PharmaSUG 2025 - Paper AP-078

Writing SAS MACROs in R? R functions can help!

Chen Ling, Yachen Wang, AbbVie Inc.

ABSTRACT

Repeated programming is an annoying task and will greatly impact the work efficiency. To avoid repetitive work, standard macros in SAS are developed. Currently, many companies are exploring the transition from SAS to R and programmers are starting to embrace R functions, similar to SAS macros, to handle repetitive programming, hence reduce workload and improve efficiency. This paper provides a detailed comparison of SAS MACROs and R functions, demonstrating how R functions can serve as an effective alternative for creating dynamic, reusable and robust code.

In this paper, the first section explores foundational aspects, comparing SAS MACROs and R functions in terms of overall structure and calling methods, the usage of parameters or arguments. This section also discusses lazy evaluation in setting default values, scoping rules, and implementing conditional logic. Both example SAS and R codes are provided to illustrate functionalities. The second section offers practical demonstrations using R functions for generating an Adverse Event (AE) overall table, which are crucial in clinical trial reporting. This paper aims to help readers understand the unique advantages of using R for automating their reporting tasks and provides practical examples to guide readers to develop R functions from scratch, thus facilitating a smooth transition from SAS to R.

INTRODUCTION

In the fields of clinical research, automating repetitive data processing is crucial. SAS has been the traditional choice in pharmaceutical industry for decades, and it has a powerful tool “macro” that helps researchers handle tedious duplicated programming, allowing for the automation of complex workflows. Users can easily define a macro, replace the parameter with macro variable and then call this macro for future use.

Despite the dominance of SAS in the industry, the R programming language has been steadily gaining traction within both academic and industry, due to its open-source nature, extensive package ecosystem, and robust data analysis capabilities (Wang & Ling, 2025). An increasing number of companies are under a SAS to R transition (Lee & Jeeva, 2022) and programmers are developing tools to perform repeated tasks in R, named R functions. Similar to SAS MACROs, it provides a means to encapsulate code for repeated use, thereby facilitating efficient and reproducible analyses.

Unfortunately, there are rare materials and guidance about comprehensive comparison between SAS MACROs and R functions, particularly with detailed practical examples in pharmaceutical industry field. This paper aims to bridge the gap between these two powerful tools by providing a thorough comparison of SAS MACROs and R functions. The paper is divided into two main sections: the first section examines structural and functional differences, including invocation methods, parameters/arguments handling, lazy evaluation in setting default values. Notably, the comparison will include a discussion on scoping rules, highlighting the differences between dynamic scoping in SAS and lexical scoping in R, and exploring conditional logic implementations. As a practical application, the second part of this paper will demonstrate the creation of an Adverse Event (AE) overall table using R functions. The example will underline the practical utility of R functions in generating clinical trial reports, aiming to empower analysts with the skills to leverage R function for their report automation needs.

Ultimately, this paper will provide readers with a comprehensive understanding of the advantages and limitations of both SAS MACROs and R functions. More importantly, it will equip them with the knowledge and skills to create and optimize R functions from scratch, enabling them to make informed decisions and enhance their automation capabilities.

SAS MACROS V.S. R FUNCTIONS

In general, R functions are similar to SAS MACROs in many aspects, it might be easy to read R function codes if you have SAS MACROs experience. But in terms of writing R functions, the different code setup might cause confusion to programmers. In this section, we are going to compare R functions and SAS MACROs in detail. With example codes provided, we will help SAS programmers to build their first R function.

1. STRUCTURE

SAS macros and R functions are similar in structure, they are both wrapped chunks of code that can be referenced or invoked multiple times. Below is the basic structure of SAS MACROs, the macro codes are encapsulated by **%MACRO** statement and **%MEND** statement. The "my-macro" is the name you choose for the macro, it's not mandatory but recommended to write the macro name after **%MEND**, because it can make your MACRO clear and easy to understand.

```
%MACRO my_macro;
       SAS macro codes;
%MEND my_macro;
```

SAS program 1. SAS macro structure

In R function, "my-function" is the name you choose for the function, then you need to use an assignment operator `<-` to assign the function name with the function. The keyword `function` is used to create a function, followed by codes encapsulated within curly brackets {}.

```
my_function<-function(){
       R function codes
       return(expression) ###Comment: This is optional
}
```

R program 1. R Function structure

As you might notice, there is a `return(expression)` function in R, which is a little different than `%return` in SAS. In R, `return()` can output `expression` as the results of the function, if `return()` is not used, the last evaluated expression will be implicitly returned. Also, the explicit `return()` is also used to return a value immediately from the function and prematurely end the function. Similarly, in SAS `%return` can cause termination of current executing macro, but it cannot return any value from the macro.

2. CALLING A SAS MACRO/ R FUNCTION

In SAS, we will add a percentage sign % in front of the MACRO name to invoke the MACRO.

```
%my_macro;
```

In R, we will add brackets after the function name to call the function. However, if the function is returning a dataset, a good practice is to assign the function to a variable for future use.

```
my_function()

my_data<-my_function()
```

It's important to note that SAS is not case sensitive, meaning you can use all caps, like **%MY_MACRO**, to call a macro. In contrast, R is case sensitive, so you must call functions exactly by their name, respecting the case.

3. ADDING PARAMETERS/ARGUMENTS TO SAS MACRO/ R FUNCTION

Adding parameters into SAS MACROs can greatly enhance its capabilities and flexibility, pre-defined values will be passed into the MACRO through parameters. For example, in SAS program 2 parameter `sortvar` and `make` are included in the brackets after the macro name in **%macro** statement. In this macro call, `&sortvar` macro variable was assigned with Cylinders and `&make` macro variable was given value "Audi". The SAS macro will first select all observations with make equals to "Audi" and then sort ascendingly by Cylinders.

```
%macro my_macro(sortvar=,make=);
data cars;
```

```
      set sashelp.cars;
      where make="&make";
run;
proc sort data=cars;
      by &sortvar;
run;
%mend my_macro;

%my_macro(sortvar=Cylinders, make=Audi)
```

SAS program 2. SAS macro with parameters

The "parameters" can also be added into R functions; however, this time they are called "arguments" instead. With the powerful R package {tidyverse}, we are able to do the same thing in a more efficient way. In R program 2, the dataset `mtcars` are filtered by `cylinders` equal to `6` first and then sorted ascendingly by mpg.

```
my_function<-function(sortvar,Cylinders){
  df<-mtcars%>%filter(cyl==Cylinders)%>%arrange({{sortvar}})
  return(df)      ###Comment: This is optional
}

my_data<-my_function(sortvar=mpg,Cylinders=6)
```

R program 2. R function with arguments

3.1 Default value for parameters/arguments

Setting default value is very similar between SAS MACROs and R functions, in SAS MACROs we use "`parameter=default-value`" in the parentheses after the macro name in `%macro` statement. In the below code, `Cylinders` is the default value for `sortvar` and `"Audi"` is the default value for `make`.

```
%macro my_macro(sortvar=Cylinders, make="Audi");
```

By default, macro variables always store data in character format (Slaughter & Delwiche, 2003). Even when you assign a numeric value to a macro variable, it is treated as a string of characters. We need to convert the string back to a numeric format to perform numeric operations.

As shown in below R codes, we use "`argument=default-value`" in parentheses to assign default values to arguments. Different from SAS MACRO, the arguments are able to store both character and numeric values.

```
function(sortvar=mpg,Cylinders=6){…}
```

3.1.1 Lazy evaluation in R functions

Also, default values can be defined in terms of other arguments due to the lazy evaluation of arguments in R (Wickham, 2019). Lazy evaluation means the arguments will be evaluated only if they are used. Implementing lazy evaluation in a programming language can enhance computation efficiency and reduce memory consumption, especially when dealing with very complicated computations, since the arguments will be evaluated only if needed. However, sometimes this would also make the codes more difficult for programmers to understand. As shown in R program 3, the argument `y` depends on `x`, the argument `z` can even be determined by the variables (`a` and `b`) defined later in the function. The lazy evaluation can ensure that y is not evaluated as 5*10, instead the evaluation is performed later when y is called in the local environment by using 2*10.

```
###examples for lazy evaluation of arguments in R
lazy_eval<-function(x=5,y=x*10,z=a+b){
  x=2
  a=3
  b=4
  c(x,y,z)
}

lazy_eval()
[1]  2 20  7    ###Results in R
```

R program 3 Lazy evaluation for arguments

3.1.2 "Lazy evaluation" in SAS Macros

SAS MACROs do not implement lazy evaluation to the default parameters in the same way that R does. However, if you try similar code in a SAS MACRO, you will get the same results as shown in SAS program 3, why is that?

```
/*examples for "lazy evaluation" of parameters in SAS*/
%macro lazy(x=5,y=%eval(&x*10),z=%eval(&a+&b));
%put _user_; /*This would display symbol tables in the log*/
%let x=2;
%let a=3;
%let b=4;
%put (&x &y &z);
%mend;

%lazy()
(2 20 7)    /*Results in SAS log*/
```

SAS program 3 "Lazy evaluation" for parameters

The reason for getting the same results as lazy evaluation in R is because of the local symbol table in SAS. In SAS, the word scanner would read in the SAS codes, tokenize them and pass them to the compiler. When the word scanner detects a macro in the codes, it would automatically trigger a macro processor, which would create a local symbol table to store parameters as macro variables (Lyons, 2004). In our case, the local symbol table would store `5` as the value of `x`, '`%eval(&x*10)`' as the value of `y` and '`%eval(&a+&b)`' as the value of `z`. We will talk more about symbol tables and variable scoping further in section 4.

Once the parameters in the macro statement are stored in the local symbol table, SAS proceeds to read the code inside of the macro, updating the value of `x` to `2` and adding `a`, `b` into the local symbol table. Then in `%put (&x &y &z)` statement, SAS searches for and retrieves the value of `x, y, z` in the local symbol table, then writes the value into the log. For example, first `%eval(&a+&b)` is initially retrieved as the value of z, then the values of a and b will also be retrieved and substituted, resulting in `%eval(3+4)`. In the following execution, value `7` will be put in the log as the resolution of `z`.

3.2 Call parameters/arguments

In SAS MACRO, we only need to add ampersand (&) in front of the parameter to call it. For example, in SAS program 2 parameter `sortvar` are called by `&sortvar`. However, it's okay to call the arguments directly by their name in base R, in R program 2 `Cylinders` are called directly in the function.

3.2.1 Pre-execution steps in SAS

Moreover, due to the pre-execution steps (Lyons, 2004) in SAS, we are able to concatenate `cars_` and `&make` directly to use as the newly generated dataset name in SAS program 4. Macro processor first retrieves macro variables, after which the concatenated name `cars_Audi` is sent to compiler before execution.

```
###example of pre-execution steps
%macro cars(make=);
data cars_&make;
      set sashelp.cars;
      where make="&make";
run;

%mend;
%cars(make=Audi);
```

SAS program 4 Pre-execution steps

Nevertheless, R does not perform pre-execution, we can use `paste0()` function to perform dynamically naming. With the help of `assign()` function, we can assign datasets or values to the names generated by `paste0()`.

3.2.2 Tidy evaluation in R

As mentioned earlier, in base R if we want to add an argument to specify a column, we can call the arguments directly by their names in the function. But this will not work properly when we are using the powerful {tidyverse} packages. As shown in R program 4, an error will occur because column cannot be found in the current execution environment. To understand the cause of this issue, we need to know why {tidyverse} function `group_by()` is searching for `group_var` instead of `cyl`.

Herein, I would like to introduce the idea of **quasiquotation**, an important component of tidy evaluation, which can divide function arguments into two classes: quoted and evaluated(unquoted) (Altman, Behrman, & Wickham, 2021). In quasiquotation, quoting means that the argument is captured as is from the environment without evaluation, and the quoted arguments can be evaluated in the proper environment by unquoting. Functions in {tidyr} like `group_by()` will quote the argument without evaluating it, and that is the reason for why `group_var` is being searched in the data `mtcars`.

```
###example of not using tidy evaluation
library(tidyverse)
library(rlang)

my_function1<-function(group_var, mpg_filter, avg_var){
  df<-mtcars%>%
    group_by(group_var)%>%
    filter(mpg_filter)%>%
    summarise(mean=mean(avg_var))
return(df)       ###Comment: This is optional
}

my_data1<-my_function1(group_var=cyl, mpg_filter= mpg > 20, avg_var= mpg)

Error in `group_by()`:
! Must group by variables found in `.data`.
✕ Column `group_var` is not found.
```

R program 4 Example of not using tidy evaluation

To avoid this, we need to capture the expression of `group_var` from the global environment and then evaluate the expression at the place it is used to get its value: `cyl`. Thanks to the function `enquo()` in {rlang} package, we are able to create a data structure called **quosure** to store both the expression and the environment. If you add a `print(group_var)` step after the `enquo()` in the function, you can inspect the content of the quosure you've just created. As shown at the end of R program 5, expr stands for the expression captured, and env stands for the environment the expression should be evaluated in.

After group_var is quoted, we need to tell `group_by()` to unquote it by using `!!` (pronounced bang-bang), in R program 5 we use `!!group_var` to unquote and evaluate `group_var`. With the release of {rlang} 0.4.0, `{{}}` (pronounced curly-curly), a more convenient way was introduced to quote and unquote arguments. Now we can use `{{}}` to replace `enquo()` and `!!`, for example, in R program 5 we can use `{{mpg_filter}}` directly as the filtering condition inside `filter()`.

```
###example of tidy evaluation
library(tidyverse)
library(rlang)
my_function2<-function(group_var, mpg_filter, avg_var){
  group_var=enquo(group_var)
  print(group_var) ###This is just for inspection purpose
  df<-mtcars%>%
    group_by(!!group_var)%>%
    filter({{mpg_filter}})%>%
    summarise(mean=mean({{avg_var}}))
  return(df)       ###Comment: This is optional
}
```

```
my_data2<-my_function2(group_var=cyl, mpg_filter= mpg > 20, avg_var= mpg)
<quosure>
expr: ^cyl
env:  global
```

R program 5 Example of tidy evaluation

There is another scenario where the arguments are given in string format, such as `group_var="cyl"`, this is especially common when you are developing an R shiny application (Ling & Wang, 2025). In this scenario, our usual quoting method `enquo()` or `{{}}` method may not work, we need to use the quoting functions that can capture strings, such as `ensym()` and `parse_expr()`(a kind of inverse to `deparse()` function). As shown in R program 6, `ensym()` is able to capture a string and return it as a quoted symbol, instead of a quosure, which means we will lose track of the environment while using `ensym()`. The other function `parse_expr()` can convert the string into an expression, and the `eval()` function can unquote the expression, functioning like `!!`. Similar to `ensym()`, `parse_expr()` will not capture the environment as well.

In addition, in R program 6, we add a new argument `avg_name` to assign a name to the column created by `summarise()`. In this context, we need to use "`:=`(colon-equals) instead of "`=`" to assign value to `!!ensym(avg_name)`, because R does not allow using "`=`" for value assignment to an expression.

```
###example of tidy evaluation
library(tidyverse)
library(rlang)

my_function3<-function(group_var, mpg_filter, avg_var ,avg_name){

  df<-mtcars%>%
    group_by(!!ensym(group_var))%>%
    filter(eval(parse_expr(mpg_filter))) %>%
    summarise(!!ensym(avg_name):=mean(!!ensym(avg_var)))

    return(df)      ###Comment: This is optional
}

my_data3<-my_function3(group_var="cyl", mpg_filter= "mpg > 20", avg_var= "mpg",
,avg_name="mpg_mean")
```

R program 6 Example of tidy evaluation with string argument

4. SCOPE OF VARIABLES

The concept of global and local variables is present in both SAS and R, however, the scoping rules are different in the two languages. In addition, variables are stored in distinct memory locations: SAS uses symbol tables, while R employs environments to manage global and local variables.

4.1 Scope of variables in SAS

4.1.1 Symbol table

In SAS, macro variables are managed within a special memory area called the symbol table, which is crucial for dynamic code substitution. The symbol table consists of two main types: global symbol table and local symbol table. There is only one global symbol table created when the SAS session starts and is deleted when the SAS session ends. However, various local symbol tables are temporarily created during macro execution under specific conditions, then they will be deleted after the execution.

4.1.2 Scoping rules

In SAS, the scope of macro variables is determined by whether they are defined in a global or local context. By default, any macro variable created outside of a macro definition automatically resides in the Global symbol table. This means it is available throughout the entire SAS session. Within a macro definition, the scoping rules prioritize existing local variables first, which means the new defined macro variables are local variables inside a macro. If a macro has parameters, those variables are automatically local to the macro. SAS will search for a macro variable within the local symbol table first, and if not found, it will look "upwards" through any calling macros, ultimately checking the Global symbol table as a last resort.

4.1.3 Dynamic scoping

From the above sections, you probably have already known how the scoping works in SAS, would you please help me to predict the value of `x1` and `y1` after running all codes in SAS program 5?

```
/*example of dynamic scoping*/
%let x1=9;
%let y1=7;
%macro my_macro2;
%let x1=19;
%let y1=17;
%mend;

%my_macro2()
```

SAS program 5 Example 1 of dynamic scoping

The answer for this test is: `x1=19` and `y1=17`, some people may find out the answer is surprising, why is SAS using the "local" variables assigned inside a macro to replace the global ones? Or our question can be: while SAS macro is executing, are all macro variables defined inside the macro local ones? The answer is obvious: no, not all the time. In this case, while we are assigning `19` to `x1` in `my_macro2`, SAS will search in the local symbol table for `x1` first. If `x1` is not found in the local symbol table, SAS then will search in the global symbol table. Since `x1` has already been in the global symbol table, in this case, SAS will directly replace the value of `x1` with `19` in the global symbol table.

Let's delve deeper to understand what's happening behind the scenes. Unlike most modern programming languages that use **lexical scoping** (also known as static scoping), SAS generally utilizes **dynamic scoping** as its scoping rule. On the one hand, lexical scoping will look up values of variables based on the lexical (textual) content defined by where the variable or function is called. On the other hand, in dynamic scoping, if a variable's scope is a macro, then its scope is the time-period during which the macro is executing. This means the determination of the variable's assignment relies on the runtime context, making it challenging to predict. Since dynamic scoping happens during the execution after the compilation, neither the macro processor nor compiler can tell whether an assignment statement is meant to assign a local macro variable or a higher scope macro variable that may exist at runtime (Joe, 2016). Let's see a more straightforward example for dynamic scoping in SAS program 6. In this example, when SAS is resolving the value for `&x`, it first considers the scope of its runtime. Since the macro `inner` is running in the `outer`, `x=6` defined in `outer` will be used in `inner` to put in the log.

```
/*example of dynamic scoping*/
%let x=3;
%macro inner();
%put &x;
%mend;
%macro outer();
%let x=6;
%inner();
%mend;
%outer()
```

SAS program 6 Example 2 of dynamic scoping

4.1.4 Controlling the scope of macro variables

As I mentioned, SAS macro "generally" employs dynamic scoping, however, there are ways to explicitly specify the macro variables to be local which can enforce lexical scoping. In SAS program 7, `x2` is used as the parameter in the macro, which can automatically make it a local variable. SAS also provides the `%local` statement, which can turn `y2` into a local macro variable, and since `x2` and `y2` are both local variables, the value global `x2` and `y2` will not change.

`%local` statement ensures that a macro variable is confined to the local environment of the macro where it is declared. If the macro variable already exists, `%local` will be ignored; otherwise, it will create a macro variable in the local symbol table. The good practice is that we should always use `%local` to specify variables defined inside a macro to avoid any conflict.

In addition, we can use %global to explicitly specify a variable inside a macro to be global variable. However, for a multi-layer nested scoping structure, you cannot force a variable to be in an intermediate scope between the global scope and most local scope.

```
/*example of explicitly specified local macro variables*/
%let x2=9;
%let y2=7;
%macro my_macro2(x2);
%local y2;
%let x2=19;
%let y2=17;
%mend;

%my_macro2()
```

SAS program 7 Example of explicitly specified local macro variables

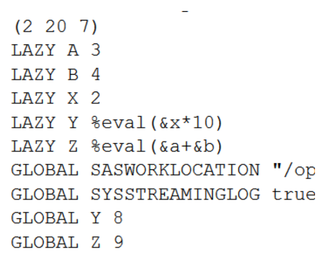
4.1.5 How to check global and local symbol table

Debugging and verifying the placement of macro variables is crucial for ensuring the accuracy of SAS programs. The `%PUT` statement with the `_user_` keyword can be used to write a list of all user-defined macro variables, their values, and their respective scopes to the SAS log (Buchecker, 2020). In SAS program 5, `%put _user_` is used to print the global and local symbol tables in the log. As shown in Figure 1, there are some extra variables in the global symbol table, please note that those are SAS system generated variables, we can ignore them most of the time.

```
/*example of checking symbol tables*/
%let y=8;
%let z=9;
%macro lazy(x=1,y=%eval(&x*10),z=%eval(&a+&b));
%let x=2;
%let a=3;
%let b=4;
%put (&x &y &z);
%put _user_;
proc print data=sashelp.vmacro;
run;
%mend;

%lazy()
```

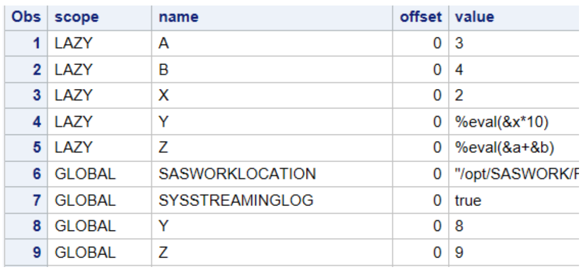
SAS program 8 Example of checking symbol tables



```
(2 20 7)
LAZY A 3
LAZY B 4
LAZY X 2
LAZY Y %eval(&x*10)
LAZY Z %eval(&a+&b)
GLOBAL SASWORKLOCATION "/op
GLOBAL SYSSTREAMINGLOG true
GLOBAL Y 8
GLOBAL Z 9
```

Figure 1 Local and global symbol tables printed in log

Alternatively, the `sashelp.vmacro` view can be accessed, which provides a comprehensive dataset of all macro variables along with their names, values, and scopes, allowing users to directly inspect and manage both local and global symbol tables effectively.



Obs	scope	name	offset	value
1	LAZY	A	0	3
2	LAZY	B	0	4
3	LAZY	X	0	2
4	LAZY	Y	0	%eval(&x*10)
5	LAZY	Z	0	%eval(&a+&b)
6	GLOBAL	SASWORKLOCATION	0	"/opt/SASWORK/F
7	GLOBAL	SYSSTREAMINGLOG	0	true
8	GLOBAL	Y	0	8
9	GLOBAL	Z	0	9

Figure 2 The content of sashelp.vmacro

4.2 Scope of variables in R

4.2.1 Environment

Similar to SAS symbol tables, R uses a memory area called an environment to store variables and their values. Most environments have a higher-level environment, known as a parent, and even the global environment has a parent: the empty environment. Notably, the empty environment is the only one without a parent. Grasping the concept of environments is essential to better understand scoping in R.

4.2.2 Scoping rules

To understand the scoping rules, we have to know how R is binding values to the free variables. If the value of a variable cannot be found inside the function, then it will look at the environment in which the function is defined (Peng, 2022). If the search fails in this environment, it will continue to search in the parent environment. The process continues through successive parent environments until reaching the top-level environment, which is typically either the global environment (workspace) or a package's namespace. Beyond the top-level environment, the search persists through the search list until it encounters the empty environment.

4.2.3 Lexical scoping (static scoping)

Just like most modern programming languages, R utilizes lexical scoping. Lexical scoping does not consider when the function is called, it's only based on where the function is defined. Let's revisit the example from SAS program 6; below is R program 7, which has a similar structure but yields different results from SAS. When `inner()` is called inside `outer()`, it first searches for x in the local environment of `inner()`. Since x is not defined there, it proceeds to search in the environment where `inner()` was defined, which is the global environment, rather than the environment of `outer()`, as would happen in SAS. Finally, it will find the `x` defined as 3 in global environment and print it out.

```
###example of lexical scoping
x=3
inner<-function(){
  print(x)
}
outer<-function(){
  x=6
  inner()
}
outer()
```

R program 7 Example of lexical programming

4.2.4 Controlling the scope of variable

In R, the `<-` operator is used to assign a value to a variable in the current environment. In situations where you might need to assign value in a parent environment, the super assignment operator `<<-` is used. Alternatively, you can control the scope by assigning a value to a variable in the global environment using `.GlobalEnv$variable <- value`, similar to `%global` in SAS.

However, these methods are generally discouraged because they can produce side effects by altering variables outside their intended scope, beyond what the function explicitly returns. This can lead to unexpected issues, especially for users unfamiliar with the functions involved. For instance, an unexpected outcome might overwrite existing variables in the calling environment, potentially impacting subsequent code. It's recommended to avoid using these scope-controlling techniques in programming to maintain clarity and prevent unintended consequences.

5. CONDITIONAL LOGIC

Unlike the standard SAS IF-THEN-ELSE statement, macro statements are used when writing codes in SAS MACROs. The below codes show the basic structure of *%IF-%THEN-%ELSE* macro statements, although they look similar as the standard SAS codes, please do not mix them up and remember to only use them inside macros.

```
%IF condition1 %THEN action1;
      %ELSE %IF condition2 %THEN action2;
      %ELSE action3;
%IF condition4 %THEN %DO;
      action4;
%END;
```

SAS program 9 Conditional logic in SAS macro

Conditional logic in R functions is the same as standard R codes. Like SAS, we still use if-else for conditional logic, but the structure is a little different. Herein, the conditions are enclosed in parentheses and the actions are enclosed in curly brackets. The curly brackets {} can enclose as many actions as you want, so no `%DO` statement is needed when we have multiple lines of actions.

```
if (condition){
      action
} else if (condition){
      action
} else {
      action
}
```

R program 8 Conditional logic in R

The following table makes a comparison between SAS macro and R function from different perspectives.

#	Aspect	SAS macro	R function

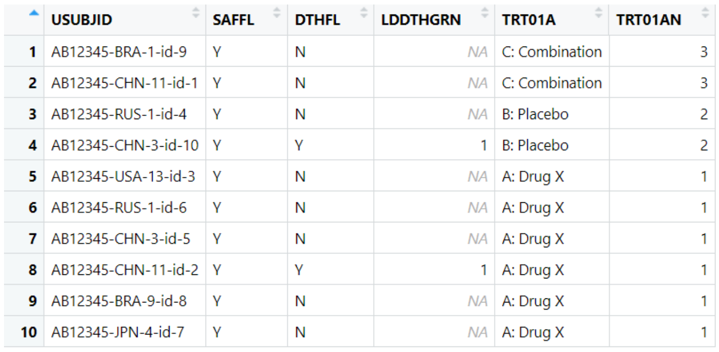
1	Structure	%MACRO macro_name(parameters);
		macro body;
		%MEND macro_name;
			function_name <-function(arguments){
			function body
			}

2	Name	Starts with a letter or underscore, contain only numbers, letters and underscore. Maximum length is 32	Starts with a letter, contain only letters, numbers, underscore or period
3	Invocation	%macro_name(parameters)	function_name(argments)
4	Default value	Can set default value to parameters, which can be defined in terms of other arguments based on local symbol table	Can set default value to arguments, which can be defined in terms of other arguments based on lazy evaluation
5	Call parameters/ arguments	Add & in front of parameter names: &parameter	For base R codes: directly use argument name For {tidyverse} system, use tidy evaluation: enquo() and !! or {{argument1}}, and use := when assigning values to an expression
6	Scoping rules	Dynamic scoping with symbol tables	Lexical Scoping with environments
7	Conditional logic	Different than standard SAS codes %IF-%THEN-%ELSE	Same as standard R codes if (condition){action}else {action}

R FUNCTION DEMOS FOR GENERATING AE TABLES

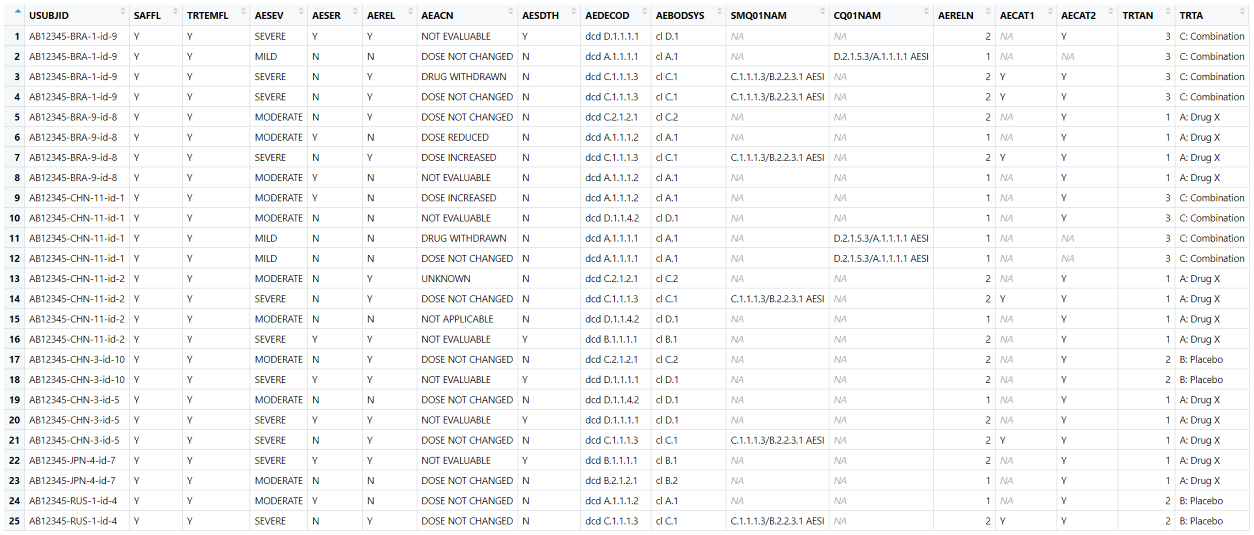
SOURCE DATA

In this paper, we use **{random.cdisc.data}** to randomly generate CDISC compliance ADSL and ADAE data as examples. Herein, we only keep the variables we use in generating AE tables.



	USUBJID	SAFFL	DTHFL	LDDTHGRN	TRT01A	TRT01AN
1	AB12345-BRA-1-id-9	Y	N	NA	C: Combination	3
2	AB12345-CHN-11-id-1	Y	N	NA	C: Combination	3
3	AB12345-RUS-1-id-4	Y	N	NA	B: Placebo	2
4	AB12345-CHN-3-id-10	Y	Y	1	B: Placebo	2
5	AB12345-USA-13-id-3	Y	N	NA	A: Drug X	1
6	AB12345-RUS-1-id-6	Y	N	NA	A: Drug X	1
7	AB12345-CHN-3-id-5	Y	N	NA	A: Drug X	1
8	AB12345-CHN-11-id-2	Y	Y	1	A: Drug X	1
9	AB12345-BRA-9-id-8	Y	N	NA	A: Drug X	1
10	AB12345-JPN-4-id-7	Y	N	NA	A: Drug X	1

Figure 3 Randomly generated ADSL



	USUBJID	SAFFL	TRTEMFL	AESEV	AESER	AEREL	AEACN	AESDTH	AEDECOD	AEBODSYS	SMQ01NAM	CQ01NAM	AERELN	AECAT1	AECAT2	TRTAN	TRTA
1	AB12345-BRA-1-id-9	Y	Y	SEVERE	Y	Y	NOT EVALUABLE	Y	dcd D.1.1.1.1	cl D.1	NA	NA	2	NA	Y	3	C: Combination
2	AB12345-BRA-1-id-9	Y	Y	MILD	N	N	DOSE NOT CHANGED	N	dcd A.1.1.1.1	cl A.1	NA	D.2.1.5.3/A.1.1.1.1 AESI	1	NA	NA	3	C: Combination
3	AB12345-BRA-1-id-9	Y	Y	SEVERE	N	Y	DRUG WITHDRAWN	N	dcd C.1.1.1.3	cl C.1	C.1.1.1.3/B.2.2.3.1 AESI	NA	2	Y	Y	3	C: Combination
4	AB12345-BRA-1-id-9	Y	Y	SEVERE	N	Y	DOSE NOT CHANGED	N	dcd C.1.1.1.3	cl C.1	C.1.1.1.3/B.2.2.3.1 AESI	NA	2	Y	Y	3	C: Combination
5	AB12345-BRA-9-id-8	Y	Y	MODERATE	N	Y	DOSE NOT CHANGED	N	dcd C.2.1.2.1	cl C.2	NA	NA	2	NA	Y	1	A: Drug X
6	AB12345-BRA-9-id-8	Y	Y	MODERATE	Y	N	DOSE REDUCED	N	dcd A.1.1.1.2	cl A.1	NA	NA	1	NA	Y	1	A: Drug X
7	AB12345-BRA-9-id-8	Y	Y	SEVERE	N	Y	DOSE INCREASED	N	dcd C.1.1.1.3	cl C.1	C.1.1.1.3/B.2.2.3.1 AESI	NA	2	Y	Y	1	A: Drug X
8	AB12345-BRA-9-id-8	Y	Y	MODERATE	Y	N	NOT EVALUABLE	N	dcd A.1.1.1.2	cl A.1	NA	NA	1	NA	Y	1	A: Drug X
9	AB12345-CHN-11-id-1	Y	Y	MODERATE	Y	N	DOSE INCREASED	N	dcd A.1.1.1.2	cl A.1	NA	NA	1	NA	Y	3	C: Combination
10	AB12345-CHN-11-id-1	Y	Y	MODERATE	N	N	NOT EVALUABLE	N	dcd D.1.1.4.2	cl D.1	NA	NA	1	NA	Y	3	C: Combination
11	AB12345-CHN-11-id-1	Y	Y	MILD	N	N	DRUG WITHDRAWN	N	dcd A.1.1.1.1	cl A.1	NA	D.2.1.5.3/A.1.1.1.1 AESI	1	NA	NA	3	C: Combination
12	AB12345-CHN-11-id-1	Y	Y	MILD	N	N	DOSE NOT CHANGED	N	dcd A.1.1.1.1	cl A.1	NA	D.2.1.5.3/A.1.1.1.1 AESI	1	NA	NA	3	C: Combination
13	AB12345-CHN-11-id-2	Y	Y	MODERATE	N	Y	UNKNOWN	N	dcd C.2.1.2.1	cl C.2	NA	NA	2	NA	Y	1	A: Drug X
14	AB12345-CHN-11-id-2	Y	Y	SEVERE	N	Y	DOSE NOT CHANGED	N	dcd C.1.1.1.3	cl C.1	C.1.1.1.3/B.2.2.3.1 AESI	NA	2	Y	Y	1	A: Drug X
15	AB12345-CHN-11-id-2	Y	Y	MODERATE	N	N	NOT APPLICABLE	N	dcd D.1.1.4.2	cl D.1	NA	NA	1	NA	Y	1	A: Drug X
16	AB12345-CHN-11-id-2	Y	Y	SEVERE	Y	Y	NOT EVALUABLE	Y	dcd B.1.1.1.1	cl B.1	NA	NA	2	NA	Y	1	A: Drug X
17	AB12345-CHN-3-id-10	Y	Y	MODERATE	N	Y	DOSE NOT CHANGED	N	dcd C.2.1.2.1	cl C.2	NA	NA	2	NA	Y	2	B: Placebo
18	AB12345-CHN-3-id-10	Y	Y	SEVERE	Y	Y	NOT EVALUABLE	Y	dcd D.1.1.1.1	cl D.1	NA	NA	2	NA	Y	2	B: Placebo
19	AB12345-CHN-3-id-5	Y	Y	MODERATE	N	N	DOSE NOT CHANGED	N	dcd D.1.1.4.2	cl D.1	NA	NA	1	NA	Y	1	A: Drug X
20	AB12345-CHN-3-id-5	Y	Y	SEVERE	Y	Y	NOT EVALUABLE	Y	dcd D.1.1.1.1	cl D.1	NA	NA	2	NA	Y	1	A: Drug X
21	AB12345-CHN-3-id-5	Y	Y	SEVERE	N	Y	DOSE NOT CHANGED	N	dcd C.1.1.1.3	cl C.1	C.1.1.1.3/B.2.2.3.1 AESI	NA	2	Y	Y	1	A: Drug X
22	AB12345-JPN-4-id-7	Y	Y	SEVERE	Y	Y	NOT EVALUABLE	Y	dcd B.1.1.1.1	cl B.1	NA	NA	2	NA	Y	1	A: Drug X
23	AB12345-JPN-4-id-7	Y	Y	MODERATE	N	N	DOSE NOT CHANGED	N	dcd B.2.1.2.1	cl B.2	NA	NA	1	NA	Y	1	A: Drug X
24	AB12345-RUS-1-id-4	Y	Y	MODERATE	Y	N	DOSE NOT CHANGED	N	dcd A.1.1.1.2	cl A.1	NA	NA	1	NA	Y	2	B: Placebo
25	AB12345-RUS-1-id-4	Y	Y	SEVERE	N	Y	DOSE NOT CHANGED	N	dcd C.1.1.1.3	cl C.1	C.1.1.1.3/B.2.2.3.1 AESI	NA	2	Y	Y	2	B: Placebo

Figure 4 Randomly generated ADAE

1. AE OVERALL TABLE

In this section, we will present a practical demonstration of an R function for generating an adverse event table. This table offers an overview of all treatment-emergent adverse events and deaths, utilizing the features of the R function discussed earlier.

1.1 Data pre-process for ADSL and ADAE

For generating the AE tables, we need two ADaM datasets: ADAE and ADSL. Before using the R functions in the next section, we need to do some data pre-processing to generate two records (for total and active total calculation) for each observation in both datasets. For our programming convenience, AE-related variables (flags) have already been populated in ADAE, and TRT01A has replaced TRTA in ADAE to keep consistent with ADSL. In addition, we also need a dataset `n_of_trt` to contain the number of subjects for each treatment, total and active total.

```
adsl_act<-adsl%>%
  filter(TRT01AN %in% tx_active)%>%
  mutate(TRT01AN = 98)
adsl_total<-adsl%>%
  group_by(USUBJID)%>%
  filter(row_number()==1)%>%
  mutate(TRT01AN=99)%>%
  rbind(adsl)%>%
  rbind(adsl_act)
adae_act<-adae%>%
  filter(TRT01AN %in% tx_active)%>%
  mutate(TRT01AN = 98)
teae<-adae%>%
  mutate(TRT01AN=99)%>%
  rbind(adae)%>%
  rbind(adae_act)%>%
  filter(TRTEMFL=="Y")
n_of_trt<-adsl_total%>%
  group_by(TRT01AN,USUBJID)%>%
  slice(1)%>%
  group_by(TRT01AN)%>%
  count()
```

R program 9 Data pre-processing for ADSL and ADAE

1.2 Count numbers of subjects for different events

In this demo, two functions, `n_count1()` and `n_count2()` have been developed to count the number of subjects with each event. The `n_count1()` function is designed to count the number of subjects that meet a specified subsetting condition, while `n_count2()` can deal with different scenarios and pass the condition to `n_count1()`. With this nested structure, we are able to use only one function `n_count2()` for all the events.

In R program 10 code chunk 1, we first subset the dataset with the condition from the argument and then `summarize()` is used to get the number of subjects with the specified event. This data is merged with the `n_of_trt` using `right_join()` to calculate the proportion of subjects experiencing the event across various treatment groups. The use of `right_join()` and followed by `mutate()` is essential, as it helps to generate a table with zeros if no subjects satisfy the condition of a specified event. Next, in the code chunk 2, the data is reshaped from long to wide format with `pivot_wider()`, we will introduce reshaping in detail later in this conference (Wang & Ling, 2025). After reshaping, order and category variables are included to aid in the generation of the table dataset.

```
library(tidyverse)
library(rlang)

n_count1<-function(data,                   ### Dataset used for counting
                   n_trt,                  ### Dataset containing # of subjects by treatments
                   cond=NULL,              ### The condition for subsetting the dataset,
                                           ### should be related to count_var
                   tx_var,                 ### The numeric variable contains treatment
                                           ### information; e.g. TRT01AN
                   id=USUBJID,             ### The id for each subject; e.g. USUBJID
                   fmt,                    ### The wording for row labels of each event
                   cat,                    ### Category of the event: "TEAE" or "Death"
                   order){                 ### The order for sorting each event

  ae_n<-data%>%filter(eval(cond))%>%
    group_by(!!tx_var,!!id)%>%
    slice(1)%>%
    ungroup()%>%
    group_by(!!tx_var)%>%
    dplyr::summarize(sum1=n())%>%
    right_join(n_trt, by=join_by(!!tx_var))%>%
    mutate(across(everything(),~ifelse(is.na(.x),0,.x)))%>%
    mutate(prop=sprintf(sum1/n*100, fmt = '%.1f'),
```

Code chunk 1

```
           nprop=ifelse(sum1==0,"0",paste0(sum1," (",prop,")")),
           ROWLBL=fmt)
```

```
ae_tran<-ae_n%>%
   pivot_wider(names_from = !!tx_var,names_prefix = "TRT",
               values_from = c(nprop,n,sum1),id_cols = ROWLBL )%>%
   mutate(order1=order,ROWLBL1=cat)%>%
   rename_with(~substring(.,7),starts_with("nprop_"))

return(ae_tran)                                          Code chunk 2
}
```

R program 10 Function n_count1()

As part of the company standards, some events are required in the AE overall table, such as Serious AE, AE leading to death, whereas others, like study-specific AEs of special interest, are optional. We use the `n_count2()` function to determine how `n_count1()` will be applied under different scenarios.

In R program 11 code chunk 3, the function checks if the event-related variable count_var is missing from the arguments for required events. If it is missing, the function will stop and display an error. For optional events, if `count_var` is missing, the function returns a NULL value since it is acceptable. You may notice the use of the `substitute()` function, which can capture unevaluated arguments similar to `enquo()`. Moreover, it can also substitute the variable with its name.

In code chunk 4, we handle scenarios where there is only one variable in `count_var`. After checking the length of `count_var`, the function continues to check for its presence in the dataset. If missing, the function stops and throws an error; otherwise, it calls `n_count1()` and returns the results. In cases where multiple variables are input as a vector, it calls `n_count1()` for each variable in the vector and returns the combined results.

```
n_count2<-function(req="Y",            ### Whether this event is required by standard or
                                           not? "Y" as default
                   data,               ### Dataset used for counting
                   n_trt,              ### Dataset containing # of subjects by treatments
                   count_var=NULL,     ### The variable(s) used to identify event(s)
                   cond=NULL,          ### The condition for subsetting the dataset,
                                       ### should be related to count_var
                   tx_var,             ### The numeric variable contains treatment
                                       ### information; e.g. TRT01AN
                   id=USUBJID,         ### The id for each subject; e.g. USUBJID
                   fmt,                ### The wording for row labels of each event
                   cat,                ### Category of the event: TEAE or Death
                   order){             ### The order for sorting each event

if (is.null(substitute(count_var))){
  if (req=="Y"){
    stop("Error: The required variable count_var is missing from the function arguments")
  }else{
    return(NULL)                                        Code chunk 3
  }
}else{

    if (length(substitute(count_var))==1){
      if (!any(names(data)==substitute(count_var))){
        stop("Error: Variable ",substitute(count_var)," is not available in ",
             substitute(data))
      }
      counts_by_trt<-n_count1(data, n_trt, enexpr(cond),enquo(tx_var),
                              enquo(id),fmt, cat, order)      Code chunk 4
    return(counts_by_trt)
    }else{
      counts_by_trt=data.frame()
```

Code chunk 5

```
      for (i in 1:length(count_var)){
        if (!(any(names(data)==count_var[i]))){
          stop("Error: Variable ",count_var[i]," is not available in ", substitute(data))
        }
        counts<-n_count1(data, n_trt, expr(!!sym(count_var[i])=="Y"), enquo(tx_var),
                         enquo(id),fmt[i], cat, order)%>% mutate(order2=i)
        counts_by_trt<-bind_rows(counts_by_trt,counts)
      }
      return(counts_by_trt)
    }
  }
}
```

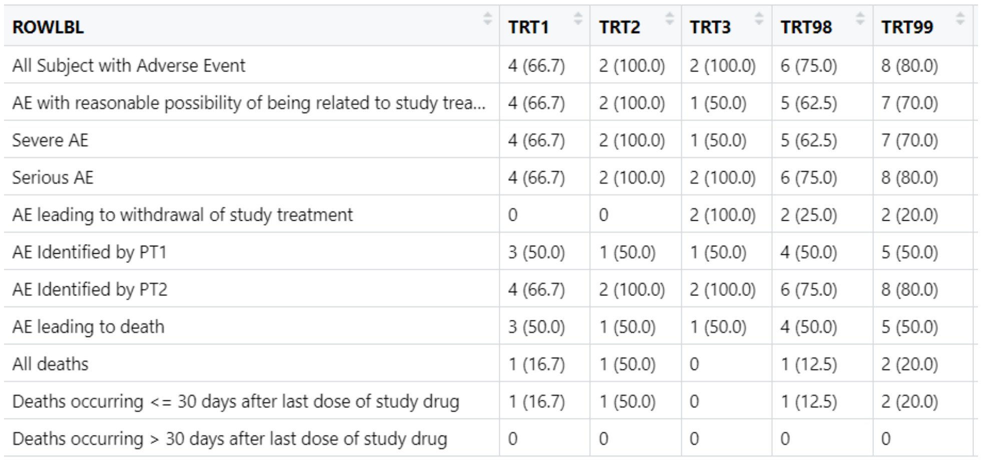
R program 11 Function n_count2()

1.3 Combine all data and output dataset for table generation

Finally, we can use R program 12 to combine all generated data. The `all_data` is a list containing all events that might be used for generating the AE overall table, then we use `bind_rows()` to combine them. By setting NULL for events not required in the study, `bind_rows()` effectively combines only the needed events in the study. It's worth noting that we use `bind_rows()` from the {dplyr} package, instead of `rbind()` from base R, because `bind_rows()` can handle combining rows with different numbers of columns. The final results of the dataset are displayed in Figure 5, herein we only display the row label (ROWLBL), treatments (TRT1-3), active total (TRT98) and total (TRT99). With the dataset, we should be able to generate a report by using packages such as {gt}, {reporter} or {pharmaRTF}.

```
all_data<-c("ae_all","ae_rel","ae_rel_var","ae_sev","ae_ser","ae_wov",
            "aew_var","aecat_var","ae_fatal","ae_death","dthse","dthlt","dthcat_var")
final<-bind_rows(mget(all_data))%>%
  arrange(order1,order2)
```

R program 12 Combine all datasets



ROWLBL	TRT1	TRT2	TRT3	TRT98	TRT99
All Subject with Adverse Event	4 (66.7)	2 (100.0)	2 (100.0)	6 (75.0)	8 (80.0)
AE with reasonable possibility of being related to study trea...	4 (66.7)	2 (100.0)	1 (50.0)	5 (62.5)	7 (70.0)
Severe AE	4 (66.7)	2 (100.0)	1 (50.0)	5 (62.5)	7 (70.0)
Serious AE	4 (66.7)	2 (100.0)	2 (100.0)	6 (75.0)	8 (80.0)
AE leading to withdrawal of study treatment	0	0	2 (100.0)	2 (25.0)	2 (20.0)
AE Identified by PT1	3 (50.0)	1 (50.0)	1 (50.0)	4 (50.0)	5 (50.0)
AE Identified by PT2	4 (66.7)	2 (100.0)	2 (100.0)	6 (75.0)	8 (80.0)
AE leading to death	3 (50.0)	1 (50.0)	1 (50.0)	4 (50.0)	5 (50.0)
All deaths	1 (16.7)	1 (50.0)	0	1 (12.5)	2 (20.0)
Deaths occurring <= 30 days after last dose of study drug	1 (16.7)	1 (50.0)	0	1 (12.5)	2 (20.0)
Deaths occurring > 30 days after last dose of study drug	0	0	0	0	0

Figure 5 AE Overall Table

1.4 Wrap it up!

For highly standardized tables, such as the AE overall table, it is possible to use a single comprehensive function that encapsulates all the necessary code and functions. While we won't delve into the specifics of constructing such an "all-inclusive" function, it follows a concept similar to the nested function structure we discussed earlier. I would like to emphasize the advantages of this approach:

1. User-friendly transition: This method is particularly beneficial for programmers transitioning from SAS to R. By simply adjusting a few arguments, they can easily generate tables without needing extensive R programming skills.

2. Automation: It facilitates the automation of table generation. By incorporating the functions within an R Shiny app (Ling & Wang, 2025), users can batch run multiple tables together with a single click. Developing automation tools based on R functions and R shiny would significantly enhance the work efficiency of programmers.

CONCLUSION

In this paper, the exploration of SAS MACROs versus R functions highlights the distinct strengths each programming language offers. By comparing structures, methods of invocation, parameter evaluation, variable scoping, and conditional logic, we shed light on the unique capabilities that define SAS MACROs and R functions. The case of using R functions in generating AE overall table underscored R's utility and adaptability, serving as a practical guide for automating complex reporting tasks. SAS macros and R functions both aim to enhance automation and code reuse. Despite differences in syntax, understanding these distinctions can facilitate a smooth transition from using SAS macros to R functions. Through this comparative analysis, this paper aimed to bridge the gap between SAS macros and R functions, equipping programmers with the insights and skills necessary to develop R functions from scratch. As the field continues to evolve, embracing the capabilities of both SAS and R will be essential for improving productivity and the analytical depth of research projects.

REFERENCES

Altman, S., Behrman, B., & Wickham, H. (2021). *Functional Programming.* Retrieved from https://dcl-prog.stanford.edu/tidy-eval-detailed.html

Buchecker, M. (2020). Think Globally, Act Locally: Understanding the Global and Local Macro Symbol Tables. *SAS GLOBAL FORUM 2020.* Virtual. Retrieved from https://support.sas.com/resources/papers/proceedings20/4160-2020.pdf

Joe. (2016, Feb 22). *Why aren't SAS Macro Variables Local-Scope by Default?* Retrieved from Stack Overflow: https://stackoverflow.com/questions/35533276/why-arent-sas-macro-variables-local-scope-by-default

Lee, K., & Jeeva, M. (2022). Enterprise-level Transition from SAS® to Open-Source Programming for the whole department. *PharmaSUG 2022 Conference Proceedings.* Austin, Texas.

Ling, C., & Wang, Y. (2025). TLFQC: A High-compatible R Shiny based Platform for Automated and Codeless TLFs Generation and Validation. *PharmaSUG 2025 conference proceedings.* San Diego.

Lyons, L. (2004). Going Under the Hood: How Does the Macro Processor Really Work? *Northeast SAS Users Group.* Baltimore, Maryland. Retrieved from https://www.lexjansen.com/nesug/nesug04/pm/pm07.pdf

Peng, R. D. (2022). *R Programming for Data Science.*

Slaughter, S. J., & Delwiche, L. D. (2003). SAS Macro Programming for Beginners. Montreal, Canada. Retrieved from https://support.sas.com/resources/papers/proceedings/proceedings/sugi29/243-29.pdf

Wang, Y., & Ling, C. (2025). Comparing SAS® and R Approaches in Reshaping data. *PharmaSUG 2025 Conference Proceedings.* San Diego, CA.

Wang, Y., & Ling, C. (2025). Controlling attributes of .xpt files generated by R. *PharmaSUG 2025 conference proceedings.* San Diego, CA.

Wickham, H. (2019). *Advanced R* (2nd ed.). Retrieved from https://adv-r.hadley.nz/index.html

ACKNOWLEDGMENTS

We would like to express our sincere gratitude to Rina Loke, Christelle Raynaud and the R team, Xiangdong Zhou, Holly Peterson and Ujjwala Powers for their support, trust, and encouragement all along.

CONTACT INFORMATION

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

Chen Ling
AbbVie Inc.
Chen.ling@abbvie.com

Yachen Wang
AbbVie Inc.
Yachen.wang@abbvie.com