

Top-Down Programming with SAS® Macros

Edward Heaton, Westat, Rockville, MD

ABSTRACT

Structured, top-down programming techniques are not intuitively obvious in the SAS language, but macros can be used to approximate a top-down structure. This can lead to programs that are more versatile, more robust, and generally easier to develop. This presentation will review a SAS macro that reads elements from a difficult-to-read ASCII text file that is typically used to control the processing for a COBOL job. Along the way, we will look at the program structure and review the engineering decisions that allow the code to be developed in an organized and robust manner.

This presentation will demonstrate methods for macro testing and debugging, as well as various features of the SAS Macro Facility.

INTRODUCTION

Top-down programming can make your programs much more easy to understand, develop, debug, and modify. This paper will demonstrate top-down programming while creating a utility to facilitate the creation of value-label formats.

One task that greatly benefits from an orderly, top-down structure is the reading of a complex external file. We will develop a SAS macro that reads a meta-data file to obtain variable and format information.

Westat developed a COBOL program to read a free-format text file and write files that facilitate many of the steps of survey processing. That text file (hereafter called a source file) is a useful first-step in creating value labels. In this presentation we will create an external SAS macro that can be called from a SAS autocall library. The macro will read the source file and will write a SAS data set. That data set needs to contain all the information necessary to create value-label formats. While we develop this macro, we want to write code that will be easily adaptable to other stages of our survey processing cycle. We will not address those other stages, but certain programming techniques help to assure that adaptability.

FILE FORMAT

Let's start by defining the source file. Perhaps the best approach is to look at the specifications for creating the source file.

1. The file is free-format; words do not need to begin and end in specific columns.
2. The statements shall be typed in lines of 100 characters or fewer, including spaces, but they may span several lines.
3. A virgule (/) marks the beginning of a line continuation. Line continuations are for the descriptive parts of the statement, such as the wording of a question or the description of a value code.

4. Each statement begins with a key letter that identifies the statement type:

- P Parameter – We will not process this statement.
- H Header – We will not process this statement.
- V Variable – Variable statements define properties of the variable.
V name width type
 E.g.: To define variable Q1 as two columns wide containing numeric data, the source file will have the following statement.
V Q1 02 N
 For this task, we will want to read the name of the variable and its data type.
- Q Question – If we were creating variable labels, we would want to read the wording of the question. We are not so we will not use this statement.
- C Code – The code statement defines the values of the variable defined in the preceding V statement. A + indicates a "missing" value. You can link a C statement to an S statement via one or more asterisks (*).
*C [*****] value = text*
 E.g.:
C ++ = inapplicable
C 01-70 = number of years
C 98 = don't know
C 99 = not ascertained
- R Remark – This is a comment statement and has no bearing on our task.
- S Skip pattern – Skip patterns tell us which variables to skip based on the values of the variable defined in the preceding C statements. We are not going to deal with skip statements; they have no bearing on this task.
- A Abbreviation – The A statement will not be processed by our macro.

Our paper will work with the variable and code statements only. The source file might look something like the following:

```
P RECLN 200
H 1 U.S. Department of Education
H 2 National Center for Educational
  / Statistics
H 3 Internet Access in U.S. Public Schools,
  / Fall 2000
V QA 01 A
Q What is the title/position of the
  / Respondent?
C 1 = Technology Coordinator
C 2 = Library/Media Specialist
C 9 = Not Ascertained
V QB 01 A
Q Does the respondent have email?
C 1 = Yes
C 2 = No
C 9 = Not Ascertained
V Q1 04 N
Q What is the total number of instructional
  / rooms in your school? (Include all rooms
  / used for any instructional purposes:
```

```

/ classrooms, computer labs and other labs,
/ library/media centers, etc.)
C 0001-0250 = Total number of Instructional
/ Rooms
C 9999 = Not Ascertained
V Q2A 04 N
Q How many computers are there in your
/ school? (Count all computers, including
/ those used by administrators, teachers, and
/ students.)
C * 0000 = No computers in school (Skip to
/ Q13.)
C 0001-0550 = Total number of computers in
/ school
C 9999 = Not Ascertained
S * Skip Q2B - Q12DC and code as +
...

```

THE PLAN OF ATTACK

We will create a SAS data set that can be written to an MS Access® database. The **Label** field of that database will be edited to and then used for value labels – the wording of value labels is usually done by non-programmers at our site. When all the labels have been updated, we use PROC DATASETS to attach the labels to the variables. The code to write the SAS data to an existing MS Access database will look something like the following. (For this example, the **%CreateValueLabelsDataSet** macro is stored in the **Macros** sub-folder. The source file is in the same directory as this job, and it is called **codebook.dat**. We are creating a SAS data set called **ValueLabels** and an MS Access table called **ValueLabels**.)

```

Options sasAutos= ( "Macros" sasAutos ) ;

LibName labels odbc complete=
    "dsn=MS Access 97 Database;
    dbq=EasyLabels.mdb"
;

%CreateValueLabelsDataSet(
    inFile="codebook.dat"
    , cntlOut= ValueLabels
)

/* Write data to Access. */
Data labels.ValueLabels ;
    Set ValueLabels (
        rename= ( Label =LongLabel )
    ) ;
    Length Label $40 ;
    Label = LongLabel ;
Run ;

LibName labels clear ;

```

The **%CreateValueLabelsDataSet** macro read the source file and wrote the SAS data set. Then we wrote that data to MS Access, in effect shortening the length of the **Label** variable to 40 characters and creating a variable called **LongLabel**. **LongLabel** contains the entire text part of the code statement. When the MS Access table has been reviewed and updated we will use the data to create value label formats and attach those value label formats to our SAS data set with code similar to the following. By referring to SASAUTOS in the SASAUTOS= option, you make sure that all of the standard SAS autocall libraries are included.

```

LibName project "Library" ;

Options

```

```

    fmtSearch= ( project )
    sasAutos= ( "Macros" sasAutos )
;

LibName labels odbc complete=
    "dsn=MS Access 97 Database;
    dbq=EasyLabels.mdb"
;

Proc format
    library=project
    cntlin=labels.ValueLabels( drop=
        VarName LongLabel
    )
    fmtLib
;
Run ;

/* Create a variable-format list as a macro
variable. */
Proc sql noprint ;
    Select
        trim(VarName) || " " ||
        trim(FmtName) || "."
        into :FormatString separated by " "
        from labels.ValueLabels
    ;
Quit ;

/* Apply the variable-format list to the data
set. */
Proc datasets ;
    Modify project.OurData ;
    Format &FormatString ;
    Run ;
Quit ;

LibName project clear ;

```

As you can see, these jobs are relatively short and simple. They could, however, be developed into macros to make the task even more simple.

EXECUTION OF THE PLAN

STANDARD DOCUMENTATION

Let's start with the standard job documentation. I use a template that prompts me for all the information that I typically want in my header.

```

/*****
MACRO:

OBJECTIVE:

PROGRAMMER:
    Ed Heaton, Senior Systems Analyst,
    Westat,
    1550 Research Boulevard, Room 2018,
    Rockville, MD 20850-3159
    Voice: (301) 610-4818
    Fax: (301) 294-3992
    mailto:EdwardHeaton@Westat.com
    http://www.Westat.com

DETAILS:

INPUT:

OUTPUT:

STORAGE:

AUDIT TRAIL:
    yyyymmdd EH : Developed macro.
    *****/
%Macro MName ( debugging=0 ) ;

```

```

%If &debugging %then %do ;
  %TurnDebugOptions( ON )
%End ;

/* SAS code and calls to other macros */

%If &debugging %then %do ;
  %TurnDebugOptions( OFF )
%End ;
%Mend MName ;
/*****

```

Maybe I need to say something about the **debugging=** parameter. I include it whenever I develop a macro. It is used for, as you guessed, debugging. The default value is zero. I pass **debugging=1** if I want to debug the macro. Then I can use this parameter to do things that I want to do only when I am in debugging mode. We will see examples of this feature in this paper. Even if I do not have immediate plans for the **debugging=** parameter, I include it. Then it is available for future use if I find a problem later on.

THE TURNDEBUGOPTIONS MACRO

When in debugging mode (called with **debugging=1**), a macro calls another macro called **%TurnDebugOptions**. That macro will either set or reset certain systems options that write copious amounts of information to the system log, depending on the value passed to the macro. The macro might be useful to you, so let's take a few minutes and look at it.

```

/*****
MACRO: TurnDebugOptions

```

OBJECTIVE:

This macro will turn macro debugging options ON or OFF. To turn them on, call this macro passing in ON. This macro will first retrieve original values of SAS system options MPRINT, SOURCE, NOTES, SYMBOLGEN, and MLOGIC. It will then set the system options to MPRINT, MLOGIC, SOURCE, NOTES, and SYMBOLGEN so that we can follow the workings of macros. To turn off the debugging options and restore the original SAS system option values, call this macro with the parameter set to OFF.

USAGE:

The most common usage of this macro will be to add the **DEBUGGING=0** parameter to your macro statements. Then, if **DEBUGGING=1** is passed in to your macro, call this macro passing "ON" and, at the end of your macro, call it again passing "OFF". E.g.:

```

%Macro Sample ( debugging=0 ) ;
  %If &debugging %then %do ;
    %TurnDebugOptions( ON )
  %End ;
  Macro Code
  %If &debugging %then %do ;
    %TurnDebugOptions( OFF )
  %End ;
%Mend Sample ;

```

PROGRAMMER:

...

STORAGE: ...

AUDIT TRAIL:

20000613 EH : Wrote code combining the ...
20000614 EH : Added code to allow ...

```

*****/
%Macro TurnDebugOptions ( switch ) ;
  %If ( %upCase(&switch) eq ON ) %then %do ;
    %Global
      _mPrint
      _mLogic
      _source
      _notes
      _symbolGen
    ;
    %Let _mPrint =
      %sysFunc( getOption( mPrint ) )
    ;
    %Let _mLogic =
      %sysFunc( getOption( mLogic ) )
    ;
    %Let _source =
      %sysFunc( getOption( source ) )
    ;
    %Let _notes =
      %sysFunc( getOption( notes ) )
    ;
    %Let _symbolGen =
      %sysFunc( getOption( symbolGen ) )
    ;
    Options
      mPrint
      mLogic
      source
      notes
      symbolGen
    ;
  %End ;
  %Else %if (
    %upCase( &switch ) eq OFF
  ) %then Options
    &_source
    &_notes
    &_mLogic
    &_mPrint
    &_symbolGen
  ;
  %Else %put
    TurnDebugOptions must pass ON or OFF.
  ;
%Mend TurnDebugOptions ;
/*****

```

So you see, if a calling macro passes **ON**, the **%TurnDebugOptions** macro checks the current settings of five systems options using the **GETOPTION** function and then writes that value to macro variables using **%SYSFUNC**. If **OFF** is passed, the macro uses the systems options settings that were stored in these macro variables to restore the options to the original settings. If the **%TurnDebugOptions** macro is called twice passing **ON** without calling it passing **OFF**, the original systems settings will be lost. It is the responsibility of the programmer to avoid this situation.

TOP-DOWN PROGRAMMING – THE TOP

Let's define the steps to create a data set of value labels. The **%CreateValueLabelsDataSet** macro exists in a file called **CreateValueLabelsDataSet.sas**. Let's define a *top-level* macro as a macro that exists in a file of the same name.

```

%Macro CreateValueLabelsDataSet (
  inFile=
  , cntlOut=
  , debugging=0
) ;
  %If &debugging %then %do ;
    %TurnDebugOptions( ON )
  %End ;

```

```

/* Create a SAS data set that has all the
data that we need. */

%If &debugging %then %do ;
    %TurnDebugOptions( OFF )
%End ;
%Mend CreateValueLabelsDataSet ;

```

Whew, that was easy – just one step! Is that all there is to it? Well, yes; but the devil *is* in the details. Let's create the DATA step. Note the macro %IF statement embedded in the DATA statement below. I typically start all temporary variable names with an underscore. That way, I can drop all of them with DROP=_: (The colon is a wild card, like the asterisk in Windows.). If I am not in debugging mode, this will drop all variables that start with an underscore.

```

Data &cntlOut (
    %If not &debugging %then drop=_: ;
) ;

    %_DefineVariables( debugging=&debugging )

    Retain VarName FmtName Type ;

    InFile &inFile truncOver end=eof ;

    %_ReadOneStatement(
        cntlOut=&cntlOut
        , debugging=&debugging
    )

    If ( upCase( _StatementType_ ) eq "C" )
        then output &cntlOut
    ;

Run ;

```

We want to define our variables (the ones that we want to keep). I almost always find this a useful first step. We retained the **VarName**, **FmtName**, and **Type** variables because there are usually multiple **C** (case) statements per variable and we want to write an observation for each **C** statement. Each observation needs the variable name, the format name, and the data type. Of course, we need to read the source file. The output data set needs data for the **VarName**, **FmtName**, **Start**, **End**, **Label**, and **Type** variables.

After the DATA step, let's include code to print the data whenever we are in the debugging mode. We can print the whole data set; it shouldn't be too long. But let's print only the first 40 characters of **Label** because that value might be quite long.

```

%If &debugging %then %do ;

    Title4 "&cntlOut" ;

    Proc print data=&cntlOut ;
        Format Label $40. ;
    Run ;

    Title4 ;

%End ;

```

SUPPORTING MACROS – THE NEXT LEVEL

Now we need to create the two macros that we called. Let's define a *supporting* macro as one that exists in the same *.SAS file as a *top-level* macro and is called by another macro in that file. In keeping with the top-down

concept, let's define *supporting* macros below the *top-level* macro. I always start a *supporting* macro with an underscore for two reasons: (1) the macro name is more unlikely to be used in the job that calls this *top-level* macro; and (2) it will be easier to explain to the users, when its name appears in the SAS log, that it is a *supporting* macro and that the users will not find a *.SAS file by that name. Let's create the macro that defines the variables. I have three criteria for deciding to create a macro rather than just including the code in the calling job.

1. Is the code used more than once? Whenever I have the same code in two different places, then I will surely, at some point, update it in only one of those places. Then my code will either not work or, worse, will work intermittently based on the data.
2. Will the code make the program block too long? I find it hard to visualize code when I can't see it from start to end on one page or screen. I.e.: I need to be able to see the start and end of a DO block at the same time. Ideally, I can see the entire DATA step, SQL statement, or PROC as a unit. I would rather see place-holders for code (i.e.; macro calls) than not be able to see both the start and the end. My rule of thumb is that I don't like any macro or program step to be over a half a page long if I can easily keep it shorter.
3. If I don't know how to write the code, I can simply insert a descriptive macro call and go on with the code that I do know how to write. (Often, coding tasks that I do know how to code will help me to see the algorithm for tasks that I didn't understand.) Then I can go back and create the macro that I called. (Sometimes, we need output from a macro but do not know how to get the values from the data. We can use the **debugging=1** parameter to hard-code values in the otherwise empty macro. Then we can continue with development of the job at hand and return to the undeveloped macro when we understand more.)

DEFINE VARIABLES

The **%_DefineVariables** macro will simply make the **DATA** step short enough to see in its entirety.

```

%Macro _DefineVariables ( debugging=0 ) ;

    Attrib
        VarName
            label= "Variable Name"
            length= $ 6
        FmtName
            label= "Format Name"
            length= $ 8
        Start
            label= "Starting Value for Format"
            length= $ 16
        End
            label= "Ending Value for Format"
            length= $ 16
        Label
            label= "Format Value Label"
            length= $200
        Type
            label= "Type of Format"
            length= $ 1
    ;

%Mend _DefineVariables ;

```

Why did we set the length of the variable name to six characters? SAS allows 32 characters. The problem is with the format name; SAS still allows only 8 characters here. To keep our process algorithmic, I want to systematically create a format name with the variable name as its base. Variable names can end with a digit; format names cannot. So I want to append an eff (**F**) onto the end of the variable name. (E.g.: If the variable name is **Q59a12**, I want the format name to be **Q59a12F**.) What if the variable is type character? Then the format name needs to start with a dollar sign (\$). (E.g.: **\$Q59a12F**.) So, you see, a 6-character variable name can become an 8-character format name.

READ ONE STATEMENT

Now we will read the values for the needed variables and output the data set. However, given the nature of our input file, this seems like a big task. Let's break it down. We saw that the source file has seven different kinds of statements. We need to determine the type of statement and process it. Let's read the first non-blank character, call it the statement type, and hold the record. Since the source-file statements can span lines, we need to hold the record with a trailing @@. Then we will use the SELECT statement to set our course.

We need only process the **V** and **C** statements; a **V** statement will provide a variable's name and its data type; and a **C** statement will provide a value label. The other statements will be skipped. However, we will want to keep the select statement very general because we will want to expand our macro to get variable labels from the source file and, maybe, to collect skip-pattern information so that we can use the data to create a data dictionary. We don't want to significantly change this macro once it works correctly, so let's make sure it is easily expandable.

```
%Macro _ReadOneStatement ( debugging=0 ) ;

  Input @1 _StatementType_ : $1. @@ ;

  Select ( _StatementType_ ) ;
    When ( "P" ) do ;
      %_ReadPStatement(
        debugging=&debugging
      )
    End ;
    When ( "H" ) do ;
      %_ReadHStatement(
        debugging=&debugging
      )
    End ;
    When ( "V" ) do ;
      %_ReadVStatement(
        debugging=&debugging
      )
    End ;
    When ( "Q" ) do ;
      %_ReadQStatement(
        debugging=&debugging
      )
    End ;
    When ( "C" ) do ;
      %_ReadCStatement(
        debugging=&debugging
      )
    End ;
    When ( "R" ) do ;
      %_ReadRStatement(
        debugging=&debugging
      )
    End ;
```

```
  When ( "S" ) do ;
    %_ReadSStatement(
      debugging=&debugging
    )
  End ;
  When ( "A" ) do ;
    %_ReadAStatement(
      debugging=&debugging
    )
  End ;
  When ( "/" ) do ;
    %_VirguleErrMsg(
      debugging=&debugging
    )
  End ;
  When ( " " ) do ; /* Empty Row */
    %_ReadNextRecordCode(
      debugging=&debugging
    )
  End ;
  Otherwise do ;
    %_GenErrMsg(
      debugging=&debugging
    )
  End ;
End ;

%Mend _ReadOneStatement ;
```

This is long, but it does fit my half-page criterion when I have longer lines of code. (That is, each WHEN statement, including the DO block, will fit on one line of code.) This macro called a lot of other macros that we will have to define.

Three notes:

1. The colon (:) format modifier tells SAS to start reading at the next non-blank column. So the statement-type identifier does not have to be in the first column.
2. Our source code can have blank records. We do not want to output these, so our macro will need to explicitly tell SAS when to output an observation. If we come to an empty record, we will simply move to the next record.
3. Every time we read a statement, we need to read all the way through every line continuation of the statement. So the **%_ReadOneStatement** macro should never see a line continuation.

LEVEL 3: SUPPORTING THE SUPPORTING MACROS

*Working on a SAS job
Going down, down, down.
Working on a SAS job;
Whoop, I wanna lie down.*

*Five o'clock in the morning,
My code is still not done.
Lord I'm so tired.
How long can this go on?*

I'm a working on a SAS job ...

Okay, the **%_ReadOneStatement** macro was easy because it only decided which other macros to call. Let's define these macros.

READ P STATEMENT

We don't need the **P** statement for this task. However, we

do want to address it and to write our code so that it is clear where and how to change things if we want to expand the scope of our macro.

```
%Macro _ReadPStatement ( debugging=0 ) ;

    %_GoToNextStatement (
        debugging=&debugging
    )

%Mend _ReadPStatement ;
```

So, this macro simply calls another macro. Does this seem a little trite? On the surface, yes. We could have called the **%_GoToNextStatement** macro directly. Remember, however, that a macro that works is best left alone. If we called **%_GoToNextStatement** directly from **%_ReadOneStatement**, we would have to change both the **%_ReadOneStatement** and **%_ReadPStatement** macros if we want to do something with the **P** statement. This way, we only have to change the **%_ReadPStatement** macro.

We need similar macros for each of the statements that we do not need for this task. We don't need the **H**, **Q**, **R**, **S**, and **A** statements.

READ V STATEMENT

This macro will read a variable statement. Variable statements start with a vee (**V**). Variable statements are in the form

V name width type [possibly more stuff we don't need]

The following statement

V Q1 02 N

defines variable **Q1** as two columns wide, containing numeric data. Since the source file is written in a free-format language, you can place the vee in a column other than column 1 and insert as many spaces between the elements as wanted. However, the elements must occur in the order as specified.

The variable name may be any valid SAS variable, except that we artificially restricted the length to six characters or fewer for this macro. COBOL keywords cannot be used because the standard job to read this file was written in COBOL.

```
%Macro _ReadVStatement ( debugging=0 ) ;

    Input VarName $ _Length_ $ Type $ @@ ;

    %_ReadNextRecordCode (
        debugging=&debugging
    )

    FmtName = trim( VarName ) || "F" ;

    Select ( Type ) ;
        When ( "G" ) delete ;
        When ( "N" ) ;
        When ( "A" ) do ;
            Type = "C" ;
            FmtName = "$" || FmtName ;
        End ;
    End ;

%Mend _ReadVStatement ;
```

Not too bad. Statement type **G** denotes a group of variables, and is not a part of the SAS data set. So we will drop that record.

As you can see, we had to do some coding here. But the macro is still quite easy to visualize. We will need to define the **%_ReadNextRecordCode** macro before we are done, but we need that macro for our **%_ReadOneStatement** macro anyway.

READ C STATEMENT

The code statement equates coded (actual or range of) values to a description. If a survey item (such as a verbatim answer or a comment) does not have codes, no **C** statement is required. You have the option to link the **C** statement to a skip pattern statement by one or more asterisks. The code statement consists of five elements:

C [***] value = text**

You use the **+** (blank) code when the question is skipped.

```
%Macro _ReadCStatement ( debugging=0 ) ;

    %_ReadValueRange ( debugging=&debugging )

    %_ReadValueLabel ( debugging=&debugging )

%Mend _ReadCStatement ;
```

Okay, we have read the **V** statement to get the variable name and its type. From that, we have built the format name. Now we have read the starting and ending values and the label. We will need to define the **%_ReadValueRange** and **%_ReadValueLabel** macros.

READ NEXT RECORD CODE

The **%_ReadNextRecordCode** macro was called by the **%_ReadOneStatement** and **%_ReadVStatement** macros. This macro simply looks ahead to see what the statement identifier is for the next line of code.

If we are *on* the last record we cannot *advance to* the next record. Duh! So simply release the record with the **INPUT** statement (no trailing **@@**). Otherwise advance to the next record and read the first character.

```
%Macro _ReadNextRecordCode (debugging=0) ;

    If EOF
        then input ;
    Else input
        / @1 _NextRecordCode_ : $1. @@
    ;

%Mend _ReadNextRecordCode ;
```

Hey, this macro calls no others! So this is the end of this branch and the code was really quite simple.

VIRGULE ERR MESSAGE

The **%_VirguleErrorMessage** macro is called by the **%_ReadOneStatement** macro, but it really should never be called at all. That is to say, all line continuations should be processed by one of the **%_Read?Statement** macros where **?** is in **{P,H,V,Q,C,R,S,A}**. However, if the source file is badly written we might try to read a line continuation from the **%_ReadOneStatement** macro and we will need to know that something is wrong.

```

%Macro _VirguleErrMsg ( debugging=0 ) ;

  Put
    "ERROR: / records should never be"
    " accessed from the"
    " _ReadOneStatement macro!"
    / _inFile_
  ;

  %_GoToNextStatement (
    debugging=&debugging
  )

%Mend _VirguleErrMsg ;

```

GEN ERR MESSAGE

Suppose a statement starts with a symbol other than those in { **P,H,V,Q,C,R,S,A,/**}. Then we have another error in the source file that we need to know about. So let's write this error and move on.

```

%Macro _GenErrMsg ( debugging=0 ) ;

  Put
    "WARNING: " _StatementType_
    "is not coded."
    / _inFile_
  ;

  %_GoToNextStatement (
    debugging=&debugging
  )

%Mend _GenErrMsg ;

```

We still have macros that have been called but not defined. We need to go deeper with our programming.

DEEPER AND DEEPER

GO TO NEXT STATEMENT

The **%_GoToNextStatement** macro was called by the **%_ReadPStatement**, **%_ReadHStatement**, ... macros. So let's define it here.

Statements can span records, so we need to keep skipping records until the next record code is not a line continuation. Since we have already defined the **%_ReadNextRecordCode** macro, we will use it here.

```

%Macro _GoToNextStatement ( debugging=0 ) ;

  %_ReadNextRecordCode (
    debugging=&debugging
  )

  Do while ( _NextRecordCode_ eq "/" ) ;
    %_ReadNextRecordCode (
      debugging=&debugging
    )
  End ;

%Mend _GoToNextStatement ;

```

Again, this is very simple code. We have already defined the **%_ReadNextRecordCode** macro.

READ VALUE RANGE

The **%_ReadValueRange** macro was called by the **%_ReadCStatement** macro. Let's investigate how to build this macro.

The value that we want to read might be a single value or

it might be a range of values of the form *lowest value – highest value*. It also might have a series of asterisks before the values that indicate a condition that prompts us to skip certain questions. About the only thing that all of the **C** statements have in common is an equals sign that separates the value label from the rest of the statement.

Let's define the a temporary variable, called **_Value_**, with plenty of space so that we can be sure to include all the source-file code for the value. How long does it need to be? Well, the limit for a record in the source file is 100 characters. The first character must be a statement identifier or a line continuation indicator. That leaves 99 characters. So let's make **_Value_** 99 characters long. Why not; it's a temporary variable. The value, or range of values, is the part before the equal sign, with any asterisks (skip indicators) removed. Since the source code is free format, and spaces have little meaning, let's compress out the spaces so that we have consistent code to process. Plus signs (+) indicate missing values; multiple plus signs simply reflect the length of the field. So we will need to convert the source file's missing value indicator to the appropriate SAS missing value indicator. For now, we will simply insert a macro call to indicate this task needs to be coded. We will probably want to pass the name of the variable that we want to set to missing, and the name of the variable that specifies the data type, in case this macro sees further use.

If we are working with a range of values rather than a single value, we will need to parse that range into the starting value and the ending value. If we are working with only one value we will simply copy the value of **Start** to **End**.

```

%Macro _ReadValueRange ( debugging=0 ) ;

  Input _Value_ $char99. @@ ;

  _Value_ = scan( _Value_ , 1 , "=" ) ;

  _Value_ = compress( _Value_ , " *" ) ;

  If index(_Value_,"+")
  then do ;
    %_SetPlusToMissing(
      var=_Value_
      , type=Type
      , debugging=&debugging
    )
  End ;
  Else do ;
    Start = scan( _Value_ , 1 , "-" ) ;
    End = scan( _Value_ , 2 , "-" ) ;
    If ( End = " " ) then End = Start ;
  End ;

%Mend _ReadValueRange ;

```

The SCAN function looks for words, and allows us to specify word partitions. So **SCAN(_Value_ , 1 , "=")** tells SAS to partition the character string called **_Value_** into smaller character strings using the equal sign (=) as the partition delimiter. The "1" tells SAS to return the first partition. So this gives us all the characters between the statement identifier (**C**) which is not part of **_Value_** and the equal sign.

The COMPRESS function removes characters specified in

the second argument from the first argument. So **COMPRESS(_Value_ , " *")** will remove the skip-pattern indicators; we are not dealing with them in this code. White space is not significant; the creators of the source file can include spaces or not, as they prefer. But we need something constant to deal with. So let's use the COMPRESS function to remove blanks. Of course, we can remove both the blanks and the asterisks with one compress function.

We will define the **%_SetPlusToMissing** macro later.

PROC FORMAT does not require that a CNTLIN= data set have an **End** variable, but if it does it must not have missing values. If we are dealing with a range of values, we need to partition our **_Value_** variable into the part before the hyphen (called **Start**) and the part after (called **End**). If there is no hyphen, we must populate the **End** variable with the contents of the **Start** variable.

READ VALUE LABEL

The **%_ReadValueLabel** macro was called by the **%_ReadCStatement** macro. The value label starts after the equal sign (=) and continues to the start of the next statement. Remember, it can span multiple input lines. So we need read, check if the new line is a new statement, and concatenate the new line to the label as long as the new line is still a continuation of the statement.

Starting after the equals sign (=), read the first of the value label. When you ask SAS to start input immediately after it finds a character string, it only searches forward for that character string. If our input pointer is already past the equals sign, then we will have to move the pointer back to the start of the line. So, to be safe, let's always back up. After reading the part of the label on the initial record of the **C** statement, we will append any line continuations to the label.

```
%Macro _ReadValueLabel ( debugging=0 ) ;

    Input @1 @=" Label ?? $char99. @@ ;

    %_ReadNextRecordCode (
        debugging=&debugging
    )
    Label = left( Label ) ;

    Do while ( _NextRecordCode_ eq "/" ) ;

        Input _MoreLabel_ ?? $char99. @@ ;

        Label =
            trim( Label ) || " " ||
            left( _MoreLabel_ )
        ;

        %_ReadNextRecordCode (
            debugging=&debugging
        )

    End ;

%Mend _ReadValueLabel ;
```

We called the same **%_ReadNextRecordCode** macro that we called in the **%_ReadOneStatement**, **%_GoToNextStatement**, and **%_ReadVStatement** macros.

GET ON DOWN

SET PLUS TO MISSING

It seems the only macro we have yet to define is the **%_SetPlusToMissing** macro which was called by the **%_ReadValueRange** macro. The **%_SetPlusToMissing** will set the variable passed in with the **var=** parameter to the appropriate missing value based on the data type as defined by the **type=** parameter.

```
%Macro _SetPlusToMissing (
    var=Start
    type=Type
    , debugging=0
) ;

    Select ( &type ) ;
        When ( "C" ) &var = " " ;
        When ( "N" ) &var = "." ;
        Otherwise put
            "ERROR: For variable " VarName
            "-- " &var= "and " &type=
            / _inFile_
        ;
    End ;

%Mend _SetPlusToMissing ;
```

Another simple macro.

Hey, we're done!

CONCLUSION

As you see, jobs are much simpler to understand, program, and modify if they are approached from a top-down perspective. Look at the general requirements for the job and, if the steps are too complicated or if you simply don't understand them yet, code a macro call and worry about the coding of the macro later. Keep working in this fashion until the tasks get simple enough to easily code.

Remember, it is much easier to read code if you can see the beginning and the end of a block of code on one page – even better a half page.

REFERENCES

"Working on a SAS Job" is a parody of Lee Dorsey's "Working in a Coal Mine."

ACKNOWLEDGMENTS

I want to thank Ian Whitlock of Westat for his continual support and encouragement in my career growth. He was an inspiration before I met him, and has proven to be a wonderful mentor and friend since.

I also want to thank Dianne Rhodes of Westat, who directed my focus toward more career-enhancing facilities such as SAS Users Groups and the SAS-L list server when we both worked elsewhere.

Finally, I want to thank all the wonderful and insightful contributors to SAS-L for their selfless contributions. They have proven to be my most valuable aid as I learn how to be a SAS programmer.

SAS is a registered trademark of SAS Institute Inc. in the

USA and other countries. ® indicates USA registration.

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

CONTACT INFORMATION

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

Edward Heaton

Westat

1650 Research Boulevard

Rockville, MD 20850

Work Phone: (301) 610-4818

Fax: (301) 294-3992

Email: EdwardHeaton@Westat.com