Package 'macro'

November 26, 2025

Type Package

Title A Macro Language for 'R' Programs

Version 0.1.5

Description A macro language for 'R' programs, which provides a macro facility similar to 'SAS®'. This package contains basic macro capabilities like defining macro variables, executing conditional logic, and defining macro functions.

License CC0

Encoding UTF-8

URL https://macro.r-sassy.org, https://github.com/dbosak01/macro

BugReports https://github.com/dbosak01/macro/issues

Depends R (>= 4.0), common

Suggests sassy, knitr, rmarkdown, testthat (>= 3.0.0), rstudioapi

Imports fmtr, utils, crayon

Config/testthat/edition 3

RoxygenNote 7.3.3

VignetteBuilder knitr

NeedsCompilation no

Author David Bosak [aut, cre],
Bill Huang [ctb],
Duong Tran [ctb]

Maintainer David Bosak <dbosak01@gmail.com>

Repository CRAN

Date/Publication 2025-11-26 20:30:02 UTC

Contents

`msource` *Macro Source Function*

Description

The `msource` function is used to pre-process and source macro-enabled programs. The function first runs the macro pre-processor to evaluate any macro commands. During macro evaluation, an output file is created that contains generated code. After pre-processing, this generated code is sourced normally.

Usage

```
msource(
  pth = NULL,
  file_out = NULL,
  envir = parent.frame(),
  exec = TRUE,
  debug = FALSE,
  debug_out = NULL,
  symbolgen = FALSE,
  echo = TRUE,
  clear = TRUE,
  ...
)
```

Arguments

`pth`	The path to the R program to process. This parameter is required. It will default to the currently activated program in the development environment.
`file_out`	If you want to save or view the generated code from the `msource` function, supply a full path and file name on this parameter. Default is NULL. When NULL, the function will create a temp file for the generated code. The temp file will be deleted when processing is complete.

envir	The environment to be used for program execution, or the keyword "local". Default is the parent frame. If the parent frame is used, all variables and data initialized in the executed program will be available in the parent frame. If you do not want variables created in the parent frame, pass the quoted string "local" to have the function execute in a local environment.
exec	Whether or not to execute the output file after pre-processing. Default is TRUE. When FALSE, only the pre-processing step is performed. If a `file_out` parameter is supplied, the generated code file will still be created.
debug	If TRUE, prints lines to the console as they are processed. This information case be useful for debugging macro code. Default is FALSE.
debug_out	A path to a file to be used for debugging. If a path is supplied, debug output will be written to the file instead of the console. Default is NULL.
symbolgen	If debugger is on, this option will display the name and value of any macro variables encountered during resolution. Default is FALSE.
echo	Whether or not to echo the generated code to the console. Default is TRUE.
clear	Whether or not to clear the macro symbol table before pre-processing begins. Default is TRUE.
...	Follow-on parameters to the `source` function. See the `source` function for additional information.

Details

R does not have a native macro language. The **macro** package attempts to make up for that deficiency. The package devised a set of macro commands inspired by SAS syntax, which can be added to any R script. The macro commands are placed in R comments, prefixed by the characters "#%".

The macro commands function as pre-processor directives, and the `msource` function is the pre-processor. These commands operate as text replacement and branching functions. They allow you to perform high-level manipulation of your program before the code is executed.

Value

The results of the `source()` function, invisibly. The path of the resolved output file is also included under the "$output" list item.

How to Use

The `msource` function works very much like the Base R `source` function. You pass the path to the code file as the first parameter, and `msource` will run it. The difference is that `msource` will first pre-process the file and resolve any macro commands. The resolved code is placed by default into a temp file and then executed. If you wish to save the generated code, supply a local path on the `file_out` parameter.

The `msource` function can be run on the command line or from an R script. When run from the command line, the function will take the currently active program in RStudio as the default input. That means if you are working in RStudio, you can easily execute your macro code just by running `msource()` on the command line with no parameters.

In addition, the **macro** package registers addin menus in RStudio when installed. These addin menus can be tied to keyboard shortcuts, which can make running `msource` even easier. See `vignette("macro-setup")` to learn how to configure the keyboard shortcuts.

Macro Commands

Here is a summary of the available macro commands:

- **#%<comment>**: A macro comment.
- **#%let <variable> <- <value>**: Declares a macro variable and assigns it a value.
- **#%include '<path>'**: Inserts code from included file as text into current program.
- **#%if (<condition>)**: Begins a macro conditional block.
- **#%elseif (<condition>)**: Defines a subsequent conditional block.
- **#%else**: Identifies the default behavior in a condition.
- **#%end**: Ends a macro condition.
- **#%do <variable> = <start> %to <end>**: Defines a macro do loop block.
- **%sysfunc(<expression>)**: Evaluates an R expression as part of a macro command.
- **%symexist(<name>)**: Determines if a macro variable name exists in the macro symbol table.
- **%symput(<expression>)**: Assigns the result of an expression in the execution environment to a macro variable.
- **%nrstr(<expression>)**: A macro quoting function that masks the enclosed expression to prevent resolution.
- **#%macro <name>(<parm1>, <parm2>, ...)**: Declares the start of a macro function, identifies the function name, and defines any parameters.
- **#%mend**: Ends a macro function definition.

You can find extensive documentation for the above macro commands in the the Macro Language vignette. To access the vignette, run `vignette("macro-language")` on the R command line.

Pre-Processor

There are three main steps to processing a macro-enabled program: pre-process the input file, generate an output file, and then execute the output file.

The pre-processor works by inputting each line of the program line by line. For each line in the input script, the function will assign and replace any macro variables. The pre-processor also evaluates any macro conditions. For any macro conditions that are TRUE, the pre-processor will output that line to the generated code file. If a condition evaluates as FALSE, the lines inside that block will be ignored.

In short, the pre-processor scans the input program from top to bottom, spitting out lines or not depending on the macro conditions. This logic makes the **macro** package perfect for code generation.

Code Generation

Code generation in R is most often performed using string concatenation, and writing out strings to a file. The **macro** package gives you a much easier way to do it. Using pre-processor directives, you can write your code as normal code. These code lines will be subject to the syntax checker, and any errors in syntax will be highlighted immediately.

The **macro** package also makes it easy to construct code from code snippets. You can store your snippets in separate files, and then pull them together using `#%include` and macro logic.

The collation process is further enhanced by the **macro** debugger. The debugger allows you to solve issues in the macro code much faster and easier than doing string concatenation.

Debugger

The `msource` function has a built-in debugger. The debugger can be very useful when identifying problems in your macro-enabled program. The debugger can be activated by setting `debug = TRUE` on your call to `msource`. When activated, the debugger will by default send debug information to the R console. The debug information can show you which lines made it into the output file, and how those lines resolved. It will also echo the source call for the generated code. If an error occurs at either of these stages, the debug information will help you pinpoint which line produced the error.

For a full explanation of the debugger capabilities and several examples, see the debugging vignette at `vignette('macro-debug')`.

Output File Execution

Once the output file has been generated successfully, the `msource` function will execute it normally using the Base R `source` function. At this point the generated code runs like a normal R program, and any errors or warnings will be sent to the console.

If you do not wish to execute the generated code, use the `exec` parameter to turn off execution.

Examples

```
library(macro)

##########################################################################
# Example 1:  Hello World Macro
##########################################################################

# Get path to demo macro program
src <- system.file("extdata/Demo1.R", package = "macro")

# Display source code
# - This is the macro input code
cd <-readLines(src)
cat(paste(cd, "\n"))
# #%let a <- 1
# #%if (&a. == 1)
# print("Hello World!")
# #%else
# print("Goodbye!")
# #%end
```

```
# Macro Execute Source Code
# - Results displayed below
msource(src)
# ---------
# print("Hello World!")
# ---------
# [1] "Hello World!"

#############################################################################
# Example 2:  Perform correlation analysis between variables in MTCARS
#############################################################################

# Get path to demo macro program
src <- system.file("extdata/Demo2.R", package = "macro")

# Display source code
cd <- readLines(src)
cat(paste(cd, "\n"))
# #% Macro for Correlation Analysis
# #%macro get_correlation(dat, xvar, yvars)
#
# # Perform Analysis ------------------------------------
#
# #%do idx = 1 %to %sysfunc(length(&yvars))
#
# #%let yvar <- %sysfunc(&yvars[&idx])
# #%let xdat <- &dat$&xvar
# #%let ydat <- &dat$&yvar
# # Correlation between &xvar and &yvar
# anl_&yvar <- data.frame(XVAR = "&xvar",
#                         YVAR = "&yvar",
#                         COR = cor(`&xdat`, `&ydat`,
#                                   method = "pearson"))
# #%end
#
#
# # Bind Results ----------------------------------------
#
# #%let anl_lst <- %sysfunc(paste0("anl_", &yvars, collapse = ", "))
#
# # Combine all analysis data frames
# final <- rbind(`&anl_lst`)
#
# # Print Results
# print(final)
#
# #%mend
#
# #% Call Macro
# #%get_correlation(mtcars, mpg, c("cyl", "disp", "drat"))

# Macro Execute Source Code
```

```
msource(src)
# ---------
# # Perform Analysis ----------------------------------
#
#
# # Correlation between mpg and cyl
# anl_cyl <- data.frame(XVAR = "mpg",
#                       YVAR = "cyl",
#                       COR = cor(mtcars$mpg, mtcars$cyl,
#                                 method = "pearson"))
#
# # Correlation between mpg and disp
# anl_disp <- data.frame(XVAR = "mpg",
#                        YVAR = "disp",
#                        COR = cor(mtcars$mpg, mtcars$disp,
#                                  method = "pearson"))
#
# # Correlation between mpg and drat
# anl_drat <- data.frame(XVAR = "mpg",
#                        YVAR = "drat",
#                        COR = cor(mtcars$mpg, mtcars$drat,
#                                  method = "pearson"))
#
#
# # Bind Results ----------------------------------------
#
#
# # Combine all analysis data frames
# final <- rbind(anl_cyl, anl_disp, anl_drat)
#
# # Print Results
# print(final)
#
# ---------
#  XVAR YVAR        COR
# 1  mpg  cyl -0.8521620
# 2  mpg disp -0.8475514
# 3  mpg drat  0.6811719
#
```

`print.symtable` *Print the Macro Symbol Table*

Description

A class-specific instance of the `print` function for a macro symbol table and function list. Use `verbose = TRUE` to print the catalog as a list.

Usage

```
## S3 method for class 'symtable'
print(x, ..., verbose = FALSE)
```

Arguments

x	The format catalog to print.
...	Any follow-on parameters.
verbose	Whether or not to print the format catalog in verbose style. By default, the parameter is FALSE, meaning to print in tabular style.

Value

The object, invisibly.

See Also

msource()

Other symtable: symclear(), symget(), symput(), symtable()

Examples

```
library(macro)

# Get path to demo macro program
src <- system.file("extdata/Demo4.R", package = "macro")

# Execute source code
msource(src, echo = FALSE)

# Examine symbol table
res <- symtable()

# View results
print(res)
# # Macro Symbol Table: 3 macro variables
#   Name Value
# 1   &x     1
# 2   &y     2
# 3   &z 1 + 2
# # Macro Function List: 1 macro functions
#   Name Parameter Default
# 1 test        vl  Hello!

# View results structure
print(res, verbose = TRUE)
# $variables
# $variables$`&x`
# [1] "1"
#
```

```
# $variables$`&y`
# [1] "2"
#
# $variables$`&z`
# [1] "1 + 2"
#
#
# $functions
# $functions$test
# $functions$test$parameters
# $functions$test$parameters$vl
# [1] "Hello!"
#
#
# $functions$test$code
# [1] "print(\"&vl\")"
# attr(,"start")
# [1] 8
# attr(,"end")
# [1] 8
```

`runMSource` *Addin Function to Run msource()*

Description

This function is exposed to the addin menu to run `msource()` interactively. The function will run either the currently selected code, or the currently active program. The function sets the "envir" parameter to the global environment and the "clear" parameter to FALSE to enhance the user experience. The function takes no parameters and is only used by the addin menu.

Usage

```
runMSource()
```

Value

The results of `msource`, invisibly.

Examples

```
# Called from addin menu
runMSource()
```

runMSourceDebug *Addin Function to Run msource() in Debug Mode*

Description

This function is exposed to the addin menu to run `msource()` interactively in debug mode. The function will run either the currently selected code, or the currently active program. On the call to `msource`, it sets the "debug" and "symbolgen" parameters to TRUE. The function also sets the "envir" parameter to the global environment and the "clear" parameter to FALSE to enhance the user experience. The function takes no parameters and is only used by the addin menu.

Usage

```
runMSourceDebug()
```

Value

The results of `msource`, invisibly.

Examples

```
# Called from addin menu
runMSourceDebug()
```

symclear *Clear the Macro Symbol Table*

Description

The `symclear` function clears the macro symbol table of any stored macro variables and macro functions. The function is used to avoid contamination between one call to `msource` and the next. It is called automatically when the "clear" parameter of `msource` is set to TRUE. If the "clear" parameter is set to FALSE, you can clear the symbol table manually with the `symclear` function.

Usage

```
symclear(variables = TRUE, functions = TRUE)
```

Arguments

`variables`	Whether or not to clear the macro symbol table. Default is TRUE.
`functions`	Whether or not to clear the macro function list. Default is TRUE.

Value

The number of objects cleared, invisibly. The function also outputs a message saying how many objects were cleared.

See Also

`msource()`

Other symtable: `print.symtable()`, `symget()`, `symput()`, `symtable()`

Examples

```
library(macro)

# Get path to demo macro program
src <- system.file("extdata/Demo4.R", package = "macro")

# Display source code
# - This is the macro input code
cd <- readLines(src)
cat(paste(cd, "\n"))
# #% Create some macro variables
# #%let x <- 1
# #%let y <- 2
# #%let z <- &x + &y
#
# #% Create a macro function
# #%macro test(vl = Hello!)
# print("&vl")
# #%mend

# Execute source code
msource(src, echo = FALSE)

# View symbol table
symtable()
# # Macro Symbol Table: 3 macro variables
#   Name Value
# 1   &x     1
# 2   &y     2
# 3   &z 1 + 2
# # Macro Function List: 1 macro functions
#   Name Parameter Default
# 1 test        vl  Hello!

# Clear symbol table
symclear()
# Clearing macro symbol table...
# 4 items cleared.

# View symbol table again
symtable()
```

```
# # Macro Symbol Table: (empty)
# # Macro Function List: (empty)
```

---

symget *Get a Variable Value from the Macro Symbol Table*

Description

The symget function extracts the value of a single macro variable from the macro symbol table.

Usage

```
symget(name)
```

Arguments

name	The name of the macro variable as a quoted string, with no leading ampersand or trailing dot (."). The leading ampersand will be added automatically by the function. This parameter is required.

Value

The value of the macro variable as a character string. If the variable name is not found, the function will return an NA.

See Also

msource()

Other symtable: print.symtable(), symclear(), symput(), symtable()

Examples

```
library(macro)

# Get path to demo macro program
src <- system.file("extdata/Demo3.R", package = "macro")

# Display source code
# - This is the macro input code
cd <- readLines(src)
cat(paste(cd, "\n"))
# #% Determine appropriate data path
# #%if ("&env." == "prod")
#   #%let pth <- /projects/prod/data
# #%else
#   #%let pth <- /projects/dev/data
# #%end
```

```
# Set environment variable using symput()
symput("env", "prod")

# Macro Execute Source Code
# - set clear to FALSE to so "env" value is not removed
msource(src, echo = FALSE, clear = FALSE)

# View "pth" macro variable
res <- symget("pth")

# View results
# - Path is set to the "prod" value
res
# [1] "/projects/prod/data"
```

---

symput *Assign a Variable in the Macro Symbol Table*

Description

The symput function assigns the value of a macro variable from regular R code.

Usage

```
symput(x, value = NULL)
```

Arguments

x	The name of the macro variable to assign, passed as a quoted string with no leading ampersand or trailing dot (".."). The leading ampersand will be added automatically by the function. This parameter is required.
value	The value of the macro variable to assign. Value will be converted to a character string. This parameter is not required. If the value parameter is not supplied, the variable will be removed from the symbol table.

Value

The macro name, invisibly.

See Also

msource()

Other symtable: print.symtable(), symclear(), symget(), symtable()

Examples

```
library(macro)

# Get path to demo macro program
src <- system.file("extdata/Demo3.R", package = "macro")

# Display source code
# - This is the macro input code
cd <- readLines(src)
cat(paste(cd, "\n"))
# #% Determine appropriate data path
# #%if ("&env." == "prod")
#   #%let pth <- /projects/prod/data
# #%else
#   #%let pth <- /projects/dev/data
# #%end

# Set env macro variable using symput()
symput("env", "prod")

# Macro Execute Source Code
# - set clear to FALSE to so "env" value is not removed
msource(src, echo = FALSE, clear = FALSE)

# View "pth" macro variable
res <- symget("pth")

# View results
# - Path is set to the "prod" value
res
# [1] "/projects/prod/data"
```

`symtable` *Examine the Macro Symbol Table*

Description

The `symtable` function extracts the contents of the macro symbol table and macro function list. The symbol table information is returned as an object. The object can be printed or navigated programatically.

Usage

```
symtable()
```

Value

An object of class "symtable". The object contains a list of macro symbols and their values. It also contains a list of macro functions, their parameters, and the associated code.

See Also

msource()

Other symtable: print.symtable(), symclear(), symget(), symput()

Examples

```
library(macro)

# Get path to demo macro program
src <- system.file("extdata/Demo4.R", package = "macro")

# Display source code
# - This is the macro input code
cd <- readLines(src)
cat(paste(cd, "\n"))
# #% Create some macro variables
# #%let x <- 1
# #%let y <- 2
# #%let z <- &x + &y
#
# #% Create a macro function
# #%macro test(vl = Hello!)
# print("&vl")
# #%mend

# Execute source code
msource(src, echo = FALSE)

# Examine symbol table
res <- symtable()

# View results
print(res)
# # Macro Symbol Table: 3 macro variables
#   Name Value
# 1   &x     1
# 2   &y     2
# 3   &z 1 + 2
# # Macro Function List: 1 macro functions
#   Name Parameter Default
# 1 test        vl  Hello!

# View results structure
print(res, verbose = TRUE)
# $variables
# $variables$`&x`
# [1] "1"
#
# $variables$`&y`
# [1] "2"
#
# $variables$`&z`
```

```
# [1] "1 + 2"
#
#
# $functions
# $functions$test
# $functions$test$parameters
# $functions$test$parameters$vl
# [1] "Hello!"
#
#
# $functions$test$code
# [1] "print(\"&vl\")"
# attr(,"start")
# [1] 8
# attr(,"end")
# [1] 8
```

Index