Paper 029-2010

A Serious Look Macro Quoting

Ian Whitlock, Kennett Square, PA

Abstract

You can make decisions macro with %IF and do looping with %DO-loops. But there are times when you don't understand why the beast does what it does. Now what? It is time to come to this presentation.

It is time to take a serious look at macro quoting. I have often said that anyone who thinks macro quoting is simple, probably doesn't understand the problem; so I have been there. Now I want to explain how simple it is.

Everything relevant to this paper is in BASE SAS®. Although the examples have been executed on a PC under Windows, the examples are independent of any particular operating system with the exception of file definitions.

You might think this paper is a compendium of all the macro quoting functions. It is not. It is more about the subject of macro quoting than about the macro quoting functions.

Introduction

Quoting in macro is really a difficult subject so, let's begin on safer ground with a familiar context, SAS. How and why does quoting arise? There has to be some way to distinguish the instruction parts of the language from the values acted on by those instructions. Consider the assignment:

```
x = abc ;
```

The question is, what is on the right, a variable or a value. The SAS rule is that if it begins with a letter or underscore and continues with letters, underscore, or digits it must be a variable. Hence values must be distinguished. Quoting comes to the rescue. Quoting is used for character values, but not numeric ones. Why is it, that the values are distinguished and not the variable names? Well programs typically refer to many more variables than values; so it is more efficient to quote the values than distinguish the variables. Moreover it is traditional with many programming languages for the same reasons.

Now consider:

```
x = 123 ;
```

Here the number, 123, cannot be taken for a variable name so there is no need to quote. In fact, it would be wrong because then we could not distinguish the number from the character string, "123".

To repeat: Why are character values quoted? They are quoted so that they can be distinguished from other elements of the SAS language.

The same problem occurs in ordinary conversation. Is Boston a six letter word, or is it a city? Again one distinguishes the literal value by quoting it, i.e. putting it between quote marks.

SAS macro is a programming language for manipulating text that is to become a SAS program, or part of a SAS program. So the first order of business is to decide how to tell the instructions from the SAS code. Macro uses a different technique from SAS. The %-symbol is used to begin macro instructions and the &-symbol to reference variables, so the instructions are distinguished rather than the values acted upon. Everything not required as part of an instruction is data, i.e. bits of SAS code. This decision was important and a good one. Either the values or the instructions must be distinguished in some form. Since the values are bits of SAS code, it would be most awkward if SAS code were required to be quoted: 1) Because we are not used to it, and 2) Because the SAS code in a typical macro can be quite extensive.

Also notice that we cannot distinguish the macro instruction objects with quote marks since they already have a meaning in SAS. Macro instructions are distinguished by beginning with a %-symbol and macro variable references are distinguished with the &-symbol. This means that most of what is left will be constant SAS code. The exception is that once an instruction is begun certain objects will be required in certain places; hence they may be considered part of the instruction without further need to distinguish.

The other consequence from this is that all true macro quoting must be done with macro functions because quote marks are not available for this purpose.

We should look at an example in detail. Consider the macro instruction assignment:

```
%let x = ABC ;
```

The %LET indicates an assignment instruction. The X is a macro variable name. Why doesn't it need to be distinguished from an X in SAS? %LET requires the macro variable name to follow the %LET; hence there is no need to distinguish. The X cannot be a SAS object. Moreover, it would be most inconvenient to distinguish the X because %X would reference a macro X whose job would be to create the macro variable name, and &X would be a macro variable reference whose value is to be the name of the assigned variable. Thus a new symbol or function would become necessary to make the distinction. Fortunately the macro language follows the rule: Don't distinguish when it isn't needed. In this case quoting would be quite wrong and the macro facility would not understand the instruction if we quoted the macro variable name. What about the equal symbol? Same principle; it is an expected part of the instruction and hence it would be wrong to quote it.

Now what about the ABC? Well, the rule is that this has to be a value because everything after the equal sign up to the semi-colon except for leading and trailing blanks is part of the value. So once again there is no need to quote it, although in this case the language permits it. Please note that quoting here means the use of a macro quoting function, not the simple use of quote marks. Why? Because the quote marks would be part of the value. They would not indicate quoting. In macro quoting must be achieved through the use of functions or new symbols. Wisely the SAS Institute chose not to introduce new symbols for this purpose. So in macro when we talk about quoting, we mean the use of quoting functions.

To repeat: the value ABC could be enclosed in a quoting function, but there is no need to do so and it is a good idea to avoid quoting when there is no need for it. The use of quoting functions will make the reading of macro code very much harder; hence avoiding unnecessary quoting should be one of the chief reasons to come to an understanding of when and why quoting is needed.

Finally we come to the semi-colon in the %LET instruction. It ends the %LET instruction, but semi-colons also end SAS instructions, so there is a conflict - who owns the semi-colon? Macro, as the active language, owns any semi-colon that could end a macro instruction. However, all other semi-colons are just data and ready to be part of the SAS code generated, i.e. they are not part of a macro instruction, they are passed on to the SAS compiler. If the SAS compiler is looking for the end of an instruction then the semi-colon provides the end. Otherwise the semi-colon is simply a null statement in SAS. Of course unexpected null statements in either SAS or macro can cause all sorts of difficulties and therefore should be avoided. This is the reason why it is important to not place semi-colons after macro invocations.

Now we have a problem. What if we want the semi-colon to be part of the value of a macro variable? Somehow we must tell macro that this is not the kind of semi-colon that ends a %LET statement. Thus quoting has become necessary in spite of the decision to distinguish macro instructions with % and & symbols from the data, SAS code. Now how did that happen? We allowed the semi-colon to be part of both the macro language and the SAS language; hence it will sometimes be necessary to distinguish the semi-colon as data, i.e. part of SAS code, from the semi-colon as the end of a macro instruction. To repeat: Any time a collection of symbols can have a meaning both in SAS and macro, it will become necessary to distinguish which language has control, whenever the context within a macro instruction cannot do so. Consequently quoting will be needed for this purpose. The macro quoting functions are used to make these distinctions, not quote marks.

The advantage of using the semi-colon to end macro instructions is that we are used to using semi-colons for ending instructions. Hence the syntax of the language is easier for the beginner to grasp. However, it comes at the expense of opening the door to requiring a complex system of quoting. Remember that was precisely the problem presented above in the discussion of SAS assignments. There ABC was allowed to be a variable name and a character string so quoting became necessary to distinguish which was meant. In the case of SAS it is hard to imagine how the problem could have been avoided. In the case of macro the decision was made to reuse standard SAS symbols such as the semi-colon; i.e. making the programs look familiar, rather than trying to invent a completely new set of symbols.

Before turning to the details of the macro quoting functions it is a good idea to step back and consider the quoting problem further. In ordinary English, a sentence is just a sentence, so something is either quoted or not, and it stays that way in the sentence. The macro language is different because there is a time when the macro facility compiles

the instruction, a time when the SAS code is generated from the instruction, a time when SAS receives and compiles the generated code, and a time when SAS executes the executable image of the code. Thus the same collection of symbols may need to change their quoted status depending on at what time we consider the instruction. Consider the SAS instruction:

```
put "abc" ;
```

What gets written? Well just the letters. What happened to the quote marks? They are used to indicate the value, but they are not part of the value. So quoting is removed when the value is used. Usually the only time one has to worry about the situation in SAS is when a character string is passed to an operating system. In that case the operating system may also require quoting, i.e., the operating system does not expect a raw value but rather a quoted one. Since most of SAS programming does not involve passing strings to an operating system, the problem is not ubiquitous and often goes unnoticed.

Now consider the situation in macro. The semi-colon quoted to macro, must appear unquoted to SAS. As a general rule any macro quoting must be removed as the values are sent to SAS. Usually this activity is performed automatically, just as SAS does when processing the above PUT statement.

Did we open the door to quoting anywhere else? Yes. What about the =-symbol in macro instruction:

```
%let x = abc ;
```

An =-symbol already has meaning in SAS, so we might expect situations where it will be necessary to hide the equal sign from the macro facility through quoting. The elements AND, OR, NOT, +, -, LE, NE, EQ, parentheses, space, comma and period, in addition to others all have a meaning in SAS and in macro instructions, so they are all ready to cause you much grief. In version 9.1 the word IN will be added to this list so it is still growing as new features are added to the macro language.

I have often wondered whether we might have been better off with the symbols such as the ones in the parenthetical list, (%; %+ %- %& %and %not %, %(%) etc.) The fact that many people write fine macros with little or no macro quoting, suggests that the correct decision was made.

To sum up, it is the fact that same word or combination of symbols can have meanings in both SAS and macro that is the root cause for requiring quoting. A computer language must provide some means for distinguishing values from instructions. If any symbols or collection of characters are meaningful as part of an instruction and as part of a value, then the problem of quoting will arise.

The same problem occurs in mathematics, when one wants to talk about a formal system, there must be a way to distinguish the language of the system talked about from the language in which it is discussed. These distinctions are not natural for the human mind and require considerable effort to understand. Hence one can predict hard problems wherever quoting becomes a serious subject. Of course, the same problem occurs in ordinary language. We must be able to distinguish what we say from the subject we talk about. Remember the statement: "Boston" is a six letter word and Boston is a city with many people living in it.

SAS, like English, uses either single or double quotes to do the job of quoting. It would be impossible for macro to choose the same method since the language distinctions must be made. In macro quoting is achieved through macro functions.

The problem gets still more difficult because it is important to recognize two distinct times - macro compile time, when the instruction is compiled; and macro execution time, when the instruction is carried out, i.e. the SAS code is generated. So a time element must also be introduced. When does the quoting take place? For example, consider

```
%let x = &macvar ;
```

Suppose the value of the variable MACVAR, which is &MACVAR, is a semi-colon. Now there is no need to hide the semi-colon from the macro facility at compile time because the macro compiler does not see the value, &MACVAR. It sees the word, &MACVAR, not the semi-colon. During execution there might be a need to hide this value, i.e. when generating the SAS code. This means that the subject of quoting in macro must be inherently harder than the subject in SAS or in any computer language that is not dealing with another computer language.

The problems of quoting for SAS macro are closer to the corresponding difficulties in meta-mathematics. The common experience of most programmers does not usually require the intertwining of two languages. Consequently, it takes an effort to do any macro coding and a greater effort to develop the intuition necessary to handle macro quoting. This is what makes the subject hard.

At this point it would be well to recognize that quoting is about the concept of hiding the meaning of various symbols from various parties at various times. Consequently it will often be easier to think in terms of hiding rather than quoting. For example: What do you want to hide? From what part of the system do you want to hide it? When must it be hidden and when should it be revealed? We will often use the language of hiding rather than that of quoting, but the subject is the same. If you keep the three basic questions in mind and think in terms of hiding, then the subject becomes easier. Why? Because the human mind has developed over millions of years in which hiding was important, i.e. it is natural to your thought process in a way that quoting is not.

The %STR quoting function

The most basic quoting function is %STR. It does the job of hiding symbols at macro compile time. In other words, if you want to express a value that could be interpreted by the macro compiler as part of an instruction, then it must be hidden. Usually the %STR function is the best one for the job. By far the most important example is the semi-colon, with the space symbol close behind in second place. This means that one can write many useful macros no more macro quoting knowledge than how to hide semi-colons and spaces with %STR. The manual gives a complete list, but it easier to think of the problem. %STR will hide any symbols that are meaningful to the macro compiler with some important exceptions.

Now how long will the meaning remain hidden? The hiding done by %STR will continue through macro execution time as long as the value is not modified. Consider:

```
%let x = %str(;) ;

%put abc &x def ;
```

Here there is no problem. We get the message:

```
abc ; def
```

on the log. During the execution of the %PUT instruction the semi-colon in the value of X will remain hidden. Thus it will not cause the instruction to be prematurely ended.

However the situation changes when macro functions act on a value. Consider:

```
%put %upcase(abc &x def) ;
```

When executed we get:

```
ABC
NOTE: Line generated by the macro function "UPCASE".
1     ABC ; DEF
            ---
            180

ERROR 180-322: Statement is not valid or it is used out of proper order.
```

Clearly the semi-colon terminated the %PUT instruction. What happened? %UPCASE removed the quoting from its argument in changing the letters to upper case.

However, we shouldn't jump to conclusions too fast. The program:

```
%macro semicolon ;
   %local x ;
   %let x = %str(;) ;
   %put abc %upcase(&x) def ;
%mend  semicolon ;

%semicolon
```

will execute without a problem. What happened?

The first %PUT was in open code. As it is being compiled the %UPCASE function is applied and produces a semi-colon. So the %PUT is finished.

Now consider the %PUT in the macro. Here the macro compiler does not evaluate the expression, %UPCASE(&X). In fact it cannot be evaluated because the value of X does not yet exist, since the %LET statement has not executed. The evaluation must be done at execution time. Remember the code between the %MACRO statement and the %MEND statement is compiled, not executed.

Common confusion on this point comes from the situation where one macro calls another. Which has to be compiled first? It does not matter. When the calling macro is compiled there is no guarantee that the called macro will already have been compiled. The only requirement is that it be compiled by the time the first macro executes the call to the second macro.

The physical semi-colon at the end of the %PUT line is seen by the macro compiler as ending the %PUT statement. Now consider the execution of this compiled statement. The %UPCASE function does produce a raw semi-colon, but what is to be done with it? It does not end the compiled %PUT statement, and it does not get passed to SAS. This is a good example of how symbols gain macro meaning only through their context. The semi-colon under question has no meaning and is simply written to the log; hence there is no error. In short, that semi-colon did not need to be hidden in the compiled %PUT statement, and it did in the open code %PUT statement because evaluation of the open code %PUT argument took place while looking for the end of the %PUT statement.

Note that this means, in general, that any macro statement capable of both compilation and direct execution should really be treated as two different types of statements. The problem with the %PUT is not due to some mistake in developing the language. It is inherent in the logic of the situation. The existence of any macro variable values or macros at the time of compiling a given macro is a mere coincidence. That fact is central to understanding how to use the macro language.

Now we see that a careful understanding of the macro processing of a macro instruction will require knowing whether or not that macro instruction is to be compiled first. Moreover, we see that it can violently affect whether quoting is needed or not. Macro quoting is not only hard because it is a hard subject; it is also hard because one must understand subtle differences in the macro processing involved.

Now back to the problem, remember we were considering how long the quoting action of %STR lasts. In general, macro functions remove quoting when they are executed, otherwise the value stays quoted until it is passed to SAS. The exceptions are functions which begin with the letter "Q". For example,

```
%let x = %str(;) ;

%put %qupcase(abc &x def) ;
```

writes

```
ABC ; DEF
```

without any problem in either open code or from a macro. Many of the non-quoting macro functions now come with a corresponding Q-version.

In general, the symbols that can be hidden by %STR do not matter because they only gain meaning in macro through specific contexts. Thus quoting is often not necessary. So when is quoting needed?

The %EVAL does arithmetic and consequently logical evaluation. Remember that the expression, 1 + 1, is just an expression of three characters. To do the arithmetic indicated, we need the expression %EVAL(1 + 1). But the whole language of arithmetic expressions and logical evaluation is shared by both SAS and macro; hence there are lots of little things to cause trouble whenever %EVAL is used. Note that all of the expressions inside the following calls to %EVAL also have a similar SAS meaning.

```
%eval ( &x and &y )
%eval ( &x or &y )
%eval ( &x ne &y )
%eval ( &x > &y )
%eval ( &x le &y )
```

The symbols, (and, or, ne, >, le) do not have a meaning in macro outside of the contexts that expect them, but they definitely do in expressions evaluated by %EVAL. You may write macro code without ever using a %EVAL, but you cannot avoid the issue because this function is called wherever an instruction or function expects an arithmetic expression. For example, consider:

```
%if ... %then ___ ;
```

Whatever goes between the %IF and the %THEN must be fed into %EVAL for logical evaluation to 0 or 1. In short the %IF statement has an implied call to the macro function, %EVAL. The statement

```
%do i = ... %to ... %by ... ;
```

has three implied calls to %EVAL, and the expression

```
%SUBSTR ( &string, ... , ... )
```

has two, as indicated by "...".

It is chiefly %EVAL that we must hide meaning from. Once you realize this it is easier to eliminate unnecessary quoting and concentrate your efforts where they are most needed. It also means that we can often remove the problem from a complex macro and use the statement,

```
%put %eval ( quoted expression ) ;
```

as a simple test to find out what we need to know about a quoting function. However, it is important, to remember to use the %PUT in a macro as was previously demonstrated using the macro named SEMICOLON.

Pair Symbols

Now what about quote marks, how does the macro facility treat them? Unlike SAS, in macro they are always part of the value being indicated and thus become a part of the generated program. However, they are also meaningful! The macro facility expects then to come in pairs, and it will not resolve macro expressions inside single quotes. That means that single quote marks indicate quoted materail to the macro facility. Double quotes were introduced into SAS when the macro language was developed in order to provide SAS with quoted material in which the macro facility would be able to generate the quoted expression. For example in the string, "%M", the macro facility will use the macro M to determine what the string will be generated. On the other hand, '%M' is just a four-character string.

All of this means that sometimes quote marks must be quoted, i.e. their meaning hidden from the macro facility. %STR does not directly hide them, but it provides the %-symbol as an escape character to allow such quoting, i.e. under appropriate conditions the character following a %-symbol is quoted (does not have its meaning). For example, consider the use of the code,

```
%macro label ;
   John's salary
%mend  label ;
```

in the SAS assignment

```
label = "%label" ;
```

It does not work because the single quote mark is meaningful, i.e. the %MEND statement becomes part of the value; hence the macro cannot even be compiled.

Using the escape character we have

```
%macro label ;
   %str(John%'s salary)
%mend  label ;
```

The enhanced editor will growl at this code because it has not been taught about quoting, but it is legal and will execute correctly.

This feature can also be used for parentheses, which would be part of the value, but also have a macro meaning that what is begun must be finished.

It is important to note that the %-symbol is not a general escape character; it only becomes an escape character when in an appropriate macro quoting function and in an appropriate context. For example, in

```
%str(%m)
```

The argument is a reference to a macro, M, and the %-symbol is not an escape character.

The %NRSTR quoting function

One can write

```
%str(%%)%str(m)
```

to mean just the two characters, a %-symbol followed by the letter M. The %-sign as an escape character is meant for only one character and should not be used within a string. Although the documentation is clear on this point, its significance did not occur to me until many years later. Why should a separate %STR be used on the M? Without it the macro facility still sees the expression as the invocation of a macro M. I do not know why?

However there is a second function, %NRSTR, which acts like %STR at compile time and hides all of the same symbols, except that it also hides the %-symbol and the &-symbol. One way to view this is that %STR hides the SAS meaning of symbols from the macro facility at compile time and %NRSTR hides in addition the macro meaning of symbols, i.e. the macro triggers. The macro triggers trigger a rescanning process in the macro facility; hence the NR in %NRSTR means no rescan.

In general quoting functions come in pairs, one to hide just SAS meaning and one with an NR prefix to in addition hide macro meaning.

Unquoting

Sometimes it is necessary to reveal the meaning of symbols at macro execution time that required hiding at macro compile. This means we need an %UNQUOTE function to remove macro quoting.

We have functions to control levels of quoting and when the quoting action takes place. Why is there only one function for removing macro quoting? There is only one state of having a symbol revealed and from the macro facility point of view that time is at execution time. Note that revealing a quoted meaning at compile time would simply means it shouldn't have been hidden at compile time in the first place.

Do not confuse %UNQUOTE with the newer SAS function DEQUOTE. DEQUOTE changes the value, while %UNUOTE changes the meaning but not the value. DEQUOTE is a convenience while %UNQUOTE is a necessity.

Note that this means a plain comma and a quoted one will be compared equal by %EVAL. For example, when the following code is executed,

```
%macro equal ;
   %local x ;
   %let x = %str(,) ;
   %let x = %unquote ( &x ) ;
   %put eval of &x=%str(,) ;
   %put %eval ( &x = %str(,) );
%mend  equal ;

%equal
```

it will produce a 1 on the log as the result of the evaluation. But this is an important point, how do we know that %UNQUOTE really did its job and that X holds the unquoted version of the comma? Well we could add a line

```
%put %substr ( a &x b, 1 , 1 ) ;
```

When this is done, SAS screams that the %SUBSTR function has too many arguments. Why? It is because there is an extra comma that came from the reference &X. When the %UNQUOTE line is removed there is no screaming because %SUBSTR does not see &X as meaning a comma.

Incidentally, it looks like the macro facility may not look inside the parentheses following the %SUBSTR at compile time. The assumption can be proven by executing the following macro.

```
%macro comma ;
   %local x ;
   %let x = ab, 2, 1 ;
```

```
   %put %substr (&x) ;
%mend  comma ;

%comma
```

Here we see that it would be impossible to consider the correctness of the argument to %SUBSTR at macro compile time, because the value of X does not yet exist. No quoting was used in the above example because none was needed. In fact it is important that the commas in the value of X not be quoted at the time of the evaluation of %SUBSTR (&X). In general, the macro facility does not look at the argument of a macro function at compile time. Of course, %STR must look at the argument at compile time because this is when %STR does the hiding; but it can do no evaluation of variables or resolution of macro calls at this time since in general, they do not exist at this time.

Just how much does the macro facility not look at macro function expressions? Consider:

```
%macro q ;
   %local x p1 p2 ;
   %let p1 = ( ;
   %let p2 = ) ;
   %let x = ab, 2, 1 ;
   %put %substr ( &x &p2 ;
   %put %eval &p1 1 + 1 &p2 ;
%mend q ;

%q
```

This code correctly writes a "b" on one line followed by a "2" on the following line. Note that the macro facility did not even need to see a parenthesis for the %EVAL function. In the case of the %SUBSTR expression the left parenthesis is essential, but not the right one. It is inconsistencies like this that make it hard to know exactly what one can do and what one cannot. However, in this case, it is fortunate that using macro variables for the parentheses makes the code hard to read, and there is little need for this use of macro variables.

In the macro named EQUAL, I used two %LET statements for clarity. Now consider:

```
%let x = %unquote(%str(,)) ;
```

It accomplishes the same thing, but is a little harder to think about. If we unquote what is quoted, how is anything accomplished? Remember %STR acts at macro compile time, while %UNQUOTE acts when the %LET assignment is executed. So, in fact, the combination simply means hide the argument from the compiler, but do not let it stay hidden at execution time. It is your choice whether to say this in one assignment or two.

Throughout this discussion, I have referred to %STR as a macro function, as do the manuals and almost everybody referring to %STR. However, we have seen that %STR is not a function in the sense that %SUBSTR is a macro function. It would be clearer to think of it as a special form of compile time directive in the disguise of a function. I will continue to refer to it as a function, but remember that is just another little way to make macro quoting a confusing subject. %UNQUOTE, on the other hand, is a true macro function.

To see the power of %UNQUOTE, consider a %LET assignment with a macro invocation on the left side of the equal. For example,

```
%macro makename ;
   x
%mend  makename ;

%macro q1 ;
   %let %makename = 77 ;
   %put &%makename ;
%mend  q1 ;

%q1 ;
```

The first macro, MAKENAME, generates a variable name in the simplest manner possible. The macro, Q1, makes the assignment correctly and then perhaps naively tries to use it. The result is

```
&x
```

instead of the expected 77. What went wrong? Well you can see that the macro MAKENAME did it's job, however, it must have done it too late for the &-symbol to indicate resolution of the named variable. The macro facility saw an &-symbol but did not see it as calling for the value of a variable, since the macro facility did not see a name did not immediately following the &-symbol at compile time. Remember macro names begin with a letter or underscore. So first an &-symbol was generated and then an X was generated and both were written to the log.

What about the first %LET? There was no problem here. Whatever comes between the %LET and the =-symbol must be the name of a macro variable, and no test is made at compile time. This suggests that the =-symbol must be seen by the macro facility at compile time. It is easy to prove by making a macro variable to hold =-symbol, but remember the %LET must be inside a macro, i.e. compiled, because the open code %LET does not mind generating the =symbol.

Since the &-symbol was not understood, let's try unquoting to make sure the compiler understands.

```
%macro q2 ;
    %let %makename = 77 ;
    %put %unquote(&%makename) ;
%mend  q2 ;

%q2 ;
```

We get the same thing. Oops, remember that %UNQUOTE works at execution time while we have already seen that the macro facility makes a decision about the meaning of the &-symbol at compile time. Well we have learned something anyway. What? The decision is made once at compile time and that quoting is not used to record that decision. The only thing left is to hide the &-symbol from the macro facility at compile time. Then no decision will be made and now the %UNQUOTE will force the issue. Can the decision to evaluate be made during execution? The use of macro variable references in open SAS code would suggest that it is a necessary feature of the macro facility. Consider:

```
%macro q3 ;
    %let %makename = 77 ;
    %put %unquote(%nrstr(&)%makename) ;
%mend  q3 ;

%q3 ;
```

First the %NRSTR hides the &-symbol from the macro facility at compile time so no decision is made as to what the symbols mean. The macro call is not in the argument of the %NRSTR. So it will be resolved at execution time. The %UNQUOTE now does two things. It reveals the &-symbol, and it glues that symbol to the name X thus creating a reference to the variable. Now as we conjectured the macro facility must have a way to do the evaluation of the reference, &X, and it does. So the number, 77, is now written to the log.

Perhaps you have now learned why you never saw a macro invoked on the left side of the =-symbol in a %LET assignment. But there is a more important lesson here. %UNQUOTE can glue together symbols to make macro objects when the macro compiler does not see them as a macro object. Note that

```
%put %unquote(%nrstr(&))%makename ;
```

does not give the correct result. Why, because the macro facility sees the &-symbol and the name as two separate things rather than as a macro variable reference.

In this case, I forced the macro facility miss the evaluation of the desired "macro variable reference" in the macro Q1. However, it is easy to see that there is room enough for confusion that the macro facility can make a mistake and interpret consecutive symbols as separate entities and hence do the wrong thing. It is particularly confusing when the problem is passed on to SAS as a line that looks perfectly good but gets an impossible error message because the SAS compiler got mislead by the macro facility. Often a judicious use of %UNQUOTE will cure the problem.

One important example comes from quoting quote marks. Consider the code:

```
%macro mkfn ;
   %let dir = c:\junk\ ;
   %let mem = mystuff.txt ;
   filename q %str(%")&dir&mem%str(%");
```

```
%mend  mkfn ;

options mprint ;
%mkfn
```

The log produced

```
ERROR: Error in the FILENAME statement.
MPRINT(MKFILENAME):
   filename q "c:\junk\mystuff.txt" ;
```

Note that there is no hint of what went wrong. The FILENAME statement looks perfectly good. It could be copied from the log and run without problem. You could complain that there was no need to quote the double quote marks, however, there is also nothing against this quoting. It follows all the rules and the quoting should be removed when the text is passed to SAS.

I think the problem is that the macro facility does not automatically unquote macro quoted quote marks. In any case, explicitly unquoting each quote mark individually cures the problem:

```
filename q %unquote(%str(%"))&dir&mem%unquote(%str(%"));
```

There is nothing special about the fact that I chose a FILENAME statement, although that together with the construction of operating system commands is where one is most likely to want this form of quoting. However, it usually comes from some requirement to enclose an expression in single quote marks where macro references are required in the expression.

To study the problem with single quotes consider this log based on the same principle as before, but with more SAS involvement.

```
21   %macro assignchar ;
22      %let x = abc ;
23      data w ;
24         z = %str(%')&x%str(%') ;
25      run ;
26   %mend  assignchar ;
27
28   options mprint ;
29   %assignchar
MPRINT(ASSIGNCHAR):   data w ;
NOTE: Line generated by the invoked macro "ASSIGNCHAR".
1     data w ;       z = '&x' ;    run ;
                                   ---
                                   180
ERROR 180-322: Statement is not valid or it is used out of proper order.


     -
     386
      ---
      202
MPRINT(ASSIGNCHAR):   z = 'abc' ;
MPRINT(ASSIGNCHAR):   run ;
ERROR 386-185: Expecting an arithmetic expression.


ERROR 202-322: The option or parameter is not recognized and will be ignored.
```

Again the quoting of the single quotes is not needed, but it is not incorrect according to any published rules of which I am aware.

I still think the problem is the failure of the macro facility to automatically unquote the macro quoted single quote marks. However, in this case it is more serious. Look at what happens when %UNQUOTE is applied to each quoted

single quote mark separately. A little code has been added after the macro is invoked in order to recover from the error and reveal it.

```
50   %macro assignchar ;
51      %let x = abc ;
52      data w ;
53         z = %unquote(%str(%'))&x%unquote(%str(%')) ;
54      run ;
55   %mend  assignchar ;
56
57   options mprint ;
58   %assignchar
MPRINT(ASSIGNCHAR):   data w ;
MPRINT(ASSIGNCHAR):   z =
59   *';
60   put z= ;
61   run;

z=&x%unquote(___) ;    run ;*
```

Note that the first quote mark got unquoted correctly (confirming the conjecture that it was not previously unquoted), but the second quote mark did not. Note the &X in the value of Z. This indicates that the macro facility did not see the macro variable reference. What happened? After handling the first quote mark, the macro facility is now processing an expression that begins with a single quote. Remember that we quoted it in the first place to avoid this situation. Now the macro facility treats everything as literal until the closing single quote. But this means that the closing %UNQUOTE will not be seen and consequently not acted upon; hence the macro facility has no way to work itself out of the single quoted expression that has been started unless there is a bare single quote at the end. This is impossible because the compiler expects quote marks in pairs. The underscores, shown above, inside the parentheses are how the Courier New font translated the boxes shown in the SAS log in the display manager for unprintable characters.

Please observe the problem is a logical one. You cannot allow quote marks to have a meaning to the macro facility and also allow them to be both quoted and unquoted separately. As soon as the first is unquoted, the second by definition can never be unquoted because the macro facility does not look inside a quoted expression. Thus the failure to automatically unquote single quote marks presents one with logical inconsistency.

The resolution of the problem lies in unquoting the entire expression rather than the individual problem points. On the other hand, I never understood the problem because I always used that solution. Use:

```
z = %unquote(%str(%')&x%str(%')) ;
```

Here the macro facility must look at the expression and resolve the reference, &X before the process of unquoting the expression. I think quoted single quote marks cannot be automatically unquoted without falling into the trap demonstrated above; hence the problem was never fixed and never will be. It is less clear why the problem cannot be fixed for double quotes.

Are there any other cases, where the macro facility fails to unquote macro quoted material? I no longer know. I did see some examples years ago before version 6, but I have not been able to reconstruct them and I do not know whether it is because the problems were fixed or it is simply that I no longer remember the circumstances well enough t recreate the conditions.

In any case, the older documentation gave the simple rule: If the mprint looks good and the SAS compiler does not understand it, then try %UNQUOTE.

Execution time macro quoting

We have seen that compile time macro quoting is sufficient for hiding symbols known at compile time. In other words, you have coded the symbol and the compiler sees it. That is the problem, you do not want the compiler to see it. %STR or %NRSTR will do the job.

Now consider a macro variable. Suppose it has a problem symbol in its value. The compiler has no problem because it does not resolve macro variable references. However, there is an execution time problem, when the variable reference is resolved. Thus there are times when you do not wish to hide the reference, but rather the result of what that reference produces. For example, I might have a comma separated list, say,

```
%let list = A, B, C ;
```

that I want to manipulate. The expression

```
%SCAN (&list, 2 )
```

cannot be used because at execution time the macro facility will see too many commas. %STR and %NRSTR are hopeless because they act at the wrong time.

The macro facility provides %QUOTE and %BQUOTE, to solve the problem. %QUOTE hides the standard stuff, just as %STR did. It can handle explicit pair symbols by using the %-symbol as an escape character, and the values of macro variables in some cases. For example:

```
%let rp = ) ;
%put %eval (%quote(%&rp) = %str(%)) ) ;
```

works fine at hiding the meaning of the value of RP, i.e. a right parenthesis. But it fails miserably when a letter is put in front of the right parenthesis. For example:

```
%let rp = x) ;
%put %eval (%quote(%&rp) = %str(x%)) );
```

%BQUOTE fixes some bugs in %QUOTE and adds the ability to handle pairing problems that arise from evaluation at execution time in a more general fashion. For example suppose you want to find out whether the value of a variable RP is a right parenthesis or not. The macro facility has no problem at compile time, but %EVAL will when it is prematurely ended. Using %BQUOTE cures the problem.

```
%let rp = x) ;
%put %eval (%bquote(&rp)=%str(x%))) ;
```

Their cousins %NRQUOTE and %NRBQUOTE are kind of funny. For example, they do not hide macro variable references. They hide the meaning of an &-symbol when it cannot be part of a macro variable reference. For example, consider:

```
%let a = b ;
%let b = q ;
%put %eval(%nrbquote(&&&a)=q) ;
```

Here the reference &A is evaluated to B and then the reference &B is evaluated, so it is clear that the &-symbol is not being hidden.

%SUPERQ hides all meaning of the result of the first level evaluation of a macro variable. However, you must be careful to understand what this means. It does not mean the variable is not resolved. It means that nothing in the value of that variable will be resolved. Moreover, you must be careful to omit the &-symbol in the %SUPERQ reference to a variable. In other words,

```
%superq(a)
```

references the variable A, but no macro symbols in the value of A will be evaluated. If there are &-symbols in the argument of %SUPERQ, they will be processed. So what does

```
%superq(&a)
```

mean? A is a macro variable that names a macro variable, say B. So be is evaluated but no further rescanning will take place. The code:

```
%let a = b ;
%let b = q ;
%put %eval(%superq(a)=b) ;
```

produces a 1 because A evaluates to the letter b.

However,

```
%put %eval(%superq(&a)=b) ;
```

is false because %SUPERQ(&A) resolves to the letter q.

Conclusion

I began this paper thinking that I could present a complete reference to the macro quoting functions. However, I did not cover all of them or even everything about some of them. Instead I decided that it was far more important to present the ideas and tools to allow you to investigate and understand macro quoting.

Just as you can only come to understand the DATA step by writing a great many very short DATA steps, so you will have to write a great many short macros with some macro quoting function involved in order to get a sound understanding of the subject. If you have come to realize how much there is to understand about the subject and how to go about finding out that information via little study macros, then I have achieved my purpose.

Macro quoting is hard because:

- Quoting is inherently a hard subject.
- The interaction between two languages makes quoting harder.
- The timing issues involved are complex, when the generated language compiles and executes during the generation of its code.
- There are two macro languages involved - the one which is compiled between %MACRO and %MEND statements and the one which executes immediately in open SAS code.
- Bugs in the macro facility prevent recognition of a consistent pattern of how macro quoting works.

The choice of problem, and the choice of what kind of macro to write, can help the macro programmer to avoid much of the macro quoting difficulties simply by avoiding area where one is likely to face symbols that will need quoting. However, the restriction will prevent the development of some interesting programs, and the macro programmer is not free to avoid all quoting issues.

In writing a macro program there will always be a tradeoff between readability and absolute protection from what the consumer may provide to your macro. It is often wise to avoid a quoting issue when it is rather unlikely to occur.

In the abstract I suggested that the person, who thinks macro quoting is simple, doesn't understand the problem. I began to doubt the truth of that statement, but now see why it must be true. I hope that in reading this article, you too have come to a better understanding of why. Never again should you be intimidated by someone who tells you that macro quoting is simple.

Contact Information

The author may be contacted as follows:

Ian Whitlock
149 Kendal Drive
Kennett Square, PA19348

Email: IW1sas@gmail.com