gsubfn: Utilities for Strings and for Function Arguments.

Gabor Grothendieck
GKX Associates Inc.

Abstract

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, formula or **proto** object, can result in significantly increased power and applicability. The resulting function, `gsubfn` is the namesake of this package. 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. In the case of a replacement formula the formula is interpreted as a function as explained in the text. In the case of a replacement **proto** object the object space is used to store persistant data to be communicated from one function invocation to the next as well as to store the replacement function/method itself.

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 without modifying its source. 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)`. This facility has widespread applicability right across R and its packages. `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.

Keywords: gsub, strings, R.

1. Introduction

The R system for statistical computing contains a powerful function for string substitution called `gsub` which takes a regular expression, replacement string and source string and replaces all matches of the regular expression in the source string with the replacement string. Parenthesized items in the regular expression, called back references, can be referred to in the replacement string further increasing the range of applications that `gsub` can address.

The key function and namesake of the **gsubfn** package is a function which is similar to `gsub` but the replacement string can optionally be a replacement function, formula (representing a function) or replacement `proto` object.

Associated functions built on top of `gsubfn` are `strapply` which is an `apply` style function that is like `gsubfn` except that it returns the output of the replacement function rather than substituting it back into the string.

In the case that a function is passed to `gsubfn`, for each match of the regular expression in the source string, the replacement function is called with one argument per backreference or if no backreferences with the match (unless instructed otherwise by the `backref` argument). Note that it determines the number of backreferences by counting the number of occurrences of `(` in the regular expression so if there are parentheses that are not back references it may be important to specify their number explicitly with `backref`. The output of the replacement function is substituted back into the string replacing the match. In those cases where persistance is needed between invocations of the function a `proto` object containing a replacement method (a method is another name for

function in this context) can be used and the object itself can be used by the replacement method as a repository for data that is to persist between calls to the replacement method. Such persistant data might be counts, prior matches and so on. Also `gsubfn` automatically places the argument values that `gsubfn` was called with as well as a `count` representing the number of matches so far into the object for use by the function. `pre` and `post` functions can also be entered into the object and are triggerred at the beginning and end, respectively, of each string.

The idea of using a replacement function is also found in the Lua language http://www.lua.org/manual/5.1/manual.html#pdf-string.gsub. . **gsubfn** follows that idea and builds on it with **proto** objects, formulas and associated function `strapply`.

The remainder of this article is organized as follows: Section 2 explains the use `gsubfn` with replacement functions. Section 4 explains the use `gsubfn` with **proto** objects for applications requiring persistance between calls. Section 5 explains the use `strapply` and Section 6 explains the use of `cat0` and `paste0`.

The functions specified in `gsubfn` can be specified as functions or using a formula notation. Facilities are included for using that notation with any R function, not just the ones in the **gsubfn** package. Section 7 explains this facility even if the function in question, e.g. `apply`, `integrate` was not so written and Section 8 explains how developers can embed this into their own functions.

Prerequisites. The reader should be familiar with R and, in particular the R `gsub` function. Within R, help on `gsub` is found via the `?gsub` command and on the net it can be found at

- http://stat.ethz.ch/R-manual/R-patched/library/base/html/grep.html

The reader should also be familiar with regular expressions. Within R, help on regular expressions is found via the command `?regex` and on the net it can be found at

- http://stat.ethz.ch/R-manual/R-patched/library/base/html/regex.html

Other Internet sources of information on regular expressions not specifically concerned with R are

- Perl compatible regular expressions. http://www.pcre.org/
- Regular expressions. http://www.regular-expressions.info/
- Wikipedia. http://en.wikipedia.org/wiki/Regular_expression

The discussions of passing **proto** objects to `gsubfn` and `strapply` require a minimal understanding of R environments using the R help command `?environment` and the R Language Manual found online at

- http://stat.ethz.ch/R-manual/R-patched/library/base/html/environment.html
- http://finzi.psych.upenn.edu/R/doc/manual/R-lang.html#Environment-objects

Since the use of the **proto** package itself is relatively restricted we will include sufficient information so that outside reference to the **proto** package will be unnecessary for the restricted purpose of using it here.[1]

2. The `gsubfn` Function

Introduction. The `gsubfn` function has a similar calling sequence to the R `gsub` function. The first argument is a regular expression, the second argument is a replacement string, replacement function, replacement formula representing a function or a replacement **proto** object. The third

[1] More about **proto** is available in the four documents listed under Documentation on the **proto** home page: http://hhbio.wasser.tu-dresden.de/projects/proto/ .

argument is the source string or a vector of such strings. In this section we are mainly concerned with replacement functions and replacement formulas representing replacement functions. In this case the replacement function is called for each match. The match and back references are passed as arguments. The input string is then copied to the output with the match being replaced with the output of the replacement function.

Replacement function. The replacement function can be specified by a formula in which the left hand side of the formula are the arguments separated by `"+"` (or any other valid formula symbol) while the right hand side represents the body. The environment of the formula will be used as the environment of the generated funciton. If the arguments are omitted then the free variables on the right hand side are used in the order encountered.

Back References. If the `backref` argument is not specified then the match followed all backreferences are passed to the function as separate arguments. If `backref` is `0` then no back references are passed. If `backref` is a postive integer, n, then the match and the first n back references are passed. If `backref` is a negative integer then the match is not passed and the absolute value of `backref` is used as the number of back references to pass. Since `gsubfn` uses a potentially time consuming trial and error algorithm to automatically determine the number of back references the performance can be sped up somewhat by specifying `backref` even if all back references are to be passed.

Example. This example below replaces `x:y` pairs in `s` with their sum. The formula in this example is equivalent to specifying the function `function(x, y) as.numeric(x) + as.numeric(y)` :

This example replaces the scale letter with an appropriate scale factor. For example, `"G"` means giga and corresponds to `"e9"`.

```
> s <- "abc 10:20 def 30:40 50"
> gsubfn("([0-9]+):([0-9]+)", ~as.numeric(x) + as.numeric(y), s)

[1] "abc 30 def 70 50"

> dat <- c("3.5G", "88P", "19")
> gsubfn("[MGP]$", ~c(M = "e6", G = "e9", P = "e12")[[x]], dat)

[1] "3.5e9" "88e12" "19"
```

A variation of the last example is available via the command `demo("gsubfn-si")`.

3. gsubfn with list objects

Example. If the replacement object is a list then the match is matched against the names of the list and the corresponding value is returned. If no name matches then the first unnamed list component is returned. If there is still no match then the string to be matched is returned so that effectively the lookup is ignored.

Here is the last example redone with a list:

```
> dat <- c("3.5G", "88P", "19")
> gsubfn("[MGP]$", list(M = "e6", G = "e9", P = "e12"), dat)

[1] "3.5e9" "88e12" "19"
```

4. gsubfn with proto objects

Introduction. In some applications one may need information from prior matches on current matches. This may be as simple as a count or as comprehensive as all prior matches. This is accomplished by passing a **proto** object whose object space can contain variables to be shared among the invocations of the matching function. The matching function itself is also be stored in the object as are the arguments to `gsubfn` and a special variable `count` which is automatically set to the match number.

Proto. A **proto** object is an R environment with an S3 class of `c("proto", "environment")`. A **proto** object is created by calling the `"proto"` function with the components to be inserted given as arguments. This is very similar to the way lists are constructed in R except that unlike a list a **proto** object represents an R environment.

Example. The use of **proto** objects is best introduced via example. In the following example `p` is a **proto** object which contains one function `fun`. A function component of a **proto** object is called a method and we will use this terminology henceforth. In this example after the `proto` command to create `p` we examine the class of `p` and check the components of `p` using `ls`. Also we display the `fun` component itself. These are some of the basic operations on **proto** objects. Finally we run `gsubfn` using the regular expression `\\w+` and the **proto** object `p`. `gsubfn` looks for a component called `fun` in `p` and uses that as the replacement method/function. The arguments to `fun` are always the object itself, often represented by the formal argument `this`, `self` or just `.`, followed by the match and back references. In this example there are no back references. Here `fun` simply returns the match suffixed by the count of the match. The `count` variable is automatically placed into `p` by `gsubfn`. This has the effect of suffixing the first word with with `1`, the second with `2` and so on. After running `gsubfn` we examine `p` again noticing all the components that were added by `gsubfn` and we also examine the `count` component which shows how many matches were found. Note that use of `paste0` which is like `paste` but has a default `sep` of `""`.

```
> p <- proto(fun = function(this, x) paste0(x, "{", count, "}"))
> class(p)

[1] "proto"       "environment"

> ls(p)

[1] "fun"

> with(p, fun)

function (this, x)
paste0(x, "{", count, "}")
<environment: 0x03000764>

> s <- c("the dog and the cat are in the house", "x y x")
> gsubfn("\\w+", p, s)

[1] "the{1} dog{2} and{3} the{4} cat{5} are{6} in{7} the{8} house{9}"
[2] "x{1} y{2} x{3}"

> ls(p)

[1] "backref"     "count"       "env"         "fun"         "match"
[6] "pattern"     "replacement" "USE.NAMES"   "x"

> p$count
```

```
[1] 3
```

pre and post. gsubfn knows about three methods: fun which we have already seen as well as pre and post. The latter two are optional and are run before each string and after each string respectively. Suppose we wish to suffix each word not by the count of all words but just by the count of that word. Thus the third occurrence of "the" will be suffixed with 3 rather than 8. In that case we will set up a words list in the pre method. This method will be invoked at the start of each of the two strings in s. The words list itself is stored in the pwords **proto** object. Since all the methods of a proto object can share its contents fun can also make use of it. In the example below, each time we match a word, pwords$fun adds it to the list words, if not already there, and increments it so that words["the"] will be 1 after "the" is encountered for the first time, 2 after the second time and so on. At the end of the example we look at what variables are in pwords and also check the contents of the words list.

```
> pwords <- proto(pre = function(this) {
+     this$words <- list()
+ }, fun = function(this, x) {
+     if (is.null(words[[x]]))
+         this$words[[x]] <- 0
+     this$words[[x]] <- words[[x]] + 1
+     paste0(x, "{", words[[x]], "}")
+ })
> gsubfn("\\w+", pwords, "the dog and the cat are in the house")

[1] "the{1} dog{1} and{1} the{2} cat{1} are{1} in{1} the{3} house{1}"

> ls(pwords)

 [1] "backref"     "count"       "env"         "fun"         "match"
 [6] "pattern"     "pre"         "replacement" "USE.NAMES"   "words"
[11] "x"

> dput(pwords$words)

structure(list(the = 3, dog = 1, and = 1, cat = 1, are = 1, "in" = 1,
    house = 1), .Names = c("the", "dog", "and", "cat", "are",
"in", "house"))
```

Additional examples of the use of **proto** objects with gsubfn are available via the command demo("gsubfn-proto").

5. strapply

Introduction. The strapply function is similar to the gsubfn function but instead of replacing the matched strings it returns the output of the function in a list or simplified structure. A typical use would be to split a string based on content rather than on delimiters. The arguments are analogous to the arguments in apply. In both the object to be applied over is the first argument. A modifier, which is an index for apply and a regular expression for strapply is the second argument. The third argument is a function in both cases although in strapply, in analogy to gsubfn it can also be a **proto** object. The simplify argument is similar to the simplify argument in sapply and, in fact, is passed to sapply if it is logical. If simplify is a function or a formula representing a function then the output of strapply is passed as output to it via do.call(simplify, output).

Example. To separate out the initial digits from the rest returning the the initial digits and the rest as two separate fields we can write this:

```
> s <- c("123abc", "12cd34", "1e23")
> strapply(s, "^([[:digit:]]+)(.*)", c, simplify = rbind)

     [,1]  [,2]
[1,] "123" "abc"
[2,] "12"  "cd34"
[3,] "1"   "e23"
```

In this example we calculate the midpoint of each interval. The `()` will fool it into thinking there is a back reference so specify `backref` explicitly telling it to pass the match.

```
> rn <- c("[-11.9,-10.6]", "(NA,9.3]", "(9.3,8e01]", "(8.01,Inf]")
> colMeans(strapply(rn, "[^][(),]+", as.numeric, backref = 0, simplify = TRUE))

[1] -11.25     NA  44.65    Inf
```

`combine`. The `combine` argument can be specified as a function which is to be applied to the output of the replacement function after each call. It defaults to `c`. Another popular choice is `list`. The following example illustrates the difference:

```
> s <- c("a:b c:d", "e:f")
> dput(strapply(s, "(.):(.)", c))

list(c("a", "b", "c", "d"), c("e", "f"))

> dput(strapply(s, "(.):(.)", c, combine = list))

list(list(c("a", "b"), c("c", "d")), list(c("e", "f")))
```

`strapply` and `proto`. `strapply` can be used with **proto** in the same way as as `gsubfn`. For example, suppose we wish to extract the words from a string together with their ordinal occurrence number. Previously we did this with `gsubfn` and inserted the number back into the string. This time we want to extract it.

```
> pwords2 <- proto(pre = function(this) {
+     this$words <- list()
+ }, fun = function(this, x) {
+     if (is.null(words[[x]]))
+         this$words[[x]] <- 0
+     this$words[[x]] <- words[[x]] + 1
+     list(x, words[[x]])
+ })
> strapply("the dog and the cat are in the house", "\\w+", pwords2,
+     combine = list, simplify = x ~ do.call(rbind, x))

      [,1]    [,2]
 [1,] "the"   1
 [2,] "dog"   1
 [3,] "and"   1
 [4,] "the"   2
 [5,] "cat"   1
 [6,] "are"   1
 [7,] "in"    1
 [8,] "the"   3
 [9,] "house" 1
```

```
> ls(pwords2)

[1] "combine"   "count"     "fun"       "pattern"   "pre"       "simplify"
[7] "USE.NAMES" "words"     "X"

> dput(pwords2$words)

structure(list(the = 3, dog = 1, and = 1, cat = 1, are = 1, "in" = 1,
    house = 1), .Names = c("the", "dog", "and", "cat", "are",
"in", "house"))
```

6. Miscellaneous

The `cat0` and `paste0` function are like `cat` and `paste` they have a default `sep` of `""`.

Here is an example of using paste0. This example retrieves overlapping segments consisting of a space, letter, space, letter and space. Only the final space, letter, space is returned. Because we did not specify `backref` it will think there are two back references (since it will interpret the lookahead expression as an extra back reference); however, the second is empty so it does no harm in passing it to `paste0`. It uses the zero-lookahead perl style pattern matching expression.

```
> strapply(" a b c d e f ", " [a-z](?=( [a-z] ))", paste0, perl = TRUE)[[1]]

[1] " b " " c " " d " " e " " f "
```

7. fn

Wherever a function can be specified in `gsubfn` and `strapply` one can specify a formula instead as discussed previously. This facility has been extended to work with any R function. Just preface the function with `fn$` and

1. formula arguments will be intercepted and translated to functions allowing a compact representation of the call. Which formulas are actually translated to functions is dependent on rules to be discussed. The right hand side of the formula represents the body of the function. The left hand side of the formula represents the arguments and defaults to the free variables in the order encountered. The environment of the function is set to the environment of the formula. `letters`, `LETTERS` and `pi` are not considered free variables and will not appear in arguments.
2. character arguments will be intercepted and quasi-perl style string interpolation will be performed. Which character strings to operate on are dependent on rules to be discussed.
3. the `simplify` argument if its value is a function is intercepted. In that case if `result` is the result of running the function without the `simplify` argument then it returns `do.call(simplify, result)`.

The rules for determining which formulas to translate and which character strings to apply quasi-perl style string interpolation are as follows:

1. any formula argument that has been specified with a double `~`, i.e. `~~`, is converted to a function after removing the double `~` and replacing it with a single `~`.
2. any character string argument that has been specified with a first character of `\1` has string interpolation applied to it after the `\1` is removed.

3. if there are no formulas with double ~ and no character strings beginning with \1 then all formulas are converted to functions and if there are no formulas then all character strings have string interpolation done.

The last possibility is the actually the most commonly used and almost all our examples will illustrate that case. For example,

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

2 with absolute error < 2.2e-14

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

[[1]]
[1] "A" "B" "C" "D"

[[2]]
[1] "A" "B" "C" "D" "E"

> fn$mapply(~seq_len(x) + y * z, 1:3, 4:6, 2)

[[1]]
[1] 9

[[2]]
[1] 11 12

[[3]]
[1] 13 14 15

> fn$by(CO2[4:5], CO2[2], x ~ coef(lm(uptake ~ ., x)), simplify = rbind)

            (Intercept)       conc
Quebec         23.50304 0.02308005
Mississippi    15.49754 0.01238113
```

Here is an example where we have two formulas, one of which should be translated and another should not. In this case we place a double ~ in the second formula to signify that one it represents a function. The first formula is then correctly left untranslated. This example places a panel number in the body of each panel.

```
> library(lattice)
> library(grid)
> print(fn$xyplot(uptake ~ conc | Plant, CO2, panel = ~~{
+     panel.xyplot(...)
+     grid.text(panel.number(), 0.1, 0.85)
+ }))
```

As mentioned briefly above, the `fn$` prefix will also intercept any `simplify` argument if that argument is a function (but will not intercept it if it is `TRUE` or `FALSE`). In the case of inteception it runs the command then applies `do.call(simplify, result)` to the result of the command. A typical use would be with `by` as in the following example to calculate the regression coefficients of `uptake` on `conc` for each `Treatment`. This replaces the sligtly uglier `do.call` construct which would otherwise have been required.

```
> fn$by(CO2, CO2$Treatment, d ~ coef(lm(uptake ~ conc, d)), simplify = rbind)

           (Intercept)       conc
nonchilled    22.01916 0.01982458
chilled       16.98142 0.01563659
```

Here are some additional examples to illustrate the wide range of application. The first replaces codes with upper case letters. Note that `LETTERS` is never interpreted as a free variable so the default argument is `x` here:

```
> fn$lapply(list(1:4, 1:3), ~LETTERS[x])

[[1]]
[1] "A" "B" "C" "D"

[[2]]
[1] "A" "B" "C"
```

The next example uses `aggregate` to calculate the midrange of each of `conc` and `uptake` for each `Type`. The calculation is repeated using `cast` from the **reshape** package and again using `summaryBy` from the **doBy** package. Note that in two of the examples there are two formulas each and one is to be regarded as a function and the other not so we must use double ~ to distinguish the cases.

```
> fn$aggregate(CO2[4:5], CO2[3], ~mean(range(x)))

   Treatment  conc uptake
1 nonchilled 547.5  28.05
2    chilled 547.5  25.05

> library(reshape)
> fn$cast(Treatment ~ variable, data = melt(CO2, id = 1:3), ~~mean(range(x)))

   Treatment  conc uptake
1 nonchilled 547.5  28.05
2    chilled 547.5  25.05

> library(doBy)
> fn$summaryBy(. ~ Treatment, CO2[3:5], FUN = ~~c(midrange = mean(range(x))))

   Treatment conc.midrange uptake.midrange
1 nonchilled         547.5           28.05
2    chilled         547.5           25.05
```

Here is another common use of `aggregate` or `by`. This calculates a weighted mean of the first column using weights in the second column all grouped by columns `A` and `B`. The `aggregate` example aggregates over indexes to circumvent the restriction of a single input to the aggregation function. Since both `i` and `X` are free variables but we only want `i` to be an argument so we must specify it explicitly. In the `by` example there is only one free variable `x` so we can rely on the default for argument processing. Also note the use of `simplify=rbind` in the `by` case:

```
> set.seed(1)
> X <- data.frame(X = rnorm(24), W = runif(24), A = gl(2, 1, 24),
+     B = gl(2, 2, 24))
> fn$aggregate(1:nrow(X), X[3:4], i ~ weighted.mean(X[i, 1], X[i,
+     2]))
```

```
  A B           x
1 1 1 -0.20178587
2 2 1  0.01591515
3 1 2  0.63162232
4 2 2  0.11378828
```

```
> fn$by(X, X[3:4], ~data.frame(wmean = weighted.mean(x[1], x[2]),
+     x[1, 3:4]), simplify = rbind)
```

```
        wmean A B
1 -0.20178587 1 1
2  0.01591515 2 1
3  0.63162232 1 2
4  0.11378828 2 2
```

A number of mathematical functions take functions as arguments. Here we show the use of `fn$` with `integrate` and `optimize`.

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

```
3.141593 with absolute error < 2.7e-05
```

```
> fn$optimize(~x^2, c(-1, 1))
```

```
$minimum
[1] 2.710505e-20

$objective
[1] 7.34684e-40
```

S4 `setGeneric` and `setMethod` calls have function arguments that `fn$` can be used with. In the following example we create an S4 class `ooc` whose representation contains a single variable `a`. We then define a generic function `incr`. In this case the function arguments cannot be deduced from the body so we specify them explicitly. Then we define an `incr` method for class `ooc`. Since `a` is a free variable again we must define the arguments explicitly to ensure that it is not automatically included. Finally we illustrate the use of the `incr` method we just defined.

```
> setClass("ooc", representation(a = "numeric"))
```

```
[1] "ooc"
```

```
> fn$setGeneric("incr", x + value ~ standardGeneric("incr"))
```

```
[1] "incr"
```

```
> fn$setMethod("incr", "ooc", x + value ~ {
+     x@a <- x@a + value
+     x
+ })
```

```
[1] "incr"
```

```
> oo <- new("ooc", a = 1)
> oo <- incr(oo, 1)
> oo
```

```
An object of class "ooc"
Slot "a":
[1] 2
```

One commonly used calculation in quantile regression is the creation of a regression plot for each of a variety of values of `tau`. Here we plot `x` vs. `y` and then superimpose quantile regression lines for various `tau` values using `lapply` to avoid a loop. The `lapply` function of `tau` is specified using a formula.

```
> plot(engel$income, engel$foodexp, xlab = "income", ylab = "food expenditure")
> junk <- fn$lapply(1:9/10, tau ~ abline(coef(rq(foodexp ~ income,
+     tau, engel))))
```

In time series we may wish to calculate a rolling summary of the data. In this case we calculate a rolling midrange of the data using the **zoo** function `rollapply`:

```
> library(zoo)
> fn$rollapply(LakeHuron, 12, ~mean(range(x)))

Time Series:
Start = 1880
End = 1966
Frequency = 1
 [1] 580.825 580.825 580.735 580.735 580.735 580.410 580.410 580.410 580.410
[10] 580.060 579.960 579.960 579.705 579.385 579.125 579.075 578.955 578.955
[19] 579.020 579.035 579.035 579.065 579.415 579.415 579.350 579.100 579.100
[28] 579.100 579.100 579.050 579.050 579.110 579.115 579.115 579.115 579.115
[37] 579.115 579.095 578.965 578.445 578.445 578.445 578.445 578.665 578.665
[46] 578.665 578.665 578.665 578.410 578.410 578.410 578.410 578.410 578.410
[55] 578.410 577.860 577.330 577.925 577.925 577.925 578.225 578.230 578.255
[64] 578.420 578.420 578.490 579.040 579.400 579.400 579.400 579.400 579.400
[73] 579.030 578.990 578.990 578.990 578.990 578.870 578.185 577.960 577.785
[82] 577.530 577.530 577.850 577.850 577.925 577.960
```

A common statistical technique for assessing statistics is the bootstrap technique provided in package **boot**. Here we compactly the bias and standard error of the median statistic using the `rivers` data set and 2000 samples.

```
> library(boot)
> set.seed(1)
> fn$boot(rivers, ~median(x[d]), R = 2000)

ORDINARY NONPARAMETRIC BOOTSTRAP


Call:
boot(data = c(735, 320, 325, 392, 524, 450, 1459, 135, 465, 600,
330, 336, 280, 315, 870, 906, 202, 329, 290, 1000, 600, 505,
1450, 840, 1243, 890, 350, 407, 286, 280, 525, 720, 390, 250,
327, 230, 265, 850, 210, 630, 260, 230, 360, 730, 600, 306, 390,
420, 291, 710, 340, 217, 281, 352, 259, 250, 470, 680, 570, 350,
300, 560, 900, 625, 332, 2348, 1171, 3710, 2315, 2533, 780, 280,
410, 460, 260, 255, 431, 350, 760, 618, 338, 981, 1306, 500,
```

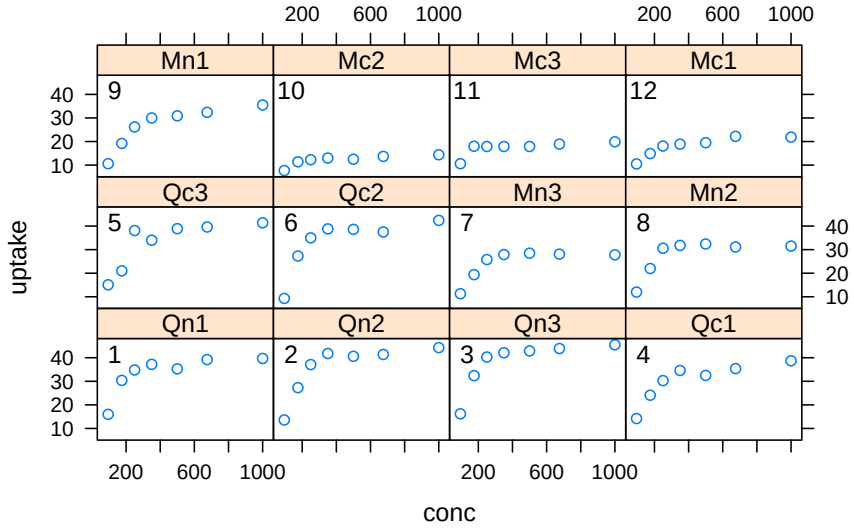

Figure 1: `fn$xyplot`

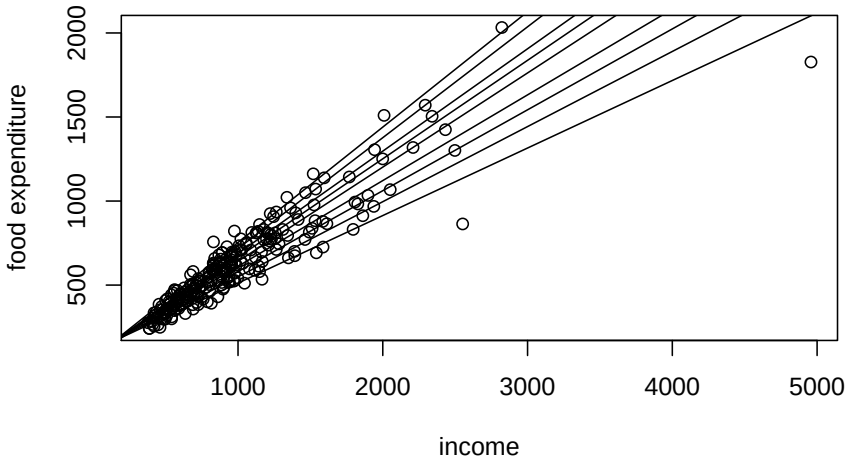

Figure 2: `Plot engel data with quantile lines`

```
696, 605, 250, 411, 1054, 735, 233, 435, 490, 310, 460, 383,
375, 1270, 545, 445, 1885, 380, 300, 380, 377, 425, 276, 210,
800, 420, 350, 360, 538, 1100, 1205, 314, 237, 610, 360, 540,
1038, 424, 310, 300, 444, 301, 268, 620, 215, 652, 900, 525,
246, 360, 529, 500, 720, 270, 430, 671, 1770), statistic = function (x,
    d)
median(x[d]), R = 2000)


Bootstrap Statistics :
    original  bias    std. error
t1*      425   2.615    26.19020
```

Here is a plotting application that illustrates that `pi` is automatically excluded from default arguments.

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

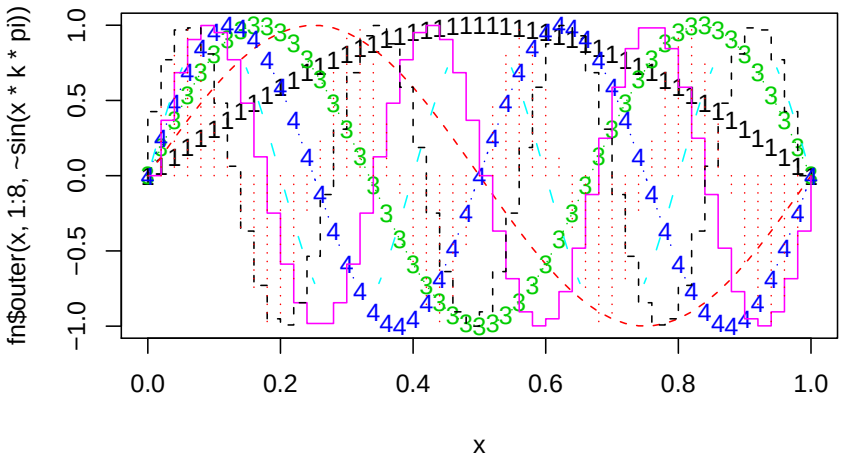

Figure 3: `matplot(x, fn$outer(x, 1:8, ~ sin(x * k*pi)), type = 'blobcsSh')`

Here we define matrix multiplication in terms of two calls to `apply` and the inner product definition. The advantage of this is that it can easily be modified to use different inner products. This illustrates a nested use of `fn$`:

```
> a <- matrix(4:1, 2)
> b <- matrix(1:4, 2)
> fn$apply(b, 2, x ~ fn$apply(a, 1, y ~ sum(x * y)))

     [,1] [,2]
[1,]    8   20
[2,]    5   13
```

```
> a %*% b

     [,1] [,2]
[1,]    8   20
[2,]    5   13
```

Another example of nesting is the following which generates all subsequences of `1:4`.

```
> L <- fn$apply(fn$sapply(1:4, ~rbind(i, i:4), simplify = cbind),
+     2, ~x[1]:x[2])
> dput(L)

list(1L, 1:2, 1:3, 1:4, 2L, 2:3, 2:4, 3L, 3:4, 4L)
```

In the Python language there exists a convenient notation for expressing lists with side conditions. For example, `[ x*x for x in range(1,11) if x%2 == 0]`. To express this in R using `fn$` we can write it like this which gets fairly close to the Python formulation:

```
> fn$sapply(1:10, ~if (x%%2 == 0) x^2, simplify = c)

[1]   4  16  36  64 100
```

Here is an example of string interpolation:

```
> fn$cat("pi = $pi, exp = `exp(1)`\n")

pi = 3.14159265358979, exp = 2.71828182845905
```

8. `match.funfn` and `as.function.formula`

Developers who wish to add the `fn$` capability to their own functions (so that the user does not have to prepend them with `fn$`) can use the supplied `match.funfn` function which in turn uses the `as.function.formula` function to convert formulas to functions. `match.funfn` is like the `match.fun` in R function except that it also converts formulas, not just character strings. For example with the definition of `sq` shown below the formal argument f can be a formula, character string or function as shown in the statements following:

```
> sq <- function(f, x) {
+     f <- match.funfn(f)
+     f(x^2)
+ }
> sq(~exp(x)/x, pi)

[1] 1958.912

> f <- function(x) exp(x)/x
> sq("f", pi)

[1] 1958.912

> f <- function(x) exp(x)/x
> sq(f, pi)
```

```
[1] 1958.912

> sq(function(x) exp(x)/x, pi)

[1] 1958.912
```

9. Summary

By simply extending the replacement string in `gsub` to functions, formulas and **proto** objects we obtain a function which on the surface appears nearly identical to `gsub` but, in fact, has powerful ramifications for processing.

Computational details

The results in this paper were obtained using R 2.7.2 with the packages **boot** 1.2–34, **doBy** 3.3, **grid** 2.7.2, **gsubfn** 0.3–7, **lattice** 0.17–15, **proto** 0.4–1, **quantreg** 4.20 and **reshape** 0.8.0.

R itself and all packages used are available from CRAN at http://CRAN.R-project.org/.