PharmaSUG 2026 - Paper AP-228

Timing, Masking, and Resolution: Understanding and Debugging SAS Macros

Carleigh Jo Crabtree, SAS Institute

ABSTRACT

SAS macro processing can produce unexpected results. Having a clear understanding of how and when macro variables are resolved is essential to debugging. This paper examines three foundational concepts essential to writing robust SAS macros: timing, masking, and resolution. Beginning with an overview of the SAS tokenization process and the role of the macro processor, this paper demonstrates how the sequence of macro execution can lead to unintended variable substitution when special characters such as ampersands and percent signs appear within macro variable values. Masking functions %STR and %NRSTR are presented as tools to prevent premature resolution, with attention given to their distinctions and appropriate use cases. This paper further explores how standard macro functions can inadvertently remove character masking during execution, and introduces the "Q" family of macro functions — such as %QUPCASE — as a mechanism for preserving masking through function execution. Annotated code examples and process diagrams are provided throughout to illustrate each concept. Readers will come away with a practical framework for anticipating, diagnosing, and resolving common macro behavior issues in SAS programming.

Related programs and files can be found on my GitHub: PharmaSUG_2026/AP_228

UNDERSTANDING MACRO TIMING

When a SAS program is submitted, it's sent to an area of memory called the **input stack**. The input stack holds the program while the **word scanner** analyzes characters and breaks them into tokens. Tokens are the smallest pieces of meaningful information to SAS.

The four token types are:

- **Literal**: One or more characters enclosed in single or double quotation marks.
- **Number**: A numeric value including integers and real (floating-point) numbers.
- **Name**: One or more characters beginning with a letter or an underscore. Other characters can be letters, underscores, and digits.
- **Special**: Any character that is not a letter, number, or underscore.

A token ends when the word scanner encounters either a blank space or when a new token begins.

Program 1 would be tokenized as shown by the color coding:

```
data orders ;
     set orders_big ;
     where orderType = "Online Sale" and productID = 9304 ;
run ;
```

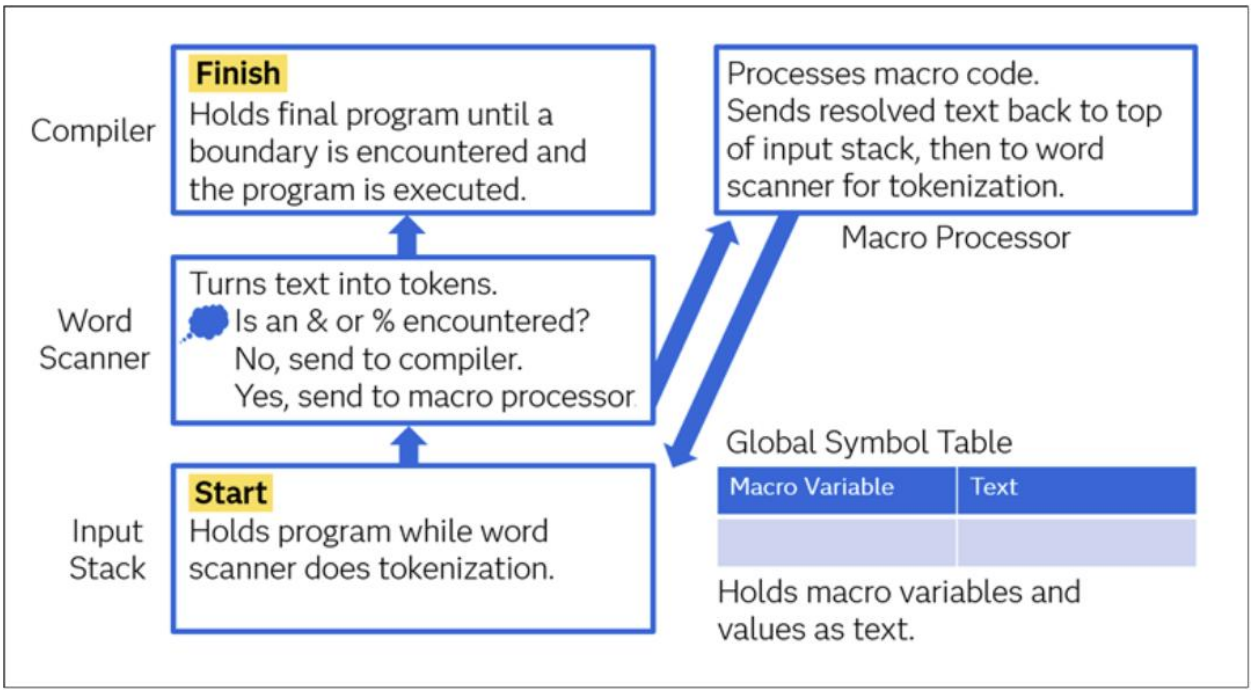
Program 1. Token types displayed by color

As the word scanner reads each token, it determines where to send it. Everything in the above program is sent from the input stack to the word scanner for tokenization. After tokenization, it's sent to the compiler and executed.

The ampersand (&) and percent sign (%) are **macro triggers**. When the word scanner encounters a macro trigger followed by an underscore or character, it sends the information to the **macro processor** instead of the compiler. The macro processor interprets the macro language and sends the result back to the top of the input stack, where it is re-tokenized by the word scanner.

SAS also creates a **global symbol table** at the beginning of each session to store automatic and global macro variable values. When a macro definition is called, a **local symbol table** is also created.

Display 1 shows the processing flow when a program is submitted:



Display 1. Macro Process

EXAMPLE 1: EXPECTED MACRO BEHAVIOR

```
%let claim=endurance;

title "Supplements Claimed to Improve: &claim";
proc print data=pharm.sports_supplements;
    where  Claimed_Improved_Fitness_Aspect="&claim";
run;
```

Program 2. Expected macro behavior

In Program 2:

1. The program is held in the input stack while the word scanner processes one token at a time.
2. The % macro trigger activates the macro processor.
3. The macro variable **claim** is created with the value *endurance* in the global symbol table.
4. Tokenization continues until the & macro trigger is found in the TITLE statement.
5. The macro processor resolves **&claim** with the value found in the global symbol table.
6. The resolved text is sent back to the input stack, tokenized again, and sent to the compiler.
7. The same thing happens when the & is encountered in the WHERE statement.
8. When the RUN statement is encountered, it's a boundary that causes SAS to execute the code.

Supplements Claimed to Improve: endurance

Obs	Supplement	Alt_Name	Evidence_Level	Claimed_Improved_Fitness_Aspect	Exercise_Tested	Popularity	Studies_Examined	Total_Citations	Notes	N_Positive_Studies	Percent_Positive_Studies	Study_Source
1	CLA	Conjugated linoleic acid	0	endurance	other	3620	1	374		0	0	Colakoglu et al (2006)
2	Fish oil	omega 3 PUFAs	0	endurance	football, cycling	10800	3	3090		0	0	Buckley et al (2009)
3	Glycerol		1	endurance	other	851	1	99	May improve performance in endurance sports by enhancing water retention, but more research is needed.	-	-	Review: Goulet et al (2007)
4	Green tea extract	Catechins	0	endurance	cycling	2990	2	38		0	0	Eichenberger et al (2010)
5	HCA	Garcinia cambogi, (-)-Hydroxycitrate	1	endurance	cycling	675	2	54	May fire up fat metabolism, sparing carbohydrate fuel and allowing you to exercise for longer before getting exhausted.	2	1	Lim et al (2003)
6	L-carnitine LT	L-carnitine L-tartrate	0	endurance	cycling	15600	1	1300		0	0	Broad et al (2005)
7	Medium chain triglycerides		0	endurance	running	801	1	22		0	0	Oopik et al (2001)
8	Ribose	D-ribose	0	endurance	rowing	1490	1	201		0	0	Dunne et al (2006)

Results 1. Results from Macro Program 1

EXAMPLE 2: WARNING- APPARENT SYMBOLIC REFERENCE NOT RESOLVED

In the Program 3, the macro variable value contains an &, which is a macro trigger. When the program is run, three warnings are generated in the log- but the correct results are returned. Why?

```
%let claim=inflammation&muscle soreness;

title "Supplements Claimed to Improve: &claim";
proc print data=pharm.sports_supplements;
     where  Claimed_Improved_Fitness_Aspect="&claim";
run;
```

Program 3. Example 2 Code

```
1          OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
WARNING: Apparent symbolic reference MUSCLE not resolved.
72
73         /* Works, but produces warnings in the log because of &muscle. */
74         %let claim=inflammation&muscle soreness;
75
76         title "Supplements Claimed to Improve: &claim";
WARNING: Apparent symbolic reference MUSCLE not resolved.
77         proc print data=pharm.sports_supplements;
78         where  Claimed_Improved_Fitness_Aspect="&claim";
WARNING: Apparent symbolic reference MUSCLE not resolved.
79         run;
```

Log 1. Log from Macro Program 2

Supplements Claimed to Improve: inflammation&muscle soreness

Obs	Supplement	Alt_Name	Evidence_Level	Claimed_Improved_Fitness_Aspect	Exercise_Tested	Popularity	Studies_Examined	Total_Citations	Notes	N_Positive_Studies	Percent_Positive_Studies	Study_Source
1	Fish oil	omega 3 PUFAs	2	inflammation&muscle soreness	weight training, cycling, swimming	10800	6	3090	Although commonly believed to lower inflammation, evidence for fish oil's anti-inflammatory effect during or after exercise is weak.	2	0.3333	Andrade et al (2007)

Results 2. Results from Macro Program 2

The %LET statement includes **&muscle**, so the macro processor tries to resolve a macro variable named **muscle**.

Because no such macro variable exists, SAS issues a warning and includes the & as regular text in the macro variable's stored value.

In the TITLE and WHERE statements, the same thing happens: **&muscle** is scanned, not found as a macro variable, and left as is.

The warnings are harmless because there's no macro variable named **muscle**, so nothing is substituted. But what if there *is* one?

EXAMPLE 3: UNWANTED RESOLUTION

```
%let muscle=UnwantedText;
%let claim=inflammation&muscle soreness;

title "Supplements Claimed to Improve: &claim";
proc print data=pharm.sports_supplements;
     where  Claimed_Improved_Fitness_Aspect="&claim";
run;
```

Program 4. Example 3 Code

In Program 4, a macro variable named **muscle** exists. Now the macro processor encounters &muscle, finds the value *UnwantedText* in the global symbol table, and substitutes it. The resulting value of **claim** becomes: *inflammationUnwantedText soreness*.

The TITLE and WHERE statements are now incorrect:

```
title "Orders placed for: inflammationUnwantedText soreness";

where product_name="inflammationUnwantedText soreness";
```

Program 5. Example 4 Updated Statements

This is not a valid value in the data, therefore no rows are returned, and no report is generated.

```
SYMBOLGEN:  Macro variable MUSCLE resolves to UnwantedText
75
SYMBOLGEN:  Macro variable CLAIM resolves to inflammationUnwantedText soreness
76         title "Supplements Claimed to Improve: &claim";
77         proc print data=pharm.sports_supplements;
78         where  Claimed_Improved_Fitness_Aspect="&claim";
SYMBOLGEN:  Macro variable CLAIM resolves to inflammationUnwantedText soreness
```

Log 2. Log from Macro Program 3

MASKING SPECIAL CHARACTERS

To protect your macro values from unintended substitutions, use the %NRSTR quoting function. %NRSTR masks special characters, including & and %, and mnemonic operators in constant text during macro compilation. This prevents the macro processor from resolving macro variables during compilation.

A similar function to %NRSTR is %STR. Both are used to mask special characters and mnemonic operators in constant text at macro compilation, however, %STR does *not* mask % and &.

The special characters and mnemonics they both mask are:

```
+ - * / < > = ¬ ^ ~ ; , # blank

AND OR NOT EQ NE LE LT GE GT IN
```

Figure 1. Masked Characters

They also mask the following characters when they occur in pairs. If only one of these characters appears in the text, add a preceding % to mask it.

```
' " ( )
```

Figure 2. Masked Pairwise Characters

EXAMPLE 4: %NRSTR

```
%let muscle=UnwantedText;
%let claim=%nrstr(inflammation&muscle soreness);

title "Supplements Claimed to Improve: &claim";
proc print data=pharm.sports_supplements;
     where  Claimed_Improved_Fitness_Aspect="&claim";
run;
```

Program 6. Example 4 Code

```
SYMBOLGEN:  Macro variable CLAIM resolves to inflammation&muscle soreness
SYMBOLGEN:  Some characters in the above value which were subject to macro quoting have been
            unquoted for printing.
77         title "Supplements Claimed to Improve: &claim";
78         proc print data=pharm.sports_supplements;
79         where  Claimed_Improved_Fitness_Aspect="&claim";
SYMBOLGEN:  Macro variable CLAIM resolves to inflammation&muscle soreness
SYMBOLGEN:  Some characters in the above value which were subject to macro quoting have been
            unquoted for printing.
```

Log 3. Log from Macro Program 4

Supplements Claimed to Improve: inflammation&muscle soreness

Obs	Supplement	Alt_Name	Evidence_Level	Claimed_Improved_Fitness_Aspect	Exercise_Tested	Popularity	Studies_Examined	Total_Citations	Notes	N_Positive_Studies	Percent_Positive_Studies	Study_Source
1	Fish oil	omega 3 PUFAs	2	inflammation&muscle soreness	weight training, cycling, swimming	10800	6	3090	Although commonly believed to lower inflammation, evidence for fish oil's anti-inflammatory effect during or after exercise is weak.	2	0.3333	Andrade et al (2007)

Results 3. Results from Macro Program 4

Adding %NRSTR around the value for **&claim** causes the macro processor to treat the & as regular text. Even though a macro variable named **muscle** exists, %NRSTR masks the & so it's treated as text, not as a macro trigger. You get the expected result, **no warnings**, and **accurate output**.

EXAMPLE 5: %STR AND PAIRWISE CHARACTERS

The %LET would cause unexpected log notes due to the unclosed quotation mark:

```
%let altName= devil's thorn, tackweed;
```

Program 7. Example 5 Code

To mask it, use %STR with a % in front of the quotation mark:

```
%let altName= %str(devil%'s thorn, tackweed);
%put &=altName;
```

Program 8. Example 5 Code

```
SYMBOLGEN:  Macro variable ALTNAME resolves to devil's thorn, tackweed
SYMBOLGEN:  Some characters in the above value which were subject to macro quoting have been
            unquoted for printing.
ALTNAME=devil's thorn, tackweed
```

Log 4. Log from %LET and Masking Pairwise Characters

CONTROLLING RESOLUTION

Suppose we want to store the value of **&claim** in all capital letters. We create a new macro variable named **&uClaim** and use the %UPCASE macro function.

```
%let muscle=UnwantedText;
%let claim=%nrstr(inflammation&muscle soreness);
%let uClaim=%upcase(&claim);

%put &=uClaim;
```

Program 9. Controlling Resolution Code

```
SYMBOLGEN:  Macro variable CLAIM resolves to inflammation&muscle soreness
SYMBOLGEN:  Some characters in the above value which were subject to macro quoting have been
            unquoted for printing.
SYMBOLGEN:  Macro variable MUSCLE resolves to UnwantedText
77
78         %put &=uClaim;
SYMBOLGEN:  Macro variable UCLAIM resolves to INFLAMMATIONUnwantedText SORENESS
UCLAIM=INFLAMMATIONUnwantedText SORENESS
```

Log 5. Log from Program 9

Despite using %NRSTR when creating **&claim**, **&muscle** is still resolved. Why does this happen?

VISUALIZING MACRO TIMING AND RESOLUTION

The answer comes down to timing. In comparison to normal SAS function, macro functions manipulate *text*, not data values. They are executed by the *macro processor* before the SAS code is compiled and executed.

While **&claim** is stored in the global symbol table with masked special characters, those masks do not automatically survive the use of standard macro functions.

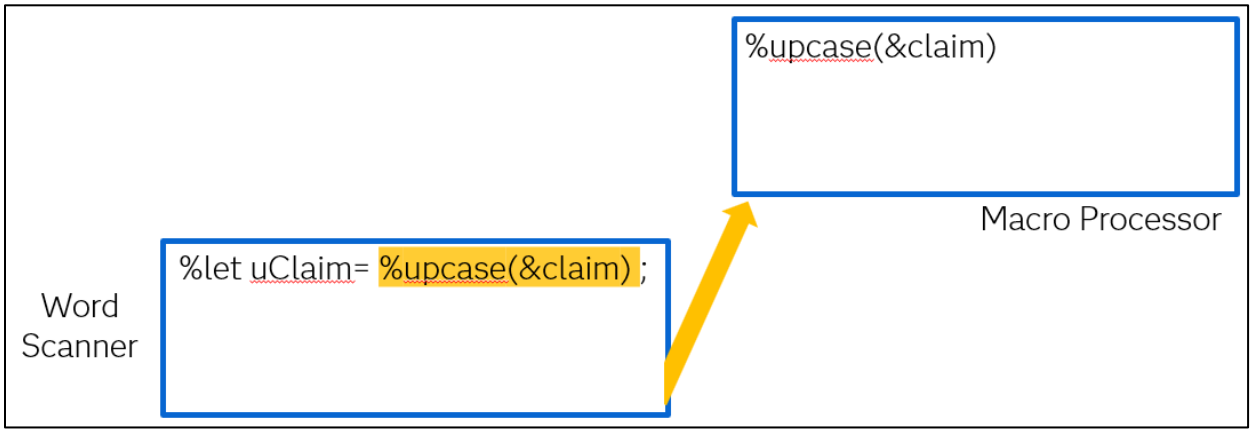
The following displays represent a simplified visual of what happens behind the scenes.

Global Symbol Table

Macro Variable	Text
muscle	UnwantedText
claim	inflammation&muscle soreness

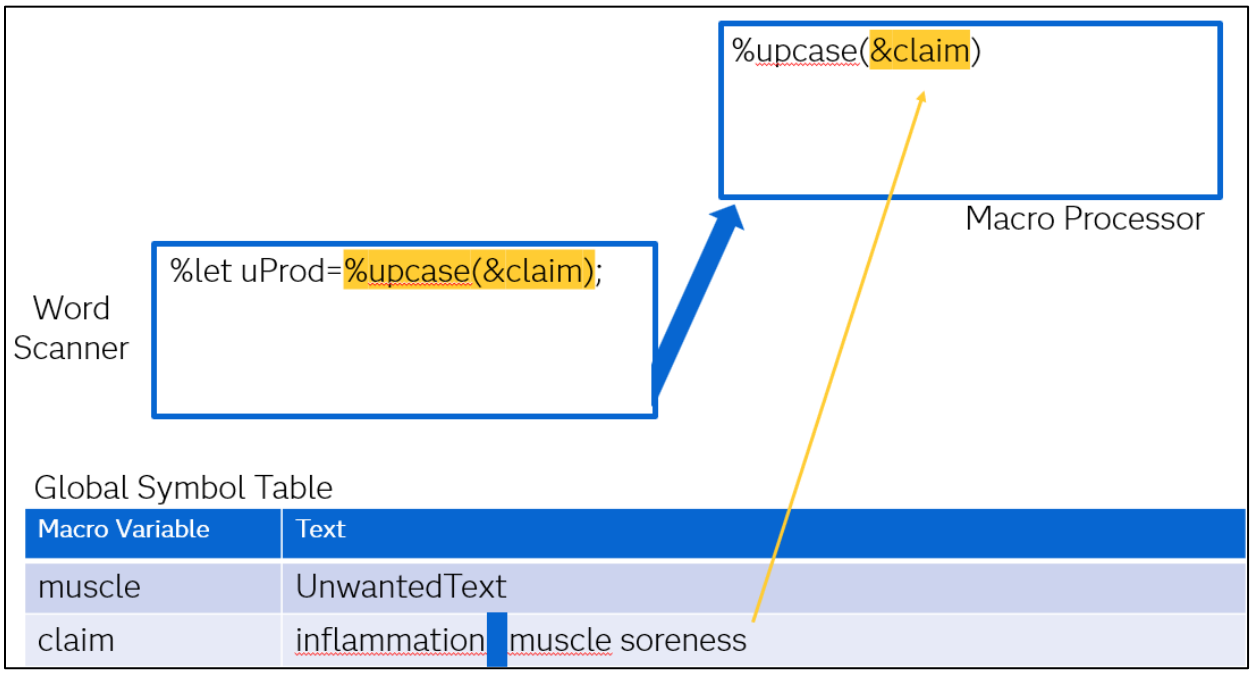
Display 2. Global Symbol Table

When the %LET statement containing %UPCASE is encountered, it is sent to the macro processor.

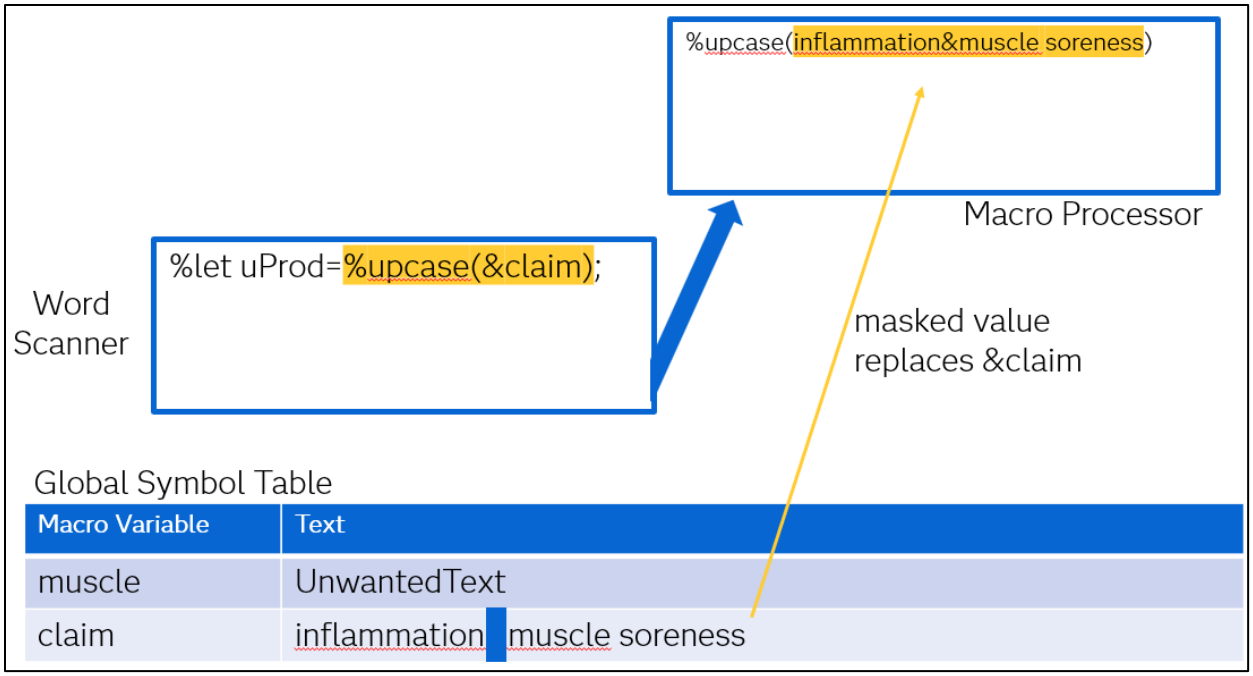


Display 3. Macro Resolution

The masked value of **&claim** is resolved in the macro processor.

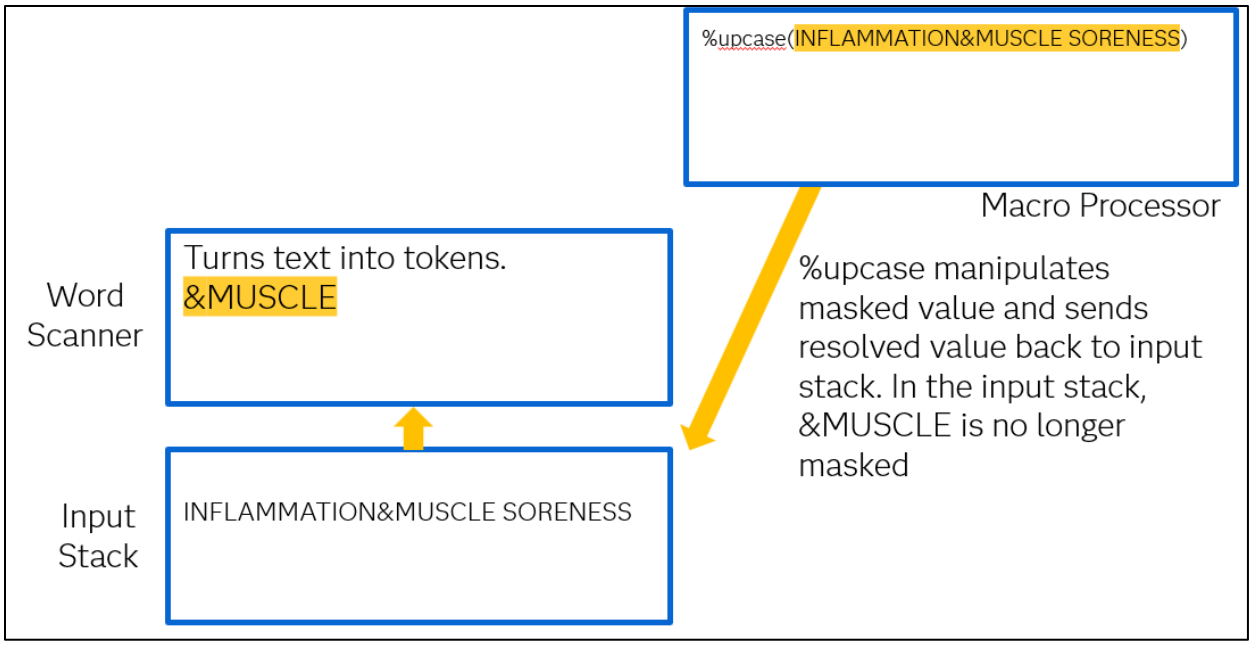


Display 4. Macro Resolution



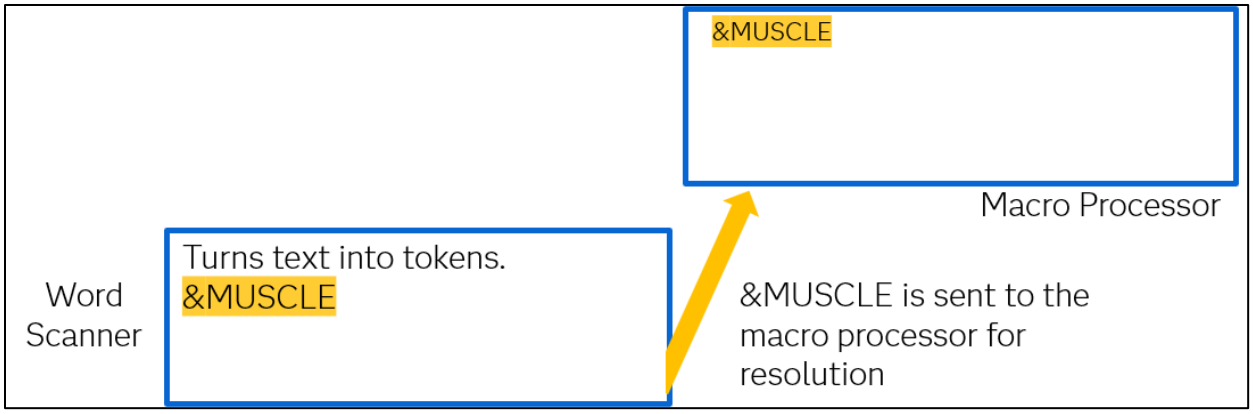
Display 5. Macro Resolution

Then, the macro processor executes %UPCASE. The masked value is manipulated, and the resolved value is sent back to the input stack. In the input stack, the resolved value is no longer masked.



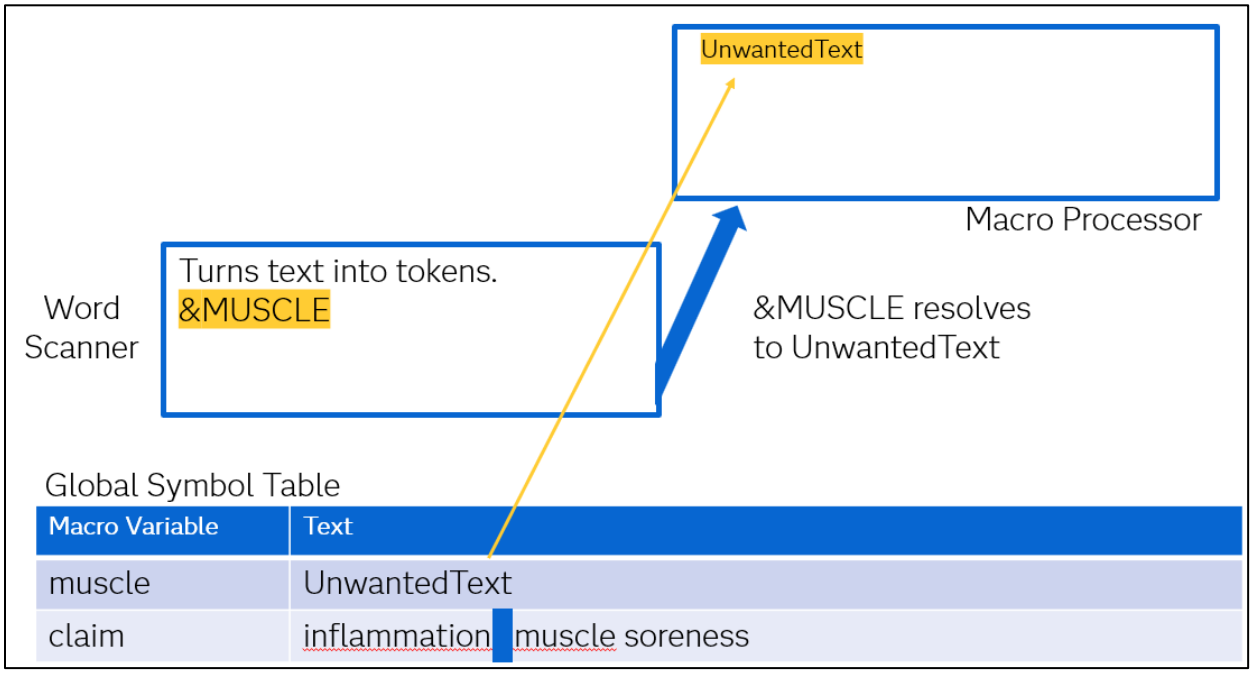
Display 6. Macro Resolution

As tokenization continues, the word scanner encounters *&MUSCLE* and sends it back to the macro processor for resolution.



Display 7. Macro Resolution

At that point **&MUSCLE** resolves to *UnwantedText* in the macro processor.



Display 7. Macro Resolution

The final value of **&uClaim** is stored in the global symbol table unmasked, with undesired results.

Global Symbol Table

Macro Variable	Text
muscle	UnwantedText
claim	inflammation muscle soreness
uClaim	INFLAMMATIONUnwantedText SORENESS

Display 8. Macro Resolution

EXAMPLE 6: MACRO Q FUNCTIONS

To remedy this, we can use the “Q” equivalent of the %UPCASE function: %QUPCASE. Macro Q functions mask the returned result after the function executes in the macro processor.

```
%let muscle=UnwantedText;
%let claim=%nrstr(inflammation&muscle soreness);
%let QuClaim=%qupcase(&claim);

%put &=QuClaim;
```

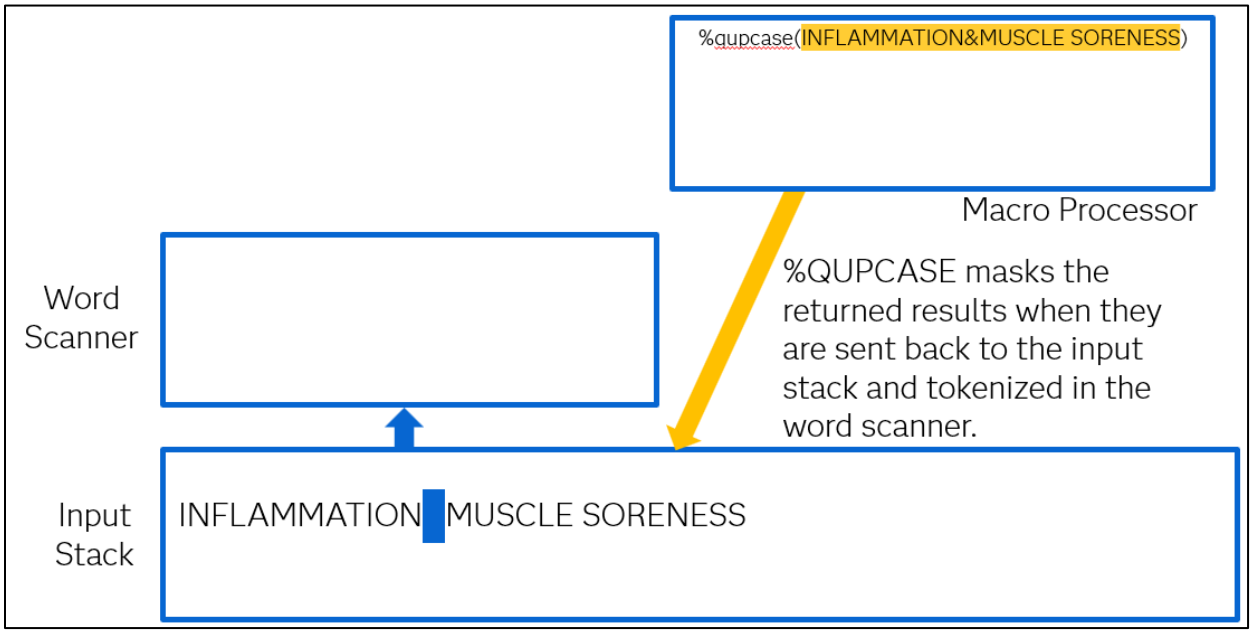
Program 10. Example 6 Code

```
77          %put &=QuClaim;
QUCLAIM=INFLAMMATION&MUSCLE SORENESS
```

Log 6. Log from Program 10

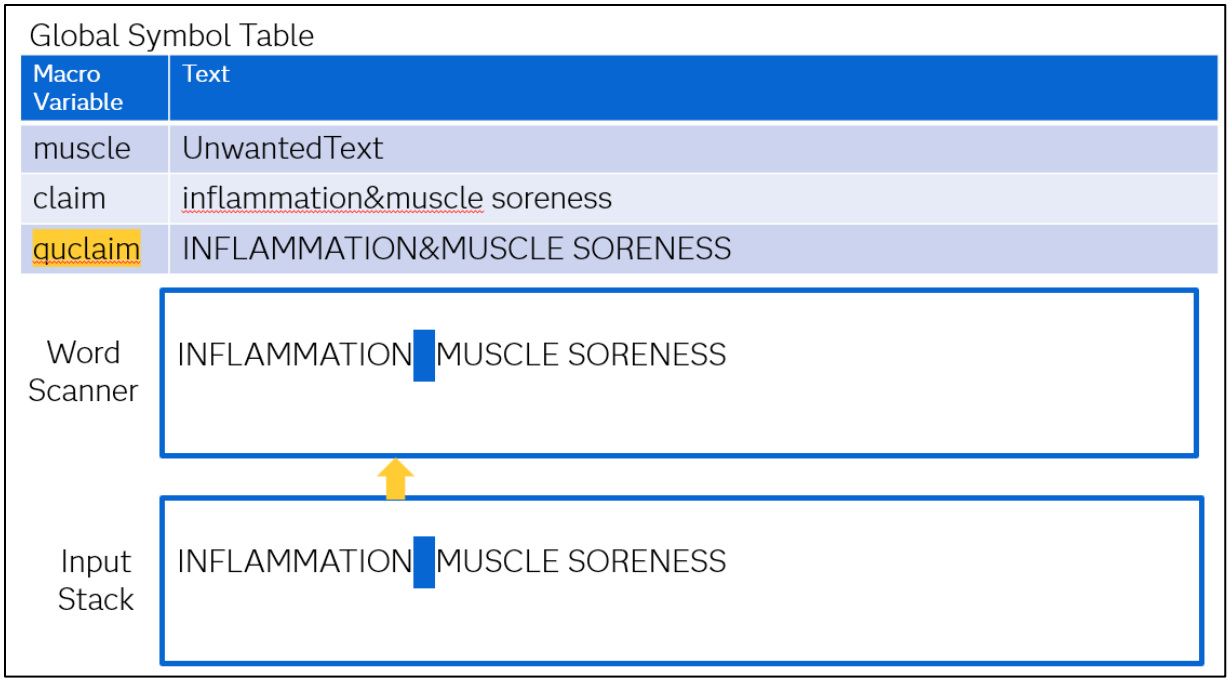
Why does this work? Let’s look at the visual again.

When %QUPCASE is sent to the macro processor, the masked value of **&claim** is resolved just as before. However, because a Q function is used, the returned value is re-masked before it is sent back to the input stack. As tokenization continues, special characters remain masked and &MUSCLE is seen as just text.



Display 9. Macro Q Functions

The masked value of **&QuClaim** is then stored in the global symbol table as desired.



Display 10. Macro Q Functions

Q functions are essential when you need to apply macro functions to values that contain masked special characters. They allow you to manipulate text while preserving the masking that prevents unintended macro resolution.

There are several other Q equivalent macro functions available. For a complete list, see the SAS Help Center documentation on macro functions.

CONCLUSION

Understanding how SAS processes macro code — from tokenization through resolution — is essential for writing reliable and predictable macros. When macro variable values contain special characters, unintended substitutions can occur silently, producing incorrect results without obvious errors in the log. By applying masking functions and leveraging Q function equivalents, programmers can maintain control over when and how macro variables are resolved. With a clear mental model of macro timing and the tools to manage special characters appropriately, SAS programmers are better equipped to write cleaner code, interpret unexpected behavior, and debug macro issues with confidence.