sql calls that may involve strings where substitution is desirable.

Depends R (>= 2.5.0), proto, tcltk

Suggests boot, chron, doBy, grid, lattice, quantreg, reshape, zoo

License GPL (>= 2)

URL <http://gsubfn.googlecode.com>

Repository CRAN

Date/Publication 2009-07-02 06:52:07

R topics documented:

gsubfn-package	2
as.function.formula	3
fn	4
gsubfn	6
match.funfn	9
strapply	10

Index	13
--------------	-----------

 gsubfn-package *gsubfn*

Description

Generalized "' gsub' " and associated functions.

Details

gsubfn is an R package used for string matching, substitution and parsing. A seemingly small generalization of gsub, namely allow the replacement string to be a replacement function, list, formula or proto object, can result in significantly increased power and applicability. The resulting function, gsubfn is the namesake of this package. In the case of a replacement formula the formula is interpreted as a function with the body of the function represented by the right hand side of the formula. In the case of a replacement proto object the object space is used to store persistent data to be communicated from one function invocation to the next as well as to store the replacement function/method itself.

Built on top of gsubfn is strapply which is similar to gsubfn except that it returns the output of the function rather than substituting it back into the source string.

The ability to have formula arguments that represent functions can be used not only in the functions of the gsubfn package but can also be used with any R function that itself passes functions without modifying its source. Such functions might include apply, lapply, mapply, sapply, tapply, by, integrate, optim, outer and other functions in the core of R and in add-on packages. Just preface any R function with `fn$` and subject to certain rules which are intended to distinguish which formulas are intended to be functions and which are not, the formula arguments will be translated to functions, e.g. `fn$integrate(~ x^2, 0, 1)` `fn$` also performs quasi-perl style string interpolation on any character arguments beginning with `\1`.

`match.funfn`, is provided to allow developers to readily build this functionality into their own functions so that even the `fn$` prefix need not be used.

The following are sources of information on "gsubfn":

News	<code>RShowDoc("NEWS", "gsubfn")</code>
Wish List	<code>RShowDoc("WISHLIST", package = "gsubfn")</code>
Thanks file	<code>RShowDoc("THANKS", package = "gsubfn")</code>
License	<code>RShowDoc("COPYING", package = "gsubfn")</code>
Citation	<code>citation(package = "gsubfn")</code>
Demo	<code>demo("gsubfn-chron")</code>
Demo	<code>demo("gsubfn-cut")</code>
Demo	<code>demo("gsubfn-gries")</code>
Demo	<code>demo("gsubfn-si")</code>
This File	<code>package?gsubfn</code>
Help files	<code>?gsubfn, ?strapply, ?cat0, ?paste0</code>
More Help files	<code>?as.function.formula, ?match.funfn, ?fn</code>
Home page	http://code.google.com/p/gsubfn/
Vignette	<code>vignette("gsubfn")</code>

Examples

```

# replace each number with that number plus 1
gsubfn("[[:digit:]]+", function(x) as.numeric(x)+1, "(10 20) (100 30)")

# same
gsubfn("[[:digit:]]+", ~ as.numeric(x)+1, "(10 20) (100 30)")

# place {1} after first word, {2} after second word
p <- proto(fun = function(this, x) paste0(x, "{", count, "}") )
gsubfn("\\w+", p, "hello world")

# replace each number with its cumulative sum
pcumsum <- proto(pre = function(this) this$sum <- 0,
  fun = function(this, x) { sum <- sum + as.numeric(x) }
)
gsubfn("[0-9]+", pcumsum, "10 abc 5 1")

# split out numbers
strapply("12abc34 55", "[0-9]+")

fn$optim(1, ~ x^2, method = "CG")

fn$integrate(~ sin(x) + cos(x), 0, pi/2)

fn$lapply(list(1:4, 1:5), ~ LETTERS[x]) # list(LETTERS[1:4], LETTERS[1:5])

fn$mapply(~ seq_len(x) + y * z, 1:3, 4:6, 2) # list(9, 11:12, 13:15)

# must specify x since . is a free variable
fn$by(CO2[4:5], CO2[1], x ~ coef(lm(uptake ~ ., x)), simplify = rbind)

# evaluate f at x^2 where f may be function or formula
square <- function(f, x, ...) { f <- match.fun(f); f(x^2, ...) }
square(~ exp(x)/x, pi)
square(function(x) exp(x)/x, pi) # same

```

```
as.function.formula
```

Make a one-line function from a formula.

Description

Create a function from a formula.

Usage

```
## S3 method for class 'formula':
as.function(x, ...)
```

Arguments

x	Formula with no left side.
...	Currently not used.

Value

A function is returned whose formal arguments are the variables in the left hand side, whose body is the expression on the right side of the formula and whose environment is the environment of the formula. If there is no left hand side the free variables on the right, in the order encountered are used as the arguments. `letters`, `LETTERS` and `pi` are ignored and not used as arguments. If the left hand side is 0 then the function is created as a zero argument function.

Note

`->`, `->>`, `=`, `<-`, `<<-` and `?` all have lower operator precedence than `~` so function bodies that contain them typically must be surrounded with `{ ... }`.

See Also

[Syntax](#).

Examples

```
as.function(~ as.numeric(x) + as.numeric(y))
as.function(x + y ~ as.numeric(x) + as.numeric(y)) # same
## Not run:
# example where function body must be surrounded with {...}
# due to use of <<-. See warning section above.
assign("mywarn", NULL, .GlobalEnv)
fn$tryCatch( warning("a warning"),
            warning = w ~ { mywarn <<- conditionMessage(w) })
print(mywarn)
## End(Not run)
```

 fn

Transform formula arguments to functions.

Description

When used in the form `fn$somefunction(...arguments...)` it converts formulas among the arguments of `somefunction` to functions using `as.function.formula`. It uses a heuristic to decide which formulas to convert. If any of the following are true then that argument is converted from a formula to a function: (1) there is only one formula among the arguments, (2) the name of the formula argument is `FUN` or (3) the formula argument is not the first argument in the argument list.

It also removes any `simplify` argument whose value is not logical and after processing it in the same way just discussed in order to interpret it as a function it passes the output of the command through `do.call(simplify, output)`.

It also performs quasi-perl style string interpolation on any character string arguments that begin with `\1` removing the `\1` character. A dollar sign followed by a variable name or R code within backticks are both evaluated.

Usage

```
## S3 method for class 'fn':
x$FUN
```

Arguments

x	fn.
FUN	Name of a function.

Value

Returns a function.

See Also

[as.function.formula](#)

Examples

```
# use of formula to specify a function.
# Note that LETTERS, letters and pi are automatically excluded from args
fn$lapply(list(1:4, 1:3), ~ LETTERS[x])
fn$sapply(1:3, ~ sin((n-1) * pi/180))

# use of simplify = rbind instead of do.call(rbind, by(...)).
# args to anonymous function are automatically determined.
fn$by(BOD, 1:nrow(BOD), ~ c(mn = min(x), mx = max(x)), simplify = rbind)

# calculate lm coefs of uptake vs conc for each Plant
fn$by(CO2, CO2$Plant, d ~ coef(lm(uptake ~ conc, d)), simplify = rbind)

# mid range of conc and uptake by Plant
fn$aggregate(CO2[,4:5], CO2[1], ~ mean(range(x)))

# string interpolation
j <- fn$cat("pi = $pi, exp = `exp(1)`\n")

## Not run:

# same but use cast/melt from reshape package
library(reshape)
fn$cast(Plant ~ variable, data = melt(CO2, id = 1:3), ~~ mean(range(x)))

# same
# uncomment when new version of doBy comes out (expected shortly)
# library(doBy)
```

```

# fn$summaryBy(.~Plant,CO2[-(2:3)],FUN= ~mean(range(x)), pref='midrange')
## End(Not run)

# generalized matrix product
# can replace sum(x*y) with any other inner product of interest
# this example just performs matrix multiplication of a times b
a <- matrix(4:1, 2)
b <- matrix(1:4, 2)
fn$apply(b, 2, x ~ fn$apply(a, 1, y ~ sum(x*y)))

# integration
fn$integrate(~1/((x+1)*sqrt(x)), lower = 0, upper = Inf)

# optimization
fn$optimize(~ x^2, c(-1,1))

# using fn with S4 definitions
setClass('ooc', representation(a = 'numeric'))
fn$setGeneric('incr', x + value ~ standardGeneric('incr'))
fn$setMethod('incr', 'ooc', x + value ~ {x@a <- x@a+value; x})
oo <- new('ooc',a=1)
oo <- incr(oo,1)
oo

## Not run:

# plot quantile regression fits for various values of tau
library(quantreg)
data(engel)
plot(engel$x, engel$y, xlab = 'income', ylab = 'food expenditure')
junk <- fn$lapply(1:9/10, tau ~ abline(coef(rq(y ~ x, tau, engel))))

# rolling mid-range
library(zoo)
fn$rollapply(LakeHuron, 12, ~ mean(range(x)))

library(lattice)
fn$xyplot(uptake ~ conc | Plant, CO2,
  panel = ... ~ { panel.xyplot(...); panel.text(200, 40, lab = 'X') })

library(boot)
set.seed(1)
fn$boot(rivers, ~ median(x, d), R = 2000)
## End(Not run)

x <- 0:50/50
matplot(x, fn$outer(x, 1:8, ~ sin(x * k*pi)), type = 'blobcsSh')

```

Description

Like `gsub` except instead of a replacement string one uses a function which accepts the matched text as input and emits replacement text for it.

Usage

```
gsubfn(pattern, replacement, x, backref, USE.NAMES = FALSE, env = parent.frame(), ...)
```

Arguments

<code>pattern</code>	Same as <code>pattern</code> in <code>gsub</code>
<code>replacement</code>	A character string, function, list, formula or proto object. See Details.
<code>x</code>	Same as <code>x</code> in <code>gsub</code>
<code>backref</code>	Number of backreferences to be passed to function. If zero or positive the match is passed as the first argument to the replacement function followed by the indicated number of backreferences as subsequent arguments. If negative then only the that number of backreferences are passed but the match itself is not. If omitted it will be determined automatically, i.e. it will be 0 if there are no backreferences and otherwise it will equal negative the number of back references. It determines this by counting the number of non-escaped left parentheses in the pattern.
<code>USE.NAMES</code>	See <code>USE.NAMES</code> in <code>sapply</code> .
<code>env</code>	Environment in which to evaluate the replacement function. Normally this is left at its default value.
<code>...</code>	Other <code>gsub</code> arguments.

Details

If `replacement` is a string then it acts like `gsub`.

If `replacement` is a function then each matched string is passed to the replacement function and the output of that function replaces the matched string in the result. The first argument to the replacement function is the matched string and subsequent arguments are the backreferences, if any.

If `replacement` is a list then the result of the regular expression match is, in turn, matched against the names of that list and the value corresponding to the first name in the list that is match is returned. If there are no names matching then the first unnamed component is returned and if there are no matches then the string to be matched is returned. If `backref` is not specified or is specified and is positive then the entire match is used to lookup the value in the list whereas if `backref` is negative then the identified backreference is used.

If `replacement` is a formula instead of a function then a one line function is created whose body is the right hand side of the formula and whose arguments are the left hand side separated by + signs (or any other valid operator). The environment of the function is the environment of the formula. If the arguments are omitted then the free variables found on the right hand side are used in the order encountered. 0 can be used to indicate no arguments. `letters`, `LETTERS` and `pi` are never automatically used as arguments.

If `replacement` is a proto object then it should have a `fun` method which is like the replacement function except its first argument is the object and the remaining arguments are as in the replacement

function and are affected by `backref` in the same way. `gsubfn` automatically inserts the named arguments in the call to `gsubfn` into the proto object and also maintains a `count` variable which counts matches within strings. The user may optionally specify `pre` and `post` methods in the proto object which are fired at the beginning and end of each string (not each match). They each take one argument, the object.

Two utility functions `cat0` and `paste0` are available. They are like `cat` and `paste` except that their default `sep` value is `" "`.

Value

As in `gsub`.

See Also

`gsub`, `strapply`, `cat`, `paste`

Examples

```
# adds 1 to each number in third arg
gsubfn("[[:digit:]]+", function(x) as.numeric(x)+1, "(10 20) (100 30)")

# same but using formula notation for function
gsubfn("[[:digit:]]+", ~ as.numeric(x)+1, "(10 20) (100 30)")

# replaces pairs m:n with their sum
s <- "abc 10:20 def 30:40 50"
gsubfn("[0-9]+:[0-9]+", ~ as.numeric(x) + as.numeric(y), s)

# default pattern for gsubfn does quasi-perl-style string interpolation
gsubfn( , , "pi = $pi, 2pi = `2*pi`")

# Extracts numbers from string and places them into numeric vector v.
# Normally this would be done in strapply instead.
v <- c(); f <- function(x) v <- append(v, as.numeric(x))
junk <- gsubfn("[0-9]+", f, "12;34:56,89,,12")
v

# same
strapply("12;34:56,89,,12", "[0-9]+", simplify = c)

# makes all letters except first in word lower case
gsubfn("\\B.", tolower, "I LIKE A BANANA SPLIT", perl = TRUE)

# replaces numbers with that many Xs
gsubfn("[[:digit:]]+", ~ paste(rep("X", n), collapse = ""), "5.2")

# replaces units with scale factor
gsubfn(".m", list(cm = "e1", km = "e6"), "33cm 45km")

# place <...> around first two occurrences
p <- proto(fun = function(this, x) if (count <= 2) paste0("<", x, ">") else x)
```

```
gsubfn("\\w+", p, "the cat in the hat is back")

# replace each number by cumulative sum to that point
p2 <- proto(pre = function(this) this$value <- 0,
            fun = function(this, x) this$value <- value + as.numeric(x))
gsubfn("[0-9]+", p2, "12 3 11, 25 9")
```

match.funfn	<i>Generic extended version of R match.fun</i>
-------------	--

Description

A generic `match.fun`.

Usage

```
match.funfn(FUN, descend = TRUE)
```

Arguments

<code>FUN</code>	Function, character name of function or formula describing function.
<code>descend</code>	logical; control whether to search past non-function objects.

Details

The default method is the same as `match.fun` and the `formula` method is the same as `as.function.formula`. This function can be used within the body of a function to convert a function specification whether its a function, character string or formula into an actual function.

Value

Returns a function.

See Also

See Also [match.fun](#), [as.function.formula](#).

Examples

```
# return first argument evaluated at second argument squared.
sq <- function(f, x) {
  f <- match.funfn(f)
  f(x^2)
}

# call sq using different forms for function
sq(function(x) exp(x)/x, pi)
f <- function(x) exp(x)/x
```

```
sq("f", pi)
sq(~ exp(x)/x, pi)
sq(x ~ exp(x)/x, pi)
```

strapply

Apply a function over a string or strings.

Description

Similar to `gsubfn` except instead of performing substitutions it returns the output of `FUN`.

Usage

```
strapply(X, pattern, FUN = function(x, ...) x, backref = NULL, ...,
         ignore.case = FALSE, perl = FALSE, engine = c("tcl", "R"),
         simplify = FALSE, USE.NAMES = FALSE, combine = c)
```

Arguments

<code>X</code>	list or (atomic) vector of character strings to be used.
<code>pattern</code>	character string containing a regular expression (or character string for <code>fixed = TRUE</code>) to be matched in the given character vector.
<code>FUN</code>	a function, formula, character string, list or proto object to be applied to each element of <code>X</code> . See discussion in gsubfn .
<code>backref</code>	See gsubfn .
<code>ignore.case</code>	If <code>TRUE</code> then case is ignored in the <code>pattern</code> argument.
<code>perl</code>	If <code>TRUE</code> then <code>engine="R"</code> is used with perl regular expressions.
<code>engine</code>	Specifies which engine to use. <code>engine="tcl"</code> , the default unless <code>FUN</code> is a proto object in which case the <code>"R"</code> engine is used (regardless of the setting of this argument).
<code>...</code>	optional arguments to <code>gsubfn</code> .
<code>simplify</code>	logical or function. If logical, should the result be simplified to a vector or matrix, as in <code>sapply</code> if possible? If function, that function is applied to the result with each component of the result passed as a separate argument. Typically if the form is used it will typically be specified as <code>rbind</code> .
<code>USE.NAMES</code>	logical; if <code>TRUE</code> and if <code>X</code> is character, use <code>X</code> as 'names' for the result unless it had names already.
<code>combine</code>	multiple results from same string are combined using this function. The default is <code>"c"</code> . <code>"list"</code> is another common choice. The default may change to be <code>"list"</code> in the future.

Details

If FUN is a function then for each character string in "X" the pattern is repeatedly matched, each such match along with back references, if any, are passed to the function "FUN" and the output of FUN is returned as a list. If FUN is a formula or proto object then it is interpreted to the way discussed in [gsubfn](#).

If FUN is a proto object or if `perl=TRUE` is specified then `engine="R"` is used and the `engine` argument is ignored.

If `backref` is not specified and `engine="R"` is specified or implied then a heuristic is used to calculate the number of backreferences. The primary situation that can fool it is if there are parentheses in the string that are not back references. In those cases the user will have to specify `backref`. If `engine="tcl"` then an exact algorithm is used and the problem sentence never occurs.

Value

A list of character strings.

See Also

See [gsubfn](#) and [sapply](#). For regular expression syntax used in tcl see http://www.tcl.tk/man/tcl8.6/TclCmd/re_syntax.htm and for regular expression syntax used in R see [regex](#).

Examples

```
strapply("12;34:56,89,,12", "[0-9]+")

# separate leading digits from rest of string
# creating a 2 column matrix: digits, rest
s <- c("123abc", "12cd34", "1e23")
t(strapply(s, "^([[:digit:]]+)(.*)", c, simplify = TRUE))

# same but create matrix
strapply(s, "^([[:digit:]]+)(.*)", c, simplify = rbind)

# running window of 5 characters using 0-lookahead perl regexp
# Note that the three ( in the regexp will fool it into thinking there
# are three backreferences so specify backref explicitly.
x <- "abcdefghijkl"
strapply(x, "(.)(?=(....))", paste0, backref = -2, perl = TRUE)[[1]]

# Note difference. First gives character vector. Second is the same.
# Third has same elements but is a list.
# Fourth gives list of two character vectors. Fifth is the same.
strapply("a:b c:d", "(.):(.)", c)[[1]]
strapply("a:b c:d", "(.):(.)", list, simplify = unlist) # same

strapply("a:b c:d", "(.):(.)", list)[[1]]

strapply("a:b c:d", "(.):(.)", c, combine = list)[[1]]
```

```
strapply("a:b c:d", "(.):(.)", c, combine = list, simplify = c) # same

# find second CPU_SPEED value given lines of config file
Lines <- c("DEVICE = 'PC'", "CPU_SPEED = '1999', '233'")
parms <- strapply(Lines, "[^ ',=]+", c, USE.NAMES = TRUE,
  simplify = ~ lapply(list(...), "[", -1))
parms$CPU_SPEED[2]

# return first two words in each string
p <- proto(fun = function(this, x) if (count <=2) x)
strapply(c("the brown fox", "the eager beaver"), "\\w+", p)

## Not run:
# convert to chron
library(chron)
x <- c("01/15/2005 23:32:45", "02/27/2005 01:22:30")
x.chron <- strapply(x, "(../../....) (...:..:..)", chron, simplify = c)
## End(Not run)
```

Index

- *Topic **character**
 - as.function.formula, 3
 - gsubfn, 6
 - strapply, 10
- *Topic **package**
 - gsubfn-package, 2
- *Topic **programming**
 - fn, 4
 - match.funfn, 9
 - \$.fn(fn), 4
- as.function.formula, 2, 3, 4, 5, 9
- cat, 8
- cat0, 2
- cat0(gsubfn), 6
- eval.with.vis(fn), 4
- fn, 2, 4
- gsub, 7, 8
- gsubfn, 2, 6, 10, 11
- gsubfn-package, 2
- match.fun, 9
- match.funfn, 2, 9
- matrixfn(fn), 4
- ostrapply(strapply), 10
- paste, 8
- paste0, 2
- paste0(gsubfn), 6
- regex, 11
- sapply, 7, 11
- strapply, 2, 8, 10
- strapply1(strapply), 10
- Syntax, 4