ABSTRACT
Thoughtful design and careful development can make the difference between a one-off macro and a useful generic utility. My intention with this paper is to complement and extend previously published design considerations (see References & Recommended Reading). I present two macros that exemplify these concepts:

- \%varinfo (access variable attributes in-line, quickly duplicate variables from another data set); and
- \%autocallpath (reveal the path of auto-compiled sasautos macros).

INTRODUCTION
Information discovery is key to generic programming. A robust macro must know or learn details, and behave accordingly. The more it discovers, the less a user must specify, contributing substantially to macro flexibility and reliability. The first macro described below, \%varinfo, demonstrates several design and development considerations as it streamlines discovery of variable attributes, promoting consistent use of these attributes and contributing to efficient generic programming. The second, \%autocallpath, compensates for SAS’ lack of transparency in code integration (prior to v9).

Information about data, variables, data sets, the SAS® environment, and the external computing environment is often available for discovery, although ease of access varies wildly. The syntactically brief SAS-supplied macro \%datatyp(arg) readily identifies the argument as either NUMERIC or CHAR. By comparison, identifying the correct network location of an autocall macro can seem, even to an experienced programmer, impossible. SAS has corrected this in version 9, implementing a new system option: \texttt{options mautolocdisplay} (documented in SAS OnlineDoc 9.1 and later; see also \url{http://support.sas.com/faq/030/FAQ03095_3_mautolocdisplay.html}). However, for the many of us that remain a year or more from v9 this is little consolation.

Somewhere in between on the continuum of convolution, discovering variable attributes is a moderately cumbersome task that involves correctly querying a data set. The stripped-down process has three steps: open() the target data set; extract the information; close() the data set. A robust process will also gracefully and informatively handle emergent problems.

Opportunity & Objective
Programmers should recognize variable attribute discovery as the perfect opportunity for a SAS macro:

- a somewhat cumbersome, but
- routine, well-defined, predictable programming task,
- requiring error handling,
- that could be far simpler than native-SAS functionality allows.

SAS macros can be broadly divided into (1) Base SAS or (2) SAS Macro Language. Base SAS macros can include setup (libname, options, titles, etc.), DATA step and PROC processing. The latter involve pure SAS Macro statements (often extended to Base SAS functions through the \%sysfunc facility) and can return information in-line, exactly where requested. From a task perspective, this same divide basically separates macros that change data from those that merely query data. Data query processes generally have in-line solutions, which provide such benefits as clean, concise code without unnecessary macro symbol creation or interruption of program flow.

- An initial critical decision, therefore, is to determine your intent and objective: Are you summarizing, changing or creating data? Or are you discovering information? If discovery is your intent, an in-line solution is probably available, although may require unfamiliar or uncommon techniques (see References & Recommended Reading, below).

Both \%varinfo and \%autocallpath are in-line macros, using no Base SAS statements. Users can therefore substitute the desired information (a variable name, format, label, length, etc.) with a macro invocation, exactly as if they already knew the information. Furthermore, themselves in-line macros, other in-line macros can invoke either without penalty.

Design
\%varinfo demonstrates several principles that standard design considerations should include. A basic tenet of data collection (Never ask twice for a single datum.) has an equally important programming analog. The rationale for both
lies in the dependable fallibility of users:

- Do not ask the user what the system should know or figure out.

SAS function `varnum()` requires the user to specify a variable `name`, while others such as `varlabel()` retrieve information for a specific variable `number`. `%Varinfo` uses `%datatyp` to allow users the flexibility of specifying variable names, numbers or mixed lists, and converts between name and number as necessary, graciously sparing the user from remembering such unintuitive distinctions:

```sas
%if %datatyp(&nxtv) eq NUMERIC %then %do;
  process numeric reference ...
%end;
%else %do;
  process character reference ...
%end;
```

The costs of macro development are recovered only through efficiencies gained by their use. A flexible and intuitive interface is essential for promoting use.

- intuitive interface – An interface consistent with familiar SAS conventions is particularly intuitive. As a minimum, a macro library should establish and adhere to its own standard conventions (eg, naming conventions for macro files or common keyword parameters such as "dsin=" for an input data set). This not only eases use of the library, but improves readability of programs that use the library.

For example, the `%scan` function, given a negative index, parses a string from the right. `%Varinfo` supports this convention. The request `%varinfo(data-set-name, -1, num)` returns the number of the last variable (i.e., the total number of variables) in `data-set-name`. SAS functions and procedures typically support parameter aliases, an efficiency convention easily extended to `%varinfo`. `%Varinfo` also supports SAS variable name lists (`_numeric_`, `_character_`, and `_all_`).

- self documenting – An incomplete or incorrect invocation of `%varinfo` produces a log message detailing proper usage. Further, if `%varinfo` realizes that it will not complete successfully, it outputs a clear explanation to the log.

- error handling – Reasonable self-defense programming is always worthwhile; the challenge lies in defining "reasonable". You cannot perfectly defend against unforeseeable network problems or user creativity. Nevertheless, your development should include thoughtful consideration of the error-handling effort that each task and audience warrants. Complex tasks intended for a broad audience benefit the most from thorough error handling.

%varinfo may encounter problems along the way from two distinct sources: user interaction with the macro; SAS interaction with its own and external environments. What if the target variable does not exist? What if SAS cannot find or open the data set? What if SAS is unable to close the data set? A robust macro will skate over such problems and inform the user rather than failing. Controlled behavior eliminates user impression that a macro “doesn’t work”, clearly identifying external causes.

- common-use extensions – Careful consideration of how a utility will be used (or perhaps experience with an initial version) or how a macro's functionality could benefit related tasks can lead to useful extensions of original functionality. Be open to these, but be careful. Whenever extending a production macro, consider how changes affect existing programs. Ideally, adding a new feature will not impair default or original behavior.

After working with `%varinfo` for a while I realized that I could easily modify it to quickly duplicate variables from another data set by implementing a new information keyword (DUP). This greatly simplifies common tasks such as creating shell data sets, reading in lookup tables from flat files, enforcing variable attributes across summary data sets, etc., while having zero impact on previous functionality.

```sas
data shell;
  %varinfo(model, _ALL_, DUP)
  retain %varinfo(model, _NUMERIC_, NAME) .
    %varinfo(model, _CHARACTER_, NAME) " ";
run;
```

The first macro call produces the necessary `attribute` statements to guarantee that variables in data set `shell`
match the attributes of data set model; the subsequent two calls initialize the numeric and character variables to their appropriate missing values (thus eliminating from the log notes that "Variable ... is uninitialized.").

Development
The distinction between design & development is a bit gray, but the following considerations are closer to SAS code than to language-independent design decisions or user requirements.

- compensate for the unexpected – SAS functions varfmt(), varinfmt() and varlabel() return null if these attributes are not explicitly assigned. However, despite SAS defining default values in each case, these native SAS functions do not make use of these defaults. To eliminate problems that such null responses would create, %varinfo instead returns SAS-defined default values.

- %local statement – Protect external symbol tables, where "external" could be the global symbol table or the local symbol table of a calling macro. Double-check it before considering the macro final (see References & Recommended Reading for a related discussion and counter-example).

- properly paired statements – Leave the programming environment uncluttered. For example, follow an open() statement with a matching close(). Extending this rule to Base SAS macros, utilities should protect the work library, making sure to not overwrite data sets or litter the work library with intermediate data sets. The same applies to macro-specific libnames, filerefs, ods destinations, etcetera.

- well-trained, well-tested quoting – During development and testing, be aware of macro parameters and how they are processed. Consider these statements which implement aliases for variable attribute requests:

  ```sas
  %if "&chr" eq "NA" %then %let nxti = NAME;
  %else %if "&chr" eq "NU" %then %let nxti = NUM;
  %else %if "&chr" eq "LA" %then %let nxti = LABEL;
  %else %if "&chr" eq "LE" %then %let nxti = LEN;
  ```

  Between %if and %then, the macro processor expects an expression (arithmetical or logical) that %evaluates to 0 or 1 (see SAS OnlineDoc http://support.sas.com/onlinedoc/913/getDoc/en/mcrolref.hlp/a000208971.htm). Without quoting &chr and the comparison strings, the macro processor would fail to recognize the text string between %if and %then as the intended logical comparison, instead resolving %eval(LE eq LE) to 0; the string "LE eq LE" is just a string of characters with no logical interpretation. %Varinfo would fail.

  However, explicit quoting clarifies the logical expression and the macro processor recognizes the expression and resolves %eval("LE" eq "LE") to 1. Note that the quote style does matter, which is not really intuitive. Consider:

  ```sas
  %put Evaluate [LE EQ LE]: %eval(LE EQ LE);
  %put Evaluate ['LE' EQ 'LE']: %eval('LE' EQ 'LE');
  %put Evaluate ["LE" EQ "LE"]: %eval("LE" EQ "LE");
  %put Evaluate ["LE" EQ 'LE']: %eval("LE" EQ 'LE');
  ```

  Since the quotes alert the macro processor of a logical expression, you may expect the response sequence 0 1 1 1, respectively. However, the processor has a conflict to deal with: all characters are text. While the quotes highlight a logical expression, the processor remains beholden to its text-processing directive and considers single- and double-quotes distinct characters. The somewhat unexpected result is:

  Evaluate [LE EQ LE]: 0
  Evaluate ['LE' EQ 'LE']: 1
  Evaluate ["LE" EQ "LE"]: 1
  Evaluate ["LE" EQ 'LE']: 0

  This example is a valuable reminder that Base SAS and SAS Macro are distinct languages, with specifications and behaviors that differ, sometimes unexpectedly. In comparison to SAS Macro, above, Base SAS interprets logical comparisons in a more accommodating and intuitive manner:

  ```sas
  %macro ds_eval(logexp);
  data _null_;
  result = put(&logexp, 1. -l);
  put "Evaluate [%bquote(&logexp)]: " result;
  run;
  %mend ds_eval;
  ```
The results now match what many may have expected for the previous %eval example:

Evaluate ['LE' EQ 'LE']: 1
Evaluate ['"LE" EQ "LE"']: 1
Evaluate ['"LE" EQ 'LE']: 1

The developer must always be careful wherever a macro inserts or generates text: parsing a user-specified parameter could produce mnemonics, special characters, unbalanced quote marks, embedded quoted strings, etcetera. Proper development and testing in anticipation of such developments is essential.

Experimenting with and learning how SAS quotes macro text is only half of the challenge. Equally important is recognizing when the macro processor %unquote text (see Whitlock 2003 – better yet: read it, study it, experiment as Ian has). In %ds_eval, above, %bquote masks the potentially disruptive single- and double-quotes that appear once the macro processor resolves the put string. Initially, keep in mind two questions during macro development:

1) The "symbol" question For assigning symbols (eg, %let sym = expression), ask yourself the symbol question: ‘How will I use &sym in the subsequent code?’ Will you risk introducing an undesirable character or mnemonic when resolving &sym in subsequent code? If so, then quote the expression.

2) The "value" question For %evaluated expressions (eg, %if ... %then, above, or the infamous %do %while(%scan(&delimited_string, &index, &delimiter) ne );), ask the value question: ‘Could the value that %scan generates introduce an undesirable character or mnemonic into this statement?’ If so, then quote appropriately (ie, %qscan versus %scan, %qtrim versus %trim, etcetera).

"Undesirable" characters or mnemonics compose the familiar list that appears in related documentation. It is these you must sometimes hide from the macro processor:

' " ( ) + - * / < > = ¬ ^ ~ ; , blank
AND OR NOT EQ NE LE LT GE GT

If you are further concerned about emergent unmatched quotes or parentheses, then apply %bquote as necessary. With practice, you will soon recognize a %substr situation from one requiring %qsubstr; your work will become less frustrating and your stumbles quite brief.

A related topic is single- versus double-quotes in character constants (so-called "literals"). SAS attempts to resolve macro references, & and %, within double-quotes. If you do not need to invoke the macro processor, you most likely do not need double quotes. You can use single-quotes even around strings containing unbalanced single quotes (see http://v8doc.sas.com/sashtml/lrcon/z0780334.htm, "[W]rite a single quotation mark as two consecutive single quotation marks and SAS treats it as one. You can then surround the character constant with single quotation marks... The same principle holds true for double quotation marks ...").

• well-trained commenting – What is the difference between /* */, */ * ; and %* ; comments? SAS knows, so you should too. The macro processor ignores both */ */ and %* ; and neither are stored with the compiled macro. Therefore, neither style comment leaves any trace in the log upon macro invocation. The third style, */ ; comments, are processed as complete SAS statements and stored with the compiled macro. Only the familiar */ */ can appear anywhere a blank space is valid. Why does this matter? Because more than likely, an incorrect */ ; comment will sabotage an in-line macro. Proper comment style will also assist log review (ie, documentation of the code actually generated by the macro). The following comments subvert macro function and log review in different ways:

* [Bad comment #1, pernicious] DATA type determines next step *
%if %datatyp(&nxtv) eq NUMERIC %then %do;
 process numeric reference ... 
%end;

%* [Bad comment #2, useless] merge &ds2 into &ds1 to create &out *
data &out; merge &ds1 &ds2;
... merge details ...;
run;

The first will appear in the generated SAS code and log, but stripped of its macro context – which could easily derail an in-line macro. The second contributes nothing to the log, which is about the only place it could be useful. You may consider this a trivial distinction that only matters to excitable, purist programmers. Just wait until the first time you realize that a * ; style comment is responsible for the incomprehensible failure of your otherwise perfect macro.

In fact, a %put statement instead of a * ; comment can often be the most useful comment style within a Base SAS macro. Bad comment # 2, above, could be quite helpful in a %put statement (especially in the log):

`%put [Useful comment #2] merge &ds2 into &ds1 to create &out *;
data &out; merge &ds1 &ds2;
... merge details ...;
run;
`

The %put approach can be quite informative, especially for in-line macros that otherwise leave little trace in the log. For further details see the SAS OnlineDoc, starting with [http://v8doc.sas.com/sashtml/lgref/z0289375.htm](http://v8doc.sas.com/sashtml/lgref/z0289375.htm), [http://v8doc.sas.com/sashtml/macro/z0543665.htm](http://v8doc.sas.com/sashtml/macro/z0543665.htm) and [http://v8doc.sas.com/sashtml/lrcon/z0780334.htm](http://v8doc.sas.com/sashtml/lrcon/z0780334.htm).

- transparency – SAS, through version 8, is woefully weak in this regard. With the autocall facility enabled in a complex programming environment (ie, a complex hierarchy of aggregate sasauto filerefs, possibly with multiple versions of programs), how long would it take you to identify the source file associated with an auto-compiled macro? Think you’re fast and clever? Now consider how fast you could do the same for an archived project in which you were not originally involved. Left to its own devices, SAS makes this task nearly impossible from log review.

I recommend two approaches. At least design a macro to announce itself on first access. %Varinfo does this by logging its compilation:

```sas
%put;
%put *+=================================+*;
%put *|         MACRO %nrstr(%VARINFO())        |*;
%put *|   Return variable information   |*;
%put *|        Version 2.1           |*;
%put *|        Date: 09-feb-2005        |*;
%put *+=================================+*;
%put;
%macro varinfo ...
%mend varinfo;
```

Note that since the %put banner is outside the macro definition, the statements appear in the log only once, labeling macro compilation and first invocation.

Complete pathname for the auto-compiled file would be even more useful. This should be much simpler beginning with SAS version 9. Until then, only with background effort and brute force, %autocallpath provides the missing simplicity. Invoke the macro before the definition (eg, as part of the %put banner, above) to minimize log noise: %autocallpath(varinfo). The macro searches sasautos paths just as SAS would and logs the first version of the file it encounters. This macro loses some value once we’ve all migrated to SAS v9, but will nonetheless remain helpful in interactive sessions.

CONCLUSION
A good macro is worth the effort; an intuitive, robust and well-documented macro even more so. I hope I’ve inspired fresh thoughts on macro design, and new appreciation for the value of careful implementation and attention to detail.

REFERENCES & RECOMMENDED READING

For an eye-opening example of a useful but uncommon technique of querying a SAS data set without using DATA step processing, examine SAS’s solution for determining whether a macro variable exists (http://support.sas.com/faq/030/FAQ03072.html). Note that the author considers this macro subtly flawed: the %local statement is incomplete, and the local symbol names are insufficiently unique to reasonably guard against misleading responses. For example, SAS’ %check(dsid) will respond “yes” even if the symbol &dsid only exists during execution of %check. That’s most likely a misleading response, once SAS returns its attention to the calling program. Note also that with SAS version 9, this macro is obsolete (try %symexist, documented in SAS OnlineDoc 9.1 and later; see also http://support.sas.com/faq/011/FAQ01183.html). The technique, however, is not.

CONTACT INFORMATION
Your comments and questions are valued and encouraged. Contact the author at:
  Dante diTommaso
  Novartis Pharma
  CH-4002, Basel, Switzerland
  Work Phone: +41 61 696-2405
  Email: dante.ditommaso@novartis.com
  Web: http://mypage.bluewin.ch/ditommaso/sas/

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.