

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 we can use macros to approximate a top-down structure. This can lead to programs that are more versatile, more robust, and generally easier to develop. In this paper, we will review a SAS macro that reads elements from a difficult-to-read ASCII text file. Along the way, we will look at the program structure and review the engineering decisions that allow us to develop the code in an organized and robust manner.

This paper demonstrates 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 easier to understand, develop, debug, and modify. This paper demonstrates top-down programming while creating a utility to facilitate the creation of value-label formats.

Top-down programming is the mainstay of traditional procedural languages. In the top-down model, design begins by specifying complex pieces and then dividing them into successively smaller pieces until the components are specific enough to code.

The technique for writing a program using top-down methods is to write a main procedure that names all the major functions it needs. Then, the programming team looks at the requirements of each of those functions and repeats the process. We eventually call sub-routines that perform actions so simple we can code them easily and concisely. Once we have written all the various sub-routines, the program is ready to test.

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

Suppose we have a text file that was created for use in another application and that can be a useful first-step in creating value labels. In this paper, we call the text file a *Source File*. The macro reads a *Source File* and writes a SAS dataset. Of course, that dataset needs to contain all the information necessary to create value-label formats. While we develop this macro, we want to write code that is easily adaptable to other stages of our survey processing cycle. We do 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 must 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. We will focus on two statement types.
 - **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.

- **C** Code – The code statement defines the values of the variable defined in the preceding **V** statement. **A +** indicates a "missing" value.

C [*****] *value = text*

E.g.:

C ++ = inapplicable
C 01-70 = number of years
C 98 = don't know
C 99 = not ascertained

Other statement types include **P**, **H**, **Q**, **R**, **S**, and **A**. This macro works with the variable and code statements only. The *Source File* might look something like *Example 1*.

Example 1: The Source File

```
P RECLEN 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

Our macro creates a SAS dataset that we can write to an MS Access® database. Our study managers or research assistants edit the **Label** field in that database. When they are finished updating the labels, we use **Proc datasets** to attach the labels to the variables. The code to write the SAS data to an existing MS Access database looks something like *Example 2*. The **%createValueLabelsTable()** macro is stored in the `\macros\` sub-folder. The *Source File* (**SourceFile.txt**) is in the same directory as this job. We are creating a SAS dataset named **ValueLabels** and an MS Access table named **ValueLabels**.

Example 2: Creating the MS Access Table

```
Options sasAutos=( '\macros\' %sysFunc( getOption( sasAutos ) ) );
LibName mdbLib '\EasyLabels.mdb' ;
FileName srcFile '\SourceFile.txt' ;
%createValueLabelsTable( inFile=srcFile , cntlOut=ValueLabels )
FileName srcFile clear ;
/* Write data to MS Access. */
Data mdbLib.ValueLabels ;
  Set ValueLabels( rename=( Label=LongLabel ) ) ;
  Length Label $40 ;
  Label = LongLabel ;
Run ;
LibName mdbLib clear ;
```

The **%createValueLabelsTable()** macro reads the *Source File* and writes the SAS dataset. Then we write 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 label from the code statement to facilitate editing by the study manager. After the study manager has reviewed and updated the **Label** field in the MS Access table, SAS programmers will read the data to create value label formats and attach those value label formats to our SAS dataset with code similar to *Example 3*. Use the **getOption()** function – wrapped in the **%sysFunc()** macro function – to make sure that all standard SAS autocall libraries are included.

Example 3: Using the Modified MS Access Table to Create and Use Value-Label Formats

```
LibName project '\Library\';
Options fmtSearch=(project);
LibName mdbLib '\EasyLabels.mdb';
/* Create the value-label formats and write a report. */
Proc format
  library=project
  cntlin=mdbLib.ValueLabels( drop= VarName LongLabel )
  fmtLib
;
Run ;
/* Create a variable-format list as a macro variable. */
Proc sql noprint ;
  Select
    catX( ' ', VarName , trim(FmtName) || '.' )
    into :formatString separated by ' '
    from mdbLib.ValueLabels
  ;
Quit ;
LibName mdbLib clear ;
/* Apply the variable-format list to the dataset. */
Proc datasets ;
  Modify project.OurData ;
  Format &formatString ;
  Run ;
Quit ;
LibName project clear ;
```

As you can see, this job is relatively short and simple. It could, however, be developed into a macro to make the task even easier.

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. (Re. *Example 4*.)

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.

Example 4: A Template for Header Documentation

```

/*****
MACRO:

OBJECTIVE:
    This macro will ...

VALID: between program steps OR between DATA step statements OR in a DATA step
        statement OR as a BOOLEAN expression OR ...

USAGE:
    %macroName(
        debugging=
        , parm1=
        , parm2=
        , parm3=
        ...
    )

PROGRAMMER:
    Edward Heaton, SAS Senior Systems Analyst,
    Westat (An Employee-Owned Research Corporation),
    1650 Research Boulevard, RW-4541, Rockville, MD 20850-3195
    Voice: (301) 610-4818           Fax: (301) 294-3879
    mailto:EdHeaton@Westat.com     http://www.Westat.com

DETAILS:

PARAMETERS:
    debugging= is set to 1 to evoke debugging options. (The default is 0.)
    parm1=
    parm2=
    parm3=
    ...

INPUT:

OUTPUT:

TESTING:

STORAGE: Specify the fully-qualified file name for this macro.

AUDIT TRAIL:
    yyymmdd EH Developed macro to ...
*****/
%macro mName( debugging=0 ) ;
/* SAS code and calls to other macros */
%mEnd mName ;
/*****/

```

Top-Down Programming – At the Top

Let's define the steps to create a dataset of value labels. The `%createValueLabelsTable()` macro exists in a file called `createValueLabelsTable.sas`. Let's define a *top-level* macro as a macro that exists in a file of the same name. (Re. Example 5.)

Example 5: %createValueLabelsTable() is the Top-Level Macro.

```
%macro createValueLabelsTable(  
    debugging=0  
    , inFile=srcFile  
    , cntlOut=ValueLabels  
);  
/* Create a table that has all the data that we need. */  
%mend createValueLabelsTable ;
```

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 add a **Data** step to create the table. Note the macro `%if` statement embedded in the **Data** statement in Example 6. I typically start all temporary variable names – and only 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.

Example 6: The Data Step in the %createValueLabelsTable() Macro

```
Data &cntlOut(  
    %if not &debugging %then drop=_: ;  
);  
    %_defineVariables()  
    Retain VarName FmtName Type ;  
    InFile &inFile truncOver end=eof ;  
    %_readOneStatement()  
    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 dataset 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. (Re. Example 7.) We can print the whole dataset; it shouldn't be too long. But let's print only the first 40 characters of **Label** because that value might be quite long.

Example 7: Example of the &debugging Parameter

```
%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 less likely 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 initially 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 make the **Data** step short enough to see in its entirety. It will also remove some of the detail from this top-level macro that would just muddy the waters. (Re. Example 8.)

Example 8: Macro to Declare Variables

```
%macro _defineVariables() ;
  Attrib
    VarName  length=$030  label='Variable Name'
    FmtName  length=$032  label='Format Name'
    Start    length=$016  label='Starting Value for Format'
    End      length=$016  label='Ending Value for Format'
    Label    length=$256  label='Format Value Label'
    Type     length=$001  label='Type of Format'
;
%mEnd _defineVariables ;
```

Why did we set the length of the variable name to 30 characters? SAS allows 32 characters. The problem is with the format name. 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 30-character variable name will become a 32-character format name. It will be the responsibility of the user to restrict the variable names to a maximum of 30 characters.

Read One Statement

Now we read the values for the needed variables and output the dataset. 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 use the **Select** statement to set our course. (Re. Example 9.)

We need only process the **V** and **C** statements; a **V** statement provides a variable's name and its data type; and a **C** statement provides a value label. The other statements can be skipped. However, we want to keep the select statement very general because we might 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.

Example 9: Macro to Read One Statement from the Source File

```
%macro _readOneStatement() ;
  Input @1 _statementType : $1. @@ ;
  Select (_statementType) ;
    When ('P') do ; %_readPStatement()      End ;
    When ('H') do ; %_readHStatement()      End ;
    When ('V') do ; %_readVStatement()      End ;
    When ('Q') do ; %_readQStatement()      End ;
    When ('C') do ; %_readCStatement()      End ;
    When ('R') do ; %_readRStatement()      End ;
    When ('S') do ; %_readSStatement()      End ;
    When ('A') do ; %_readAStatement()      End ;
    When ('/') do ; %_virguleErrMsg()       End ;
    When (' ') do ; %_readNextRecordCode()  End ;
    Otherwise do ; %_genErrMsg()           End ;
  End ;
%mend _readOneStatement ;
```

This is long, but it does fit my half-page criterion. This macro calls a lot of other macros that we have yet to define.

Three notes:

1. The colon (:) format modifier in the **Input** statement 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 needs to explicitly tell SAS when to output an observation. If we come to an empty record, we 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.

Notice how the code in the lines is aligned. This simply makes it easier for me to catch my typos. I can scan down and quickly see when something is not the same.

Level 3: Supporting the Supporting Macros

Okay, the **%_readOneStatement()** macro was easy because it only decides which other macros to call. Now we have to define these macros.

Read a 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. (Re. Example 10.)

Example 10: Macro to Bypass a Parameter Statement

```
%macro _readPStatement() ;
  %_goToNextStatement()
%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 macros similar to the **%_readPStatement()** 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 a 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 30 characters or fewer for this macro. The macro to read this statement is shown in Example 11.

Example 11: Macro to Read a Variable Statement

```
%macro _readVStatement() ;
  Input VarName $ _length_ $ Type $ @@ ;
  %_readNextRecordCode()
  FmtName = catT( 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 dataset. 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 a 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
```

1. The cee identifies the statement as a code statement.
2. The one or more asterisks link to **S** statements. Our macro skips **S** statements, but we do need to write the code to skip over the asterisks.
3. The value can be a single value or a range of values. Character values are not quoted. Users code plusses (+) to represent missing values.
4. An equals sign separates the value – or range of values – from the value label.
5. The value label follows the equals sign and can span lines of the *Source File* with virgules as line continuation indicators.

We only need two things from the **C** statements – value ranges and labels.

Example 12: Macro to Read a Code Statement

```
%macro _readCStatement() ;
  %_readValueRange()
  %_readValueLabel()
%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. (Re. Example 12.)

Read the 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. (Re. Example 13.)

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.

Example 13: Macro to Read the Record Code for the Next Line in the Source File

```
%macro _readNextRecordCode() ;
  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.

Print an Error Message When the Record Type is a Virgule

The `%_virguleErrorMessage()` macro (Re. Example 14.) 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.

Example 14: Macro to Report an Error for an Out-of-Place Line Continuation

```
%macro _virguleErrorMessage() ;
  Put
    'ERROR: / records should never be accessed from the'
  +1 '%_readOneStatement() macro!'
  / _inFile_
  ;
  %_goToNextStatement()
%mEnd _virguleErrorMessage ;
```

Generate an Error Message for Invalid Statement Types

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. (Re. Example 15.)

Example 15: Macro to Warn of an Invalid Statement Type

```
%macro _genErrorMessage() ;
  Put
    'WARNING:' +1 _statementType= 'is not valid.'
  / _inFile_
  ;
  %_goToNextStatement()
%mEnd _genErrorMessage ;
```

Deeper and Deeper

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

Go To Next Statement

The `%_goToNextStatement()` macro was called by several macros – `%_readPStatement()`, `%_readHStatement()`, etc. It's defined in Example 16.

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.

Example 16: Macro to Advance to the Next Source File Statement

```
%macro _goToNextStatement() ;
  %_readNextRecordCode()
  Do while ( _nextRecordCode eq '/' ) ;
    %_readNextRecordCode()
  End ;
%mEnd _goToNextStatement ;
```

Again, this is very simple code. We have already defined the `%_readNextRecordCode()` macro.

Read the 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 holds the value range 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**.

Example 17: Macro to Parse a Value Range

```
%macro _readValueRange( rangeVar=_value ) ;
  Input _value $char99. @@ ;
  /* Get the string before the equal sign and then compress out the asterisks and
  the spaces. */
  &rangeVar = compress( scan( &rangeVar , 1 , '=' ) , ' *' ) ;
  If index( _value , '+' )
    then do ;
      %_SetPlusToMissing( var=&rangeVar , type=Type )
    End ;
  Else do ;
    Start = scan( &rangeVar , 1 , '-' ) ;
    /* If the starting value is a negative number, we need to go back and
    add the negative sign. */
    If ( substr( &rangeVar , 1 , 1 ) eq '-' ) then Start = '-' || Start ;
    End = substr( &rangeVar , length(Start) + 2 ) ;
    If missing(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 equals 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 equals sign.

The **compress()** function removes characters specified in the second argument from the first argument. So **compress(_value , '*')** removes 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 call of the **compress()** function.

We will define the **%_setPlusToMissing()** macro later.

PROC FORMAT does not require that a **cntlIn=** dataset 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. We want to allow negative numbers. Fortunately, the **scan()** function treats consecutive delimiters as one. If the first value is negative, we will need to go back and add the negative sign. We will use the **substr()** function to return the part of **_value** after the **Start** value.

Read the Value Label

The **%_readValueLabel()** macro was called by the **%_readCStatement()** macro. The value label starts after the equals sign (=) and continues to the start of the next statement. Remember, it can span multiple input lines. So, we need to 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. (Re. Example 18.)

Starting after the equals sign (=), read the first part 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 append any line continuations to the label.

Example 18: Macro to Read the Label from a C Statement

```
%macro _readValueLabel() ;
  Input @1 @'=' Label ?? $char99. @@ ;
  %_readNextRecordCode()
  Label = left(Label) ;
  Do while ( _nextRecordCode eq '/' ) ;
    Input _moreLabel ?? $char99. @@ ;
    Label = catX( ' ', Label , _moreLabel ) ;
    %_readNextRecordCode()
  End ;
%mEnd _readValueLabel ;
```

We called the same **%_readNextRecordCode()** macro that we called in the **%_readOneStatement()**, **%_goToNextStatement()**, and **%_readVStatement()** macros.

Get on Down

Now, we need to define macros that were called by the macros we just coded.

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()** macro will set the variable passed in through the **var=** parameter to the appropriate missing value based on the data type as defined by the **type=** parameter.

Example 19: Macro to Convert Plus Signs to a Missing Value

```
%macro _setPlusToMissing( var=Start , type=Type ) ;
  Select (&type) ;
    When ('C') &var = ' ' ;
    When ('N') &var = '.' ;
    Otherwise put
      'ERROR: For variable' +1 VarName '---' +1 &var= 'and' +1 &type=
      / _inFile_
    ;
  End ;
%mEnd _setPlusToMissing ;
```

Another simple macro.

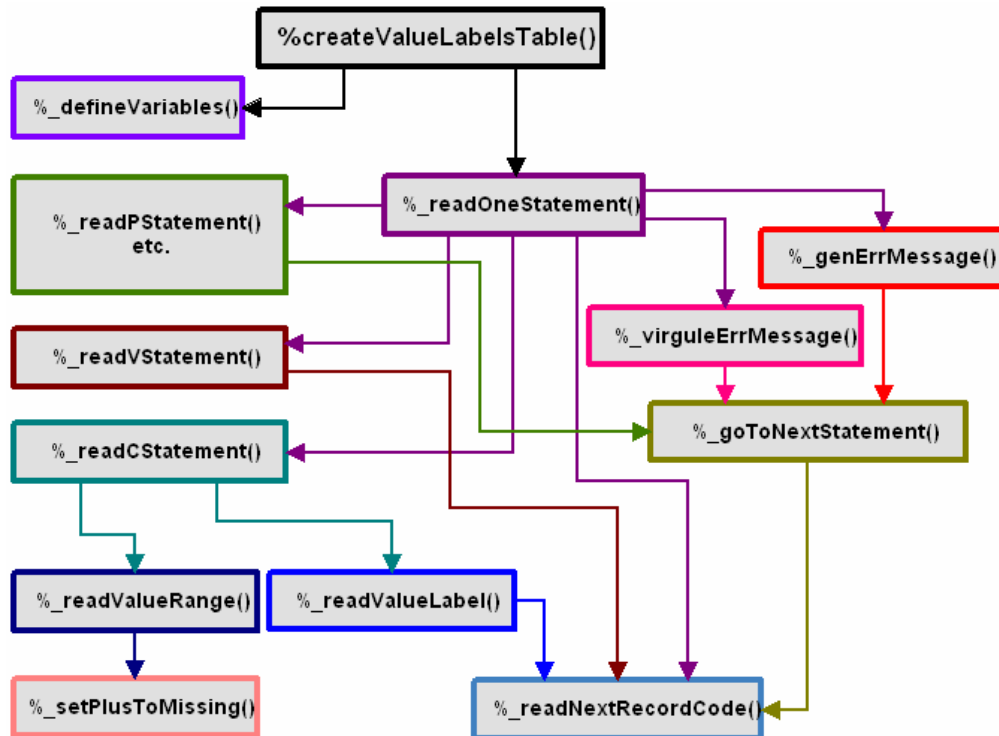
Hey, we're done!

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.

Here's a schematic of how these macros work together.



Acknowledgments

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.

Disclaimer

The content of this paper is the work of the author and does not necessarily represent the opinions, recommendations, or practices of Westat.

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

Appendix: The %createValueLabelsTable() Macro

```
/******  
MACRO: createValueLabelsTable  
  
OBJECTIVE:  
    This macro will read a COED Source File and create a table that can be used  
    to create value-label formats.  
  
VALID: between program steps  
  
USAGE:  
    %createValueLabelsTable(  
        debugging=  
        , inFile=  
        , cntlOut=  
    )  
  
PROGRAMMER:  
    Edward Heaton, SAS Senior Systems Analyst,  
    Westat (An Employee-Owned Research Corporation),  
    1650 Research Boulevard, RW-4541, Rockville, MD 20850-3195  
    Voice: (301) 610-4818           Fax: (301) 294-3879  
    mailto:EdHeaton@Westat.com     http://www.Westat.com  
  
DETAILS:  
  
PARAMETERS:  
    debugging= is set to 1 to evoke debugging options. (The default is 0.)  
    inFile=    is either a fileRef or a quoted file specification to a valid  
               COED Source File.  
    cntlOut=   is a table that can be used to create a value-label format via  
               the cntlIn= option in a Proc format statement.  
  
INPUT: This macro will read a COED Source File.  
  
OUTPUT:  
    The output table will need to be modified so that the Label variable is the  
    desired length. In doing so, the values in that field might need to be  
    manually adjusted.  
  
TESTING: No test plan has been developed.  
  
STORAGE: \\path\autocall\createValueLabelsTable.sas  
  
AUDIT TRAIL:  
    20001005 EH Developed macro to demonstrate at WesSUG.  
    20010425 EH Modified macro for a paper at SSU 2001.  
    20060527 EH Modified macro for a paper at SESUG 2006; allowed the scope of  
               &debugging to trickle down to the subordinate macros.
```

```

*****/
%macro createValueLabelsTable(
    debugging=0
    , inFile=srcFile
    , cntlOut=ValueLabels
) ;
/* Create a table that has all the data that we need. */
Data &cntlOut (
    %if not &debugging %then drop=_: ;
) ;
    %defineVariables(
        Retain VarName FmtName Type ;
        InFile &inFile truncOver end=eof ;
        %readOneStatement( cntlOut=&cntlOut )
        If ( upCase( _statementType ) eq 'C' ) then output &cntlOut ;
    Run ;

    %if &debugging %then %do ;
        Title4 "&cntlOut" ;
        Proc print data=&cntlOut ;
            Format Label $40. ;
        Run ;
        Title4 ;
    %End ;
%mEnd createValueLabelsTable ;
/*-----*/
%macro _defineVariables() ;
    Attrib
        VarName    length=$030    label='Variable Name'
        FmtName    length=$032    label='Format Name'
        Start      length=$016    label='Starting Value for Format'
        End        length=$016    label='Ending Value for Format'
        Label      length=$256    label='Format Value Label'
        Type       length=$001    label='Type of Format'
    ;
%mEnd _defineVariables ;
/*-----*/
%macro _readOneStatement() ;
    Input @1 _statementType : $1. @@ ;
    Select ( _statementType ) ;
        When ( 'P' ) do ; %_readPStatement()      End ;
        When ( 'H' ) do ; %_readHStatement()      End ;
        When ( 'V' ) do ; %_readVStatement()      End ;
        When ( 'Q' ) do ; %_readQStatement()      End ;
        When ( 'C' ) do ; %_readCStatement()      End ;
        When ( 'R' ) do ; %_readRStatement()      End ;
        When ( 'S' ) do ; %_readSStatement()      End ;
        When ( 'A' ) do ; %_readAStatement()      End ;
        When ( '/' ) do ; %_virguleErrMessage()    End ;
        When ( '20'x ) do ; %_readNextRecordCode() End ;

```

```

        Otherwise      do ; %_genErrorMessage()      End ;
    End ;
%mEnd _readOneStatement ;
/*-----*/
%macro _readPStatement() ;
    %_goToNextStatement()
%mEnd _readPStatement ;
/*-----*/
%macro _readHStatement() ;
    %_goToNextStatement()
%mEnd _readHStatement ;
/*-----*/
%macro _readQStatement() ;
    %_goToNextStatement()
%mEnd _readQStatement ;
/*-----*/
%macro _readRStatement() ;
    %_goToNextStatement()
%mEnd _readRStatement ;
/*-----*/
%macro _readSStatement() ;
    %_goToNextStatement()
%mEnd _readSStatement ;
/*-----*/
%macro _readAStatement() ;
    %_goToNextStatement()
%mEnd _readAStatement ;
/*-----*/
%macro _readVStatement() ;
    Input VarName $ _length_ $ Type $ @@ ;
    %_readNextRecordCode()
    FmtName = catT( VarName , 'F' ) ;
    Select (Type) ;
        When ('G') delete ;
        When ('N') ;
        When ('A') do ;
            Type = 'C' ;
            FmtName = '$' || FmtName ;
        End ;
    End ;
%mEnd _readVStatement ;
/*-----*/
%macro _readCStatement() ;
    %_readValueRange()
    %_readValueLabel()
%mEnd _readCStatement ;
/*-----*/
%macro _readNextRecordCode() ;
    If EOF
        then input ;

```

```

        Else input / @1 _nextRecordCode : $1. @@ ;
%macro _readNextRecordCode ;
/*-----*/
%macro _virguleErrMsg() ;
    Put
        'ERROR: / records should never be accessed from the'
    +1 '%_readOneStatement() macro!'
        / _inFile_
    ;
    %_goToNextStatement()
%macro _virguleErrMsg ;
/*-----*/
%macro _genErrMsg() ;
    Put
        'WARNING:' +1 _statementType= 'is not valid.'
        / _inFile_
    ;
    %_goToNextStatement()
%macro _genErrMsg ;
/*-----*/
%macro _goToNextStatement() ;
    %_readNextRecordCode()
    Do while ( _nextRecordCode eq '/' ) ;
        %_readNextRecordCode()
    End ;
%macro _goToNextStatement ;
/*-----*/
%macro _readValueRange() ;
    Input _value $char99. @@ ;
/* Get the string before the equal sign and then compress out the asterisks and
the spaces. */
    _value = compress( scan( _value , 1 , '=' ) , ' *' ) ;
    If index( _value , '+' )
        then do ;
            %_SetPlusToMissing( var=_value , type=Type )
        End ;
    Else do ;
        Start = scan( _value , 1 , '-' ) ;
        /* If the starting value is a negative number, we need to go back and
add the negative sign. */
        If ( substr( _value , 1 , 1 ) eq '-' ) then Start = '-' || Start ;
        End = substr( _value , length(Start) + 2 ) ;
        If missing(End) then End = Start ;
    End ;
%macro _readValueRange ;
/*-----*/
%macro _readValueLabel() ;
    Input @1 @'=' Label ?? $char99. @@ ;
    %_readNextRecordCode()
    Label = left(Label) ;

```

```

Do while ( _nextRecordCode eq '/' ) ;
  Input _moreLabel ?? $char99. @@ ;
  Label = catX( '20'x , Label , _moreLabel ) ;
  %_readNextRecordCode()
End ;
%mEnd _readValueLabel ;
/*-----*/
%macro _setPlusToMissing( var=Start , type=Type ) ;
  Select (&type) ;
    When ('C') &var = '20'x ;
    When ('N') &var = '.' ;
    Otherwise put
      'ERROR: For variable' +1 VarName '--' +1 &var= 'and' +1 &type=
      / _inFile_
    ;
  End ;
%mEnd _setPlusToMissing ;
/*****/

```