A. Zeileis, F. Leisch, K. Hornik, and C. Kleiber. strucchange: An R package for testing for structural change in linear regression models. Report 55, SFB "Adaptive Information Systems and Modelling in Economics and Management Science", May 2001. URL http://www.wu-wien.ac.at/am/reports.htm#55. 10

Achim Zeileis
Technische Universität Wien, Austria
zeileis@ci.tuwien.ac.at

Programmer's Niche: Macros in R

Overcoming R's virtues

by Thomas Lumley

A familiar source of questions on the R mailing lists is the newly converted R user who is trying to write SAS or Stata code in R. Bill Venables then points out to them that R is not a macro language, and gently explains that there is a much easier solution to their problems. In this article I will explain what a macro is, why it's good that R isn't a macro language, and how to make it into one.

There are two reasons for this. It has been famously observed[1] that a Real Programmer can write Fortran code in any language, and it is similarly an interesting exercise to see how R can implement macros. Secondly, there are a few tasks for which macros are genuinely useful, which is why languages like LISP, for example, provide them.

What is a macro language?

Suppose you have a series of commands

```
table(treatment, gender)
table(treatment, race)
table(treatment, age.group)
table(treatment, hospital)
table(treatment, diabetic)
```

These commands can be created by taking the skeleton

```
table(treatment, variable)
```

substituting different pieces of text for `variable`, and evaluating the result. We could also repeatedly call the `table()` function with two arguments, the first being the values of `treatment` and the second being the values of the other variable.

R takes the latter approach: evaluate the arguments then use the values. We might define

```
rxtable <- function(var){
  table(treatment, var)
}
```

Stata typically takes the former approach, substituting the arguments then evaluating. The 'substitute then evaluate' approach is called a *macro expansion*, as opposed to a *function call*. I will write this in pseudo-R as

```
rxtable <- macro(var){
  table(treatment, var)
}
```

Why not macros?

In this simple example it doesn't make much difference which one you use. In more complicated examples macro expansion tends to be clumsier. One of its advantages is that you get the actual argument names rather than just their values, which is useful for producing attractive labels, but R's lazy evaluation mechanism lets you do this with functions.

One problem with macros is that they don't have their own environments. Consider the macro

```
mulplus <- macro(a, b){
  a <- a+b
  a * b
}
```

to compute $(a+b)(b)$. This would work as a function, but as a macro would have undesirable side-effects: the assignment is not to a local copy of a but to the original variable. A call like `y <- mulplus(x, 2)` expands to `y <- {x<-x+2; x*2}`. This sets `y` to the correct value, $2x+4$, but also increments `x` by 2. Even worse is `mulplus(2, x)`, which tries to change the value of 2, giving an error.

We could also try

```
mulplus <- macro(a, b){
 temp <- a+b
 temp * b
}
```

This appears to work, until it is used when we already have a variable called `temp`. Good macro languages need some way to provide variables like `temp` that are guaranteed not to already exist, but even this requires the programmer to declare explicitly which variables are local and which are global.

The fact that a macro naturally tends to modify its arguments leads to one of the potential uses of macro expansion in R. Suppose we have a data frame

[1] "Real Programmers don't use Pascal" by Ed Post — try any web search engine

in which one variable is coded -9 for missing. We need to replace this with NA, eg,

```
library(survival)
data(pbc)
pbc$bili[pbc$bili %in% -9] <- NA
```

For multiple missing values and many variables this can be tedious and error-prone. Writing a function to do this replacement is tricky, as the modifications will then be done to a copy of the data frame. We could use the <<- operator to do the assignment in the calling environment. We then face the problem that the function needs to know the names pbc and bili. These problems are all soluble, but indicate that we may be going about things the wrong way.

We really want to take the expression

```
df$var[df$var %in% values] <- NA
```

and substitute new terms for df, var and values, and then evaluate. This can be done with the substitute() function

```
eval(substitute(
  df$var[df$var %in% values] <- NA,
  list(df=quote(pbc), var=quote(bili),
       values=-9)))
```

but this is even more cumbersome than writing out each statement by hand. If we could define a macro

```
setNA<-macro(df, var, values){
  df$var[df$var %in% values] <- NA
}
```

we could simply write

```
setNA(pbc, bili, -9)
```

Using macro expansion in R

The example using substitute() shows that macro expansion is possible in R. To be useful it needs to be automated and simplified. Adding macro to the language as a new keyword would be too much work for the benefits realised, so we can't quite implement the notation for macros that I have used above. We can keep almost the same syntax by defining a function defmacro() that has the argument list and the body of the macro as arguments.

Using this function the setNA macro is defined as

```
setNA <- defmacro(df, var, values, expr={
  df$var[df$var %in% values] <- NA
})
```

and used with

```
setNA(pbc, bili, -9).
```

The argument list in defmacro can include default arguments. If -9 were a commonly used missing value indicator we could use

```
setNA <- defmacro(df, var, values = -9, expr={
  df$var[df$var %in% values] <- NA
})
```

Macros can also provide another implementation of the 'density of order statistics' example from the R-FAQ. The density of the rth order statistic from a sample of size n with cdf F and density f is

$$f_{(r),n}(x) = \frac{n(n-1)!}{(n-r)!(r-1)!}F(x)^{r-1}(1-F(x))^{n-r}f(x).$$

The FAQ explains how to use lexical scope to implement this, and how to use substitute() directly. We can also use a macro

```
dorder <- defmacro(n, r, pfun, dfun,expr={
  function(x) {
    con <- n*choose(n-1, r-1)
    con*pfun(x)^(r-1)*(1-pfun(x))^(n-r)*dfun(x)
  }
})
```

so that the median of a sample of size 11 from an exponential distribution has density

```
dmedian11 <- dorder(11, 6, pexp, dexp)
```

In this case lexical scope may be an easier solution, but 'functions to write functions' are a standard use of macros in LISP.

So how does it work?

The function defmacro() looks like

```
defmacro <- function(..., expr){
  expr <- substitute(expr)
  a <- substitute(list(...))[-1]
  ## process the argument list
  nn <- names(a)
  if (is.null(nn)) nn <- rep("", length(a))
  for(i in seq(length=length(a))) {
    if (nn[i] == "") {
      nn[i] <- paste(a[[i]])
      msg <- paste(a[[i]], "not supplied")
      a[[i]] <- substitute(stop(foo),
                 list(foo = msg))
    }
  }
  names(a) <- nn
  a <- as.list(a)
  ## this is where the work is done
  ff <- eval(substitute(
          function(){
            tmp <- substitute(body)
            eval(tmp, parent.frame())
          },
          list(body = expr)))
  ## add the argument list
  formals(ff) <- a
  ## create a fake source attribute
  mm <- match.call()
  mm$expr <- NULL
  mm[[1]] <- as.name("macro")
  attr(ff, "source") <- c(deparse(mm),
                          deparse(expr))
  ## return the 'macro'
  ff
}
```

The kernel of `defmacro()` is the call

```
ff <- eval(substitute(
    function(){
      tmp <- substitute(body)
      eval(tmp, parent.frame())
    },
    list(body = expr)))
```

In the `setNA` example this creates a function

```
function(){
  tmp <- substitute(
        df$var[df$var %in% values] <- NA)
  eval(tmp, parent.frame())
}
```

that performs the macro expansion and then evaluates the expanded expression in the calling environment. At this point the function has no formal argument list and most of `defmacro()` is devoted to creating the correct formal argument list.

Finally, as printing of functions in R actually uses the `source` attribute rather deparsing the function, we can make this print in a more user-friendly way. The last lines of `defmacro()` tell the function that its source code should be displayed as

```
macro(df, var, values){
  df$var[df$var %in% values] <- NA
}
```

To see the real source code, strip off the `source` attribute:

```
  attr(setNA, "source") <- NULL
```

It is interesting to note that because `substitute` works on the parsed expression, not on a text string, `defmacro` avoids some of the problems with C preprocessor macros. In

```
mul <- defmacro(a, b, expr={a*b})
```

a C programmer might expect `mul(i, j + k)` to expand (incorrectly) to `i*j + k`. In fact it expands correctly, to the equivalent of `i*(j + k)`.

Conclusion

While `defmacro()` has many (ok, one or two) practical uses, its main purpose is to show off the powers of `substitute()`. Manipulating expressions directly with `substitute()` can often let you avoid messing around with pasting and parsing strings, assigning into strange places with `<<-` or using other functions too evil to mention. To make `defmacro` really useful would require local macro variables. Adding these is left as a challenge for the interested reader.

Thomas Lumley
University of Washington, Seattle
tlumley@u.washington.edu

More on Spatial Data Analysis

by Roger Bivand

Introduction

The second issue of *R News* contained presentations of two packages within spatial statistics and an overview of the area; yet another article used a fisheries example with spatial data. The issue also showed that there is still plenty to do before spatial data is as well accommodated as date-time classes are now. This note will add an introduction to the **splancs** package for analysing point patterns, mention briefly work on packages for spatial autocorrelation, and touch on some of the issues raised in handling spatial data when interfacing with geographical information systems (GIS).

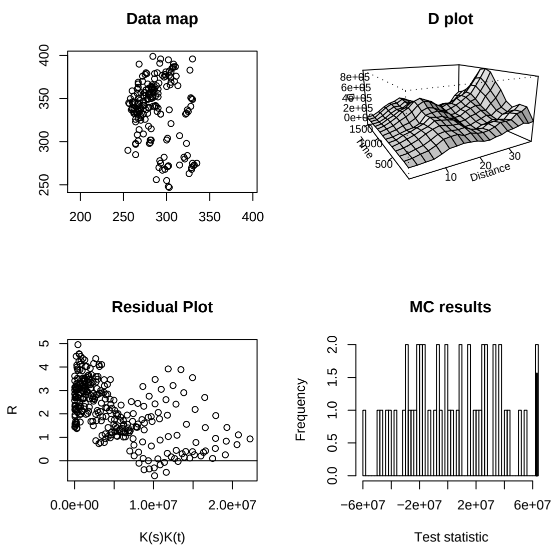

Figure 1: Burkitt's lymphoma — `stdiagn()` output.