PharmaSUG 2026 - Paper ET-178

Easy Code Generation in R: The “macro” package

David J. Bosak, Archytas Clinical Solutions

ABSTRACT

You can write a program. Or you can write a program that writes a program. That is called "code generation". Code generation is advantageous in some scenarios. One advantage is that you can parameterize the outer program, and still produce concise code in the generated program. Another advantage is that the code can be generated in such a way that it is complete, transparent, and readable: as if a human programmer wrote it. This paper will explain how to generate such code in R using the "macro" package. The "macro" package provides a meta-language inspired by SAS® Macro, and motivated specifically for the need to generate code easily and efficiently. This package is perfect for creating standard analysis, safety tables, and reusable modules of all kinds. The paper will give an overview of the system and show some simple examples to illustrate how it works.

INTRODUCTION

R offers the following strategies for code generation:

- Base R metaprogramming
- Tidy evaluation
- String concatenation

Each of the above strategies has some weaknesses. Metaprogramming in R using either Base R or tidy evaluation is extraordinarily complicated. Furthermore, it is most suitable for generating small, one-line expressions or small functions, not entire programs.

String concatenation is problematic because the code inside the strings is not parsed until it is emitted and sourced. That means there can be errors in the syntax that you don’t notice until it is too late.

The **macro** package was written to overcome these weaknesses, and do it in a simple and straight-forward way. The package offers a macro language and pre-processor that can do text-based substitution and manipulation of your code similar to SAS® Macro. The output of the pre-processor is generated code that can be saved to an external file, and executed immediately or at a later time.

Let’s take a quick overview of the package.

PACKAGE OVERVIEW

At a high level, the macro system can be broken down into three areas:

- The macro language
- The pre-processor
- Symbol table functions

First, we will examine the R macro language.

MACRO LANGUAGE SYNTAX

The **macro** package supports the most basic functionality of a macro system. That includes:

1. **Macro Comments**: Like regular R comments, but are not emitted to the output file.
2. **Macro Variables**: Allow you to assign a value to a macro variable, and use the macro variable as a text replacement token.
3. **Macro Conditionals**: If-Then-Else logic that allows you to conditionally execute a chunk of code.

4. **Macro Include**: Inserts the contents of an external file into your program.
5. **Built-in Macro Functions**: A small number of macro functions to make the system more practical.
6. **Macro Do Loops**: Copies code inside the do block for each iteration of the loop.
7. **User-Defined Macro Functions**: The ability to create a user-defined text-replacement function, with or without parameters.

The macro language functions are designed as code comments. The comments allow them to be integrated into a standard R program without causing syntax errors. Otherwise, the macro functions in the **macro** package very much resemble the macro functions in SAS.

Here is a table of SAS macro features and their equivalents in R:

	SAS	R
Comment:	`%* Here is a comment;`	`#% Here is a comment`
Assignment:	`%let x = 1;`	`#%let x <- 1`

Conditional:	`%if (&x = 1) %then %do;`
	`  %put "First condition";`
	`%end;`
	`%else %if (&x = 2) %then %do;`
	`  %put "Second condition";`
	`%end;`
	`%else %do;`
	`  %put "Default condition";`
	`%end;`
		`#%if (&x == 1)`
		`  print("First condition")`
		`#%elseif (&x == 2)`
		`  print("Second condition")`
		`#%else`
		`  print("Default condition")`
		`#%end`

Include:	`%include "myfile.sas";`	`#%include "myfile.R"`

Built in Functions:	`%sysfunc()`
	`%symexist()`
	`%symput()`
	`%nrstr()`
		`%sysfunc()`
		`%symexist()`
		`%symput()`
		`%nrstr()`

Do Loop:	`%do y = 1 %to 3;`
	`  %put Y is &y;`
	`%end;`
		`#%do y = 1 %to 3`
		`  print("Y is &y")`
		`#%end`

User-Defined Function:	`%macro test();`
	`  %put "Hello!";`
	`%mend;`
	
	`%test();`
		`#%macro test()`
		`  print("Hello!")`
		`#%mend`
		
		`#%test()`

As you can see from the above table, the R macro syntax is a blend of SAS syntax and R syntax. The design of the **macro** functions allow a SAS programmer to easily remember the function names, yet still allows them to work inside the R editing environment.

DIFFERENCES FROM SAS® MACRO

Here are some of the major differences between the SAS macro language and the R macro language:

- R macro statements do not end with a semicolon (";"). They are single-line commands, and end with a carriage return.
- R macro commands can be used anywhere in the program. They are not restricted to use inside macro functions.
- Single-quotes will not prevent macro variable resolution. In the R macro language, single and double quotes operate the same.
- All operators in the R macro language use R syntax, including the comparison operator ("==").
- R macro variables in open code must be surrounded by backticks ("`") to avoid syntax errors. Inside comments, strings, and other macro commands, they do not need backticks.

Otherwise, the R macro language is similar to SAS in terms of basic functionality.

THE MSOURCE() FUNCTION

The macro language is interpreted by a macro pre-processor. The pre-processor is contained in a special function: `msource()`.

The `msource()` function is similar to the R `source()` function. The difference is that before executing the code, the `msource()` function first pre-processes any macro language statements. The function creates a file for the resolved macro code, and then executes the resolved code.

The `msource()` function inputs the source code file as it's first parameter. It has several other parameters that can be used to modify its operation. Those parameters are:

Name	Description	Valid Values	Default
pth	The path to the program to process.	A file path. Optional.	NULL
file_out	A path for the generated code file.	A file path. Optional.	NULL
envir	The environment to use for program execution.	An R environment or the keyword "local", which will use a new local environment.	parent.frame()
exec	Whether to execute the pre-processed code.	TRUE or FALSE	TRUE
debug	Prints preprocessor input and output lines to the console.	TRUE or FALSE	FALSE
debug_out	A file path to send debug information to.	A file path. Optional.	NULL
symbolgen	Whether or not to generate symbol values during debugging.	TRUE or FALSE	FALSE
echo	Whether or not to echo the generated code to the console.	TRUE or FALSE.	TRUE
clear	Whether or not to clear the macro symbol table before pre-processing begins.	TRUE or FALSE	TRUE

For additional information on the `msource()` function, see the function documentation here.

SYMBOL TABLE FUNCTIONS

While the `msource()` function is the most important function in the **macro** package, there are some other functions worth mentioning. These functions help manage the **macro** symbol table.

Like SAS, the R macro language has a symbol table. The symbol table holds the names of macro variables and their value. There are several functions in the macro package that help manage the symbol table:

- **symtable():** Exports the macro symbol table as an object.
- **symclear():** Clears the macro symbol table.
- **symput():** Initializes a macro variable in the symbol table.
- **symget():** Gets the values of a macro variable from the symbol table.

These functions are useful when writing and debugging macros.

HOW TO USE

Now that we have covered an overview of the package, let's see some macro code in action.

WRITE MACRO CODE

The code we will use is a very simple "Hello World" program. Despite being very simple, this example will actually cover many of the basic features of the system. Here is the annotated macro code:

```
1
2  #% Example1: Hello World  1
3
4  #% Set number of iterations
5  #%let x <- 3  2
6
7  print("Macro variable %nrstr(&x) is &x..")  3  4
8
9  #% Set up loop
10 #%do i = 1 %to &x  5
11
12   #%if (&i %% 2 == 0)  6
13     print("World!")
14   #%else
15     print("Hello!")
16   #%end
17 #%end
```

Annotations Explained

#	Description
1	A macro comment introduces the program. Macro comments and all other macro statements are removed during pre-processing.
2	Illustrates a macro variable assignment. You may use either a R style "left arrow" assignment operator, or an equals sign ("=").
3	The %nrstr() function prevents macro variable resolution. This function allows us to print a macro variable name as part of a text string.

4	A macro variable to resolve. The macro variable was defined above in the %let statement. A macro variable reference is prefixed with an ampersand ("&"). Optionally, it may have a trailing dot ("..").
5	This is how to set up a macro do loop. The syntax is very similar to SAS.
6	The R macro condition is actually simpler than the SAS macro condition. For the R macro condition, you do not need the "%do", or need to close each condition with an "%end;". There is only one "%end" at the end of the conditional block.

EXECUTE MACRO CODE

There are 3 ways to execute a macro program:

1. Call the `msource()` function, passing the program name in the first parameter.
2. Call the `msource()` function, passing nothing on the first parameter. By default, the function will run the currently active program in RStudio.
3. Run the `msource` keyboard shortcut. To set up keyboard shortcuts for the macro package, see the documentation here.

After running the above macro, the following will appear in the RStudio console:

```
> msource() 1
---------
print("Macro variable &x is 3.")

     print("Hello!")

     print("World!")

     print("Hello!")

---------                          2
[1] "Macro variable &x is 3."
[1] "Hello!"
[1] "World!"                       3
[1] "Hello!"
>
```

Annotations Explained

#	Description
1	In this case, we executed the macro from the command line using the `msource()` function and no parameters.
2	This section contains the resolved code, after pre-processing. It is often helpful to see what code the pre-processor generated. If you don't want to see it, turn the "echo" parameter on `msource()` to FALSE.
3	This section contains any messages, errors, warnings, or print statements from the executed code.

The above program only demonstrates a subset of the available macro functionality. You can find a full language reference for the **macro** package here.

EXAMINE THE SYMBOL TABLE

Having run a macro program, let's go look at the symbol table. We can use the symbol table functions:

```
> symget("x")
[1] "3"
> symtable()
# Macro Symbol Table: 2 macro variables
  Name Value
1   &i     3
2   &x     3
# Macro Function List: (empty)
>
```

As shown above, the `symget()` function returns an individual macro variable value. If you want to return the entire symbol table, use `symtable()`. You can also set symbol table values directly, and clear the symbol table. See the symbol table vignette for a more complete explanation of these functions.

RUN WITH DEBUG

If you have a problem with your macro-enabled program, it is helpful to run the program in the debugger. Like this:

```
> msource(debug = TRUE, symbolgen = TRUE)
*******************************************************************************
**  Pre-Processing
*******************************************************************************
-    File In: C:/packages/Testing/macro/paper1.R
-   File Out: C:\Users\dbosa\AppData\Local\Temp\RtmpiW0Xo0/paper1.R
*******************************************************************************
[ In#][Out#]:
[   1][   1]:
[   2][    ]: #% Example1: Hello World
[   3][    ]:
[   4][    ]: #% Set number of iterations
[   5][    ]: #%let x <- 3
[   6][    ]:
SYMBOLGEN: &x = 3
[   7][   2]: print("Macro variable &x is 3.")
[   8][   3]:
[   9][    ]: #% Set up loop
SYMBOLGEN: &x = 3
SYMBOLGEN: &x = 3
[  10][    ]: #%do i = 1 %to 3
[  11][    ]: #%let i <- 1
[  12][    ]:
SYMBOLGEN: &i = 1
[  13][    ]:   #%if (&i %% 2 == 0)
[  14][    ]:     print("World!")
[  15][    ]:   #%else
[  16][   4]:     print("Hello!")
[  17][    ]:   #%end
[  18][    ]: #%let i <- 2
[  19][   5]:
SYMBOLGEN: &i = 2
[  20][    ]:   #%if (&i %% 2 == 0)
[  21][   6]:     print("World!")
[  22][    ]:   #%else
[  23][    ]:     print("Hello!")
[  24][    ]:   #%end
[  25][    ]: #%let i <- 3
[  26][   7]:
SYMBOLGEN: &i = 3
[  27][    ]:   #%if (&i %% 2 == 0)
[  28][    ]:     print("World!")
[  29][    ]:   #%else
[  30][   8]:     print("Hello!")
[  31][    ]:   #%end
[  32][    ]: #% end do
[  33][   9]:
```

The first part of the debug output shows the pre-processor operations. The pre-processor steps through the input code line by line, resolves any macro code, and determines whether to emit the line or not. You can see the input line numbers and output line numbers, as well as the resolved symbol values. If there is no output line number, that means the line was not emitted to the generated code file.

The debug output allows us to see very clearly which lines were output, which lines were not, and how the line resolved. Further, if there are errors in the macro code, the error will appear at the point it occurs. This feature is extremely helpful when debugging macro code.

The next section of the debug output shows the generated and executed code:

```
*********************************************************************************
**  Execution
*********************************************************************************

> print("Macro variable &x is 3.")
[1] "Macro variable &x is 3."

>     print("Hello!")
[1] "Hello!"

>     print("World!")
[1] "World!"

>     print("Hello!")
[1] "Hello!"

*********************************************************************************
**  End
*********************************************************************************
```

Likewise, if there are errors during execution, the error will be shown at the point it occurs. This feature allows to you easily pinpoint a problem in the generated code.

CODE GENERATION

To generate and save the code, simply run the `msource()` function as above, but supply a path on the `file_out` parameter. When a `file_out` is supplied, the function will use that path as the generated code file, instead of creating a temporary file. Here is how to do it:

```
> msource(file_out = "c:/packages/Testing/macro/test1.R")
---------
print("Macro variable &x is 3.")

     print("Hello!")

     print("World!")

     print("Hello!")

---------
[1] "Macro variable &x is 3."
[1] "Hello!"
[1] "World!"
[1] "Hello!"
>
```

The generated code file *test1.R* looks like this:

```
1
2  print("Macro variable &x is 3.")
3
4      print("Hello!")
5
6      print("World!")
7
8      print("Hello!")
9
```

Notice that all macro statements have been removed, and only the regular R code is remaining. In this way, you can produce a generated code file with only the coded needed to perform the task at hand. This code is extremely easy to read. Also, once the code file is generated, it can be run all by itself, completely independent of the **macro** package. This aspect is a huge advantage over R function libraries.

WHEN TO USE

Strictly speaking, you could use the R macro language in the same way you use the SAS macro language. The R macro language actually has all the basic the capabilities of SAS macro.

However, it not recommended for this use. The macro package should not be used as an excuse to avoid learning real R program control structures: functions, conditions and loops.

The best use case for the **macro** package is for meta-programming or code generation. The package provides a much easier set of tools for these use cases than is currently available. The package is very good for creating re-usable, parameterized template programs. Such programs give you both flexibility and concise, readable code.

CONCLUSION

Base R has inherent meta-programming capabilities. These capabilities are further enhanced by the Tidyverse, with packages such as **rlang**. However, these tools are very complicated to use, and generally not suitable for the average programmer.

The **macro** package was designed provide meta-programming capabilities to the average R programmer. The package allows you to dynamically generate code, re-use code, and parameterize code in a very convenient way. It is most useful for creating parameterized, re-useable modules, especially when you want to produce concise, readable code that can be easily transferred to a sponsor or regulatory agency.

REFERENCES

Bosak, David “A Macro Language for R Programs.” 2025. Available at https://macro.r-sassy.org

ACKNOWLEDGMENTS

The author would like to thank Bartosz Jabłoński, Duong Tran, and Chang-Yu Huang for their valuable suggestions and feedback during the early stages of **macro** package development. The package is much improved because of their suggestions.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

David J. Bosak
Archytas Clinical Solutions
dbosak01@gmail.com
www.r-sassy.org