Macros

SAS has pre-defined *macro functions* that provide functionality which does not exist otherwise in its programming language. Also, users can create their own macro functions to automate particular tasks that work in a similar manner as functions/procedures in other programming languages. This latter case is often used by SAS users for research purposes.

All programs and data sets used for these notes are available from my course website. New files that we have not used before are binary.sas, cereal_macro.sas, pre-defined.sas, and normal_sim.sas.

Pre-defined

All macro functions start with the `%` percentage symbol. The most basic macro function is `%let` which allows one to declare a constant value to be used throughout a program. Below is a simple example to create two *macro variables* using the `%let` function:

```
title1 "Chris Bilder, STAT 850";

%let newtitle = My title;
%let x1 = year;

*Current day/time;
data time;
  time = hour(time());
  minute = minute(time());
  second = second(time());
  month = month(today());
  day = day(today());
  year = year(today());
run;
```

```
title1 "&newtitle";
title2 "What time is it?";
proc print data=time;
run;

title1 "&newtitle";
title2 "What year is it?";
proc print data=time;
  var &x1;
run;
```

My title
What time is it?

Obs	time	minute	second	month	day	year
1	16	14	12.6580	7	7	2016

My title
What year is it?

Obs	year
1	2016

The syntax to access a value of the macro variable is to put an & ampersand symbol prior to the variable's name.

I have used `%let` a number of times in my research when I needed to set a particular value to be constant throughout a long SAS program. For example, my control_2mrcv.final.sas program at `http://www.chrisbilder.com/mrcv/Comm` shows a number of instances where I used this macro function in a program which is meant to disseminate my research to others. Within the program, you will see the setting of a number of constants, including 1) a convergence criteria, 2) the number of resamples to take for bootstrapping, and 3) the confidence level to use with confidence intervals. These constants are made easily available for users of my program to change if desired.

The `%put` function can write specific information to the Log

window:

```
%let x2 = 1;
%put &x2;
%put Go Big Red!;

%let x3 = 1 + &x2;
%put &x3;
%put %eval(&x3);
%put %sysevalf(&x3 + 0.1);


655     %let x2 = 1;
656     %put &x2;
1
657     %put Go Big Red!;
Go Big Red!

658     %let x3 = 1 + &x2;
659     %put &x3;
1 + 1
660     %put %eval(&x3);
2
661     %sysevalf(&x3 + 0.1);
2.1
```

Comments:

- Simple mathematics can be performed with macro variables as long as the `%eval()` macro function is used. The `%syseval()` function performs the same operations as `%eval()` but it allows for non-integer values.
- My main use of `%put` has been for program debugging purposes.

The `%include` function allows one to include a separate program in another program. For example, below is how I can include code from the cereal.sas program in my current program.

```
%include "C:\chris\unl\cereal.sas";
```

Simply, all of the code in cereal.sas is run when this line of code is submitted. I have frequently used the `%include` function in my own research to separate out particular parts of one overall program. For example, you will see that my control_2mrcv.final.sas uses `%include` three times.

User-created

Syntax

A user-created macro function uses the following syntax:

```
%macro myfunc(var1, var2);

  <put code here>

%mend myfunc;
```

where `myfunc` was a simple name that I chose here for the macro function's name. The `var1` and `var2` represent macro variables (other names can be used), and they can be referred to within the code section by `&var1` and `&var2`. The code section can include datasteps, procedure calls, and other code.

To run the macro function, the entire macro function code must be executed first so that SAS recognizes it as a macro. Next, call the macro with

```
%myfunc(var1 value, var2 value);
```

where actual values are given for `var1` and `var2` in the above code.

Example #1

A macro function that I have used frequently in my research finds the current day and time when the code is run. Through running this macro in a number of places within a program, it allowed me

to determine how long a set of code took to complete. Below is my `%time` macro:

```
%macro time(title);

  data time;
    time = hour(time());
    minute = minute(time());
    second = second(time());
    month = month(today());
    day = day(today());
    year = year(today());
  run;

  title2 "&title";
  proc print data=time;
  run;

%mend time;


%time(This part of my program ended at:)
```

Chris Bilder, STAT 850
This part of my program ended at:

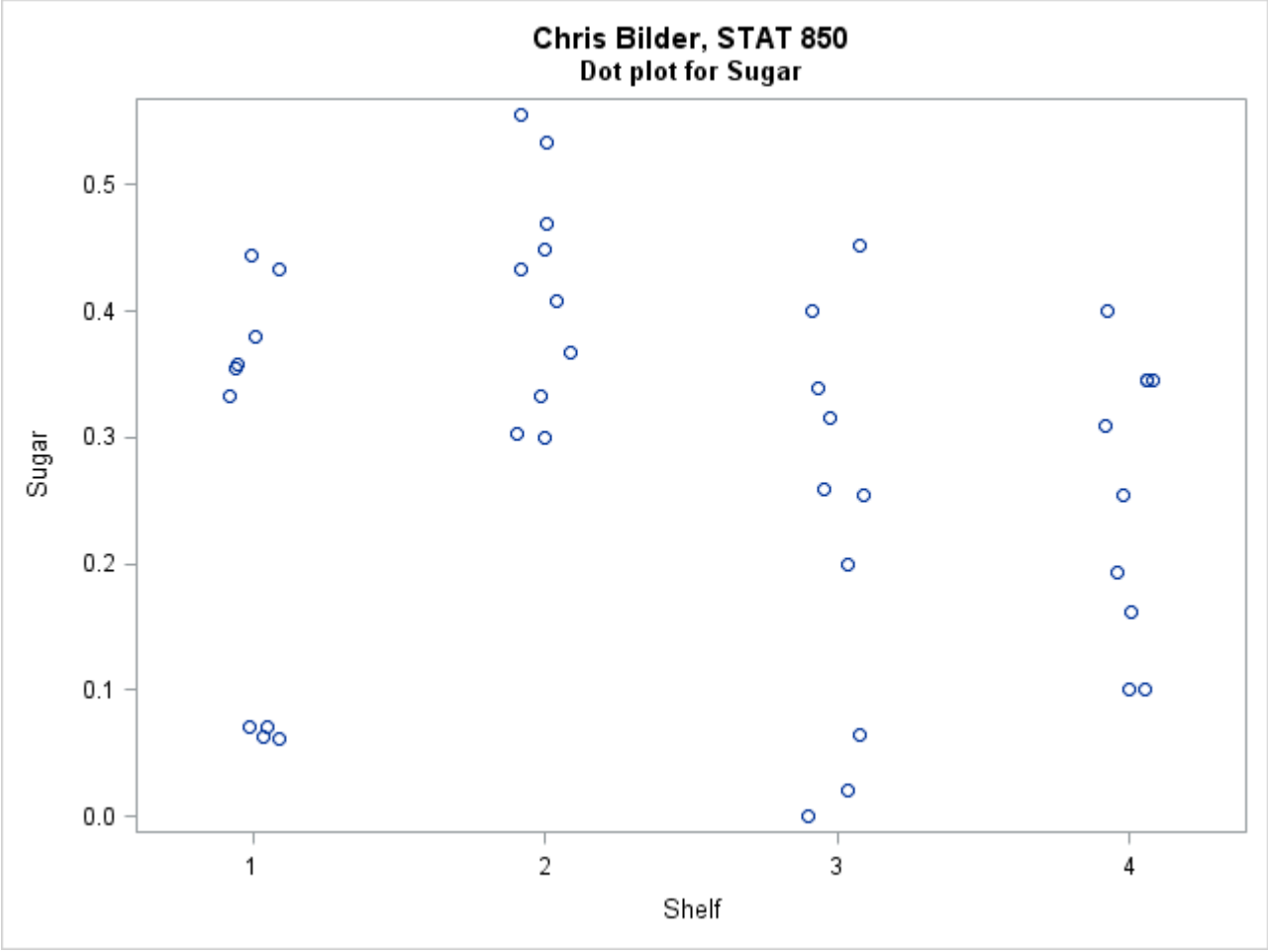
Obs	time	minute	second	month	day	year
1	22	43	27.2120	7	7	2016

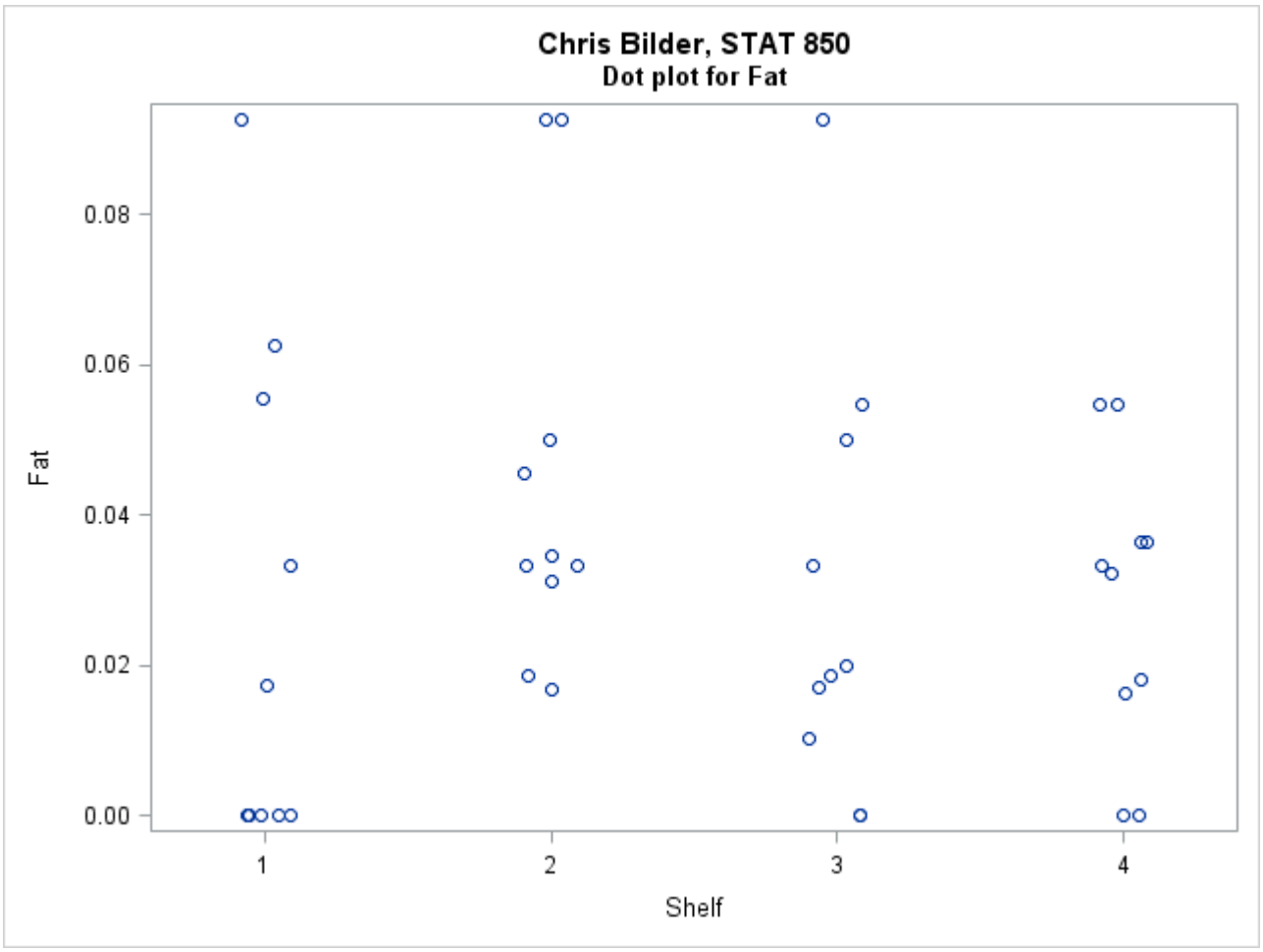
The `title` macro variable resolves to "This part of my program ended at:". In actual application, I may make this title more descriptive.

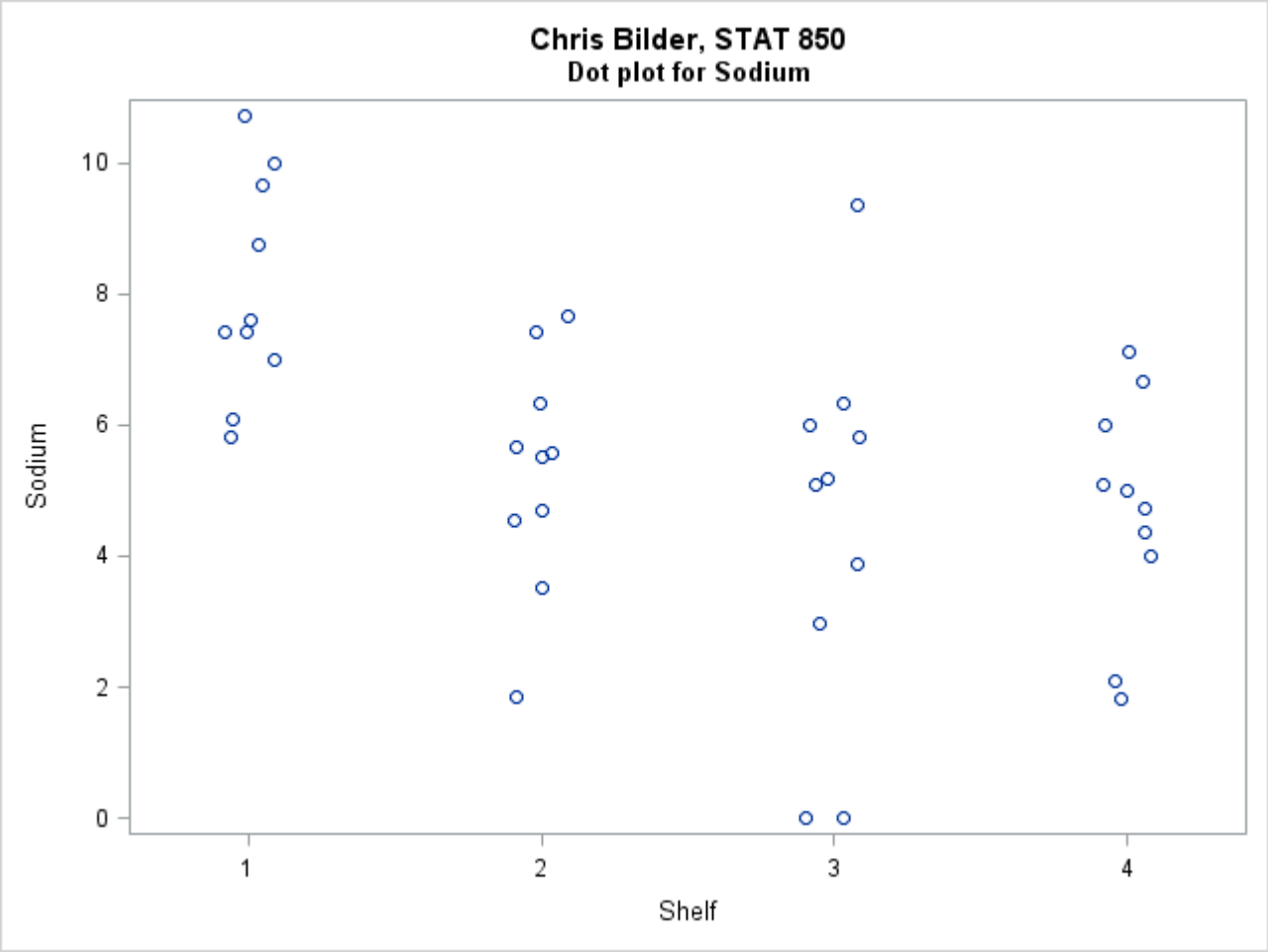
Example #2

A common use of macros is to produce multiple plots that differ only on a few aspects. For example, below is a simple macro used to produce dot plots with the cereal data set. The y-axis is the main item that differs for each plot.

```
*set1 contains the cereal data;

%macro dotplot(var1);

  title2 "Dot plot for &var1";
  proc sgplot data=set1;
    scatter x=shelf y=&var1 / jitter jitterwidth=0.2;
    yaxis label="&var1";
    xaxis values=(1 to 4 by 1);
  run;

%mend dotplot;

%dotplot(Sugar);
%dotplot(Fat);
%dotplot(Sodium);
```




Chris Bilder, STAT 850
Dot plot for Fat
Fat
0.08
0.06
0.04
0.02
0.00
1
2
3
4
Shelf



Options

Below is the Log window after running Example #2.

```
861     %macro dotplot(var1);
862
863       title2 "Dot plot for &var1";
864       proc sgplot data=set1;
865         scatter x=shelf y=&var1 / jitter jitterwidth=0.2;
866         yaxis label="&var1";
867         xaxis values=(1 to 4 by 1);
868       run;
869
870     %mend dotplot;
871
872     %dotplot(Sugar);

NOTE: PROCEDURE SGPLOT used (Total process time):
      real time           0.18 seconds
```

```
      cpu time            0.06 seconds

NOTE: There were 40 observations read from the data set
      WORK.SET1.

873    %dotplot(Fat);

NOTE: PROCEDURE SGPLOT used (Total process time):
      real time           0.18 seconds
      cpu time            0.06 seconds

NOTE: There were 40 observations read from the data set
      WORK.SET1.

874    %dotplot(Sodium);

NOTE: PROCEDURE SGPLOT used (Total process time):
      real time           0.37 seconds
      cpu time            0.07 seconds

NOTE: There were 40 observations read from the data set
      WORK.SET1.
```

While in this simple setting it is not too difficult to figure out which macro is being executed and what the one macro variable resolves to, this will not be true in other settings when you have

- macros that invoke other macros
- many macro variables
- macros that are hundreds of lines long

For this reason, I include the following line of code at the top of my programs whenever I use macros:

```
options mprint symbolgen mlogic;
```

These options instruct SAS to print additional information to the Log window whenever macros are being used. Below is a description of each option:

- `mprint`: Traces the SAS statements generated by macro execution
- `symbolgen`: Provides the value for a macro variable
- `mlogic`: Traces the flow of execution in the macro (e.g., comments on when a new macro is beginning to be executed)

Each new line of information in the Log window will begin with `mprint`, `symbologen`, or `mlogic` to explain its purpose.

Below is part of my Log window from running Example #2 with the three options.

```
954    %macro dotplot(var1);
955
956      title2 "Dot plot for &var1";
957      proc sgplot data=set1;
958        scatter x=shelf y=&var1 / jitter jitterwidth=0.2;
959        yaxis label="&var1";
960        xaxis values=(1 to 4 by 1);
961      run;
962
963    %mend dotplot;
964
965    %dotplot(Sugar);
MLOGIC(DOTPLOT):  Beginning execution.
MLOGIC(DOTPLOT):  Parameter VAR1 has value Sugar
SYMBOLGEN:  Macro variable VAR1 resolves to Sugar
MPRINT(DOTPLOT):   title2 "Dot plot for Sugar";
MPRINT(DOTPLOT):   proc sgplot data=set1;
SYMBOLGEN:  Macro variable VAR1 resolves to Sugar
MPRINT(DOTPLOT):   scatter x=shelf y=Sugar / jitter
  jitterwidth=0.2;
SYMBOLGEN:  Macro variable VAR1 resolves to Sugar
MPRINT(DOTPLOT):   yaxis label="Sugar";
MPRINT(DOTPLOT):   xaxis values=(1 to 4 by 1);
MPRINT(DOTPLOT):   run;

NOTE: PROCEDURE SGPLOT used (Total process time):
      real time           0.21 seconds
      cpu time            0.07 seconds
```

```
NOTE: There were 40 observations read from the data set
      WORK.SET1.

MLOGIC(DOTPLOT):  Ending execution.
966    %dotplot(Fat);
MLOGIC(DOTPLOT):  Beginning execution.
MLOGIC(DOTPLOT):  Parameter VAR1 has value Fat
SYMBOLGEN:  Macro variable VAR1 resolves to Fat
MPRINT(DOTPLOT):   title2 "Dot plot for Fat";
MPRINT(DOTPLOT):   proc sgplot data=set1;
SYMBOLGEN:  Macro variable VAR1 resolves to Fat
MPRINT(DOTPLOT):   scatter x=shelf y=Fat / jitter
  jitterwidth=0.2;
SYMBOLGEN:  Macro variable VAR1 resolves to Fat
MPRINT(DOTPLOT):   yaxis label="Fat";
MPRINT(DOTPLOT):   xaxis values=(1 to 4 by 1);
MPRINT(DOTPLOT):   run;

NOTE: PROCEDURE SGPLOT used (Total process time):
      real time           0.19 seconds
      cpu time            0.04 seconds

NOTE: There were 40 observations read from the data set
      WORK.SET1.

MLOGIC(DOTPLOT):  Ending execution.
967    %dotplot(Sodium);
MLOGIC(DOTPLOT):  Beginning execution.
MLOGIC(DOTPLOT):  Parameter VAR1 has value Sodium
SYMBOLGEN:  Macro variable VAR1 resolves to Sodium
MPRINT(DOTPLOT):   title2 "Dot plot for Sodium";
MPRINT(DOTPLOT):   proc sgplot data=set1;
SYMBOLGEN:  Macro variable VAR1 resolves to Sodium
MPRINT(DOTPLOT):   scatter x=shelf y=Sodium / jitter
  jitterwidth=0.2;
SYMBOLGEN:  Macro variable VAR1 resolves to Sodium
MPRINT(DOTPLOT):   yaxis label="Sodium";
MPRINT(DOTPLOT):   xaxis values=(1 to 4 by 1);
MPRINT(DOTPLOT):   run;
```

```
NOTE: PROCEDURE SGPLOT used (Total process time):
      real time           0.17 seconds
      cpu time            0.03 seconds

NOTE: There were 40 observations read from the data set
      WORK.SET1.

MLOGIC(DOTPLOT):  Ending execution.
```

Loops and conditional execution

There are pre-defined macro functions available for loops and conditional execution *within* user-created macro functions. I have used these a lot with my research (e.g., Monte Carlo simulations) and to have finer control over when a particular set of code is executed.

Loops are performed using a `%do`, `%to`, and `%end` code set. Below is the standard syntax where a set of code is repeated 10 times.

```
%do i = 1 %to 10;

  <put code here>

%end;
```

The `i` in the do loop is a macro variable can be referred within it as `&i`. Some other letter or name could have been used rather than `i`.

As an example of using a do loop, suppose I would like to create 6 data sets each with 1000 observations from populations characterized by normal probability distributions. These distributions have the same mean, but the standard deviation is different for each. Below is my code and output:

```
%macro example1;
```

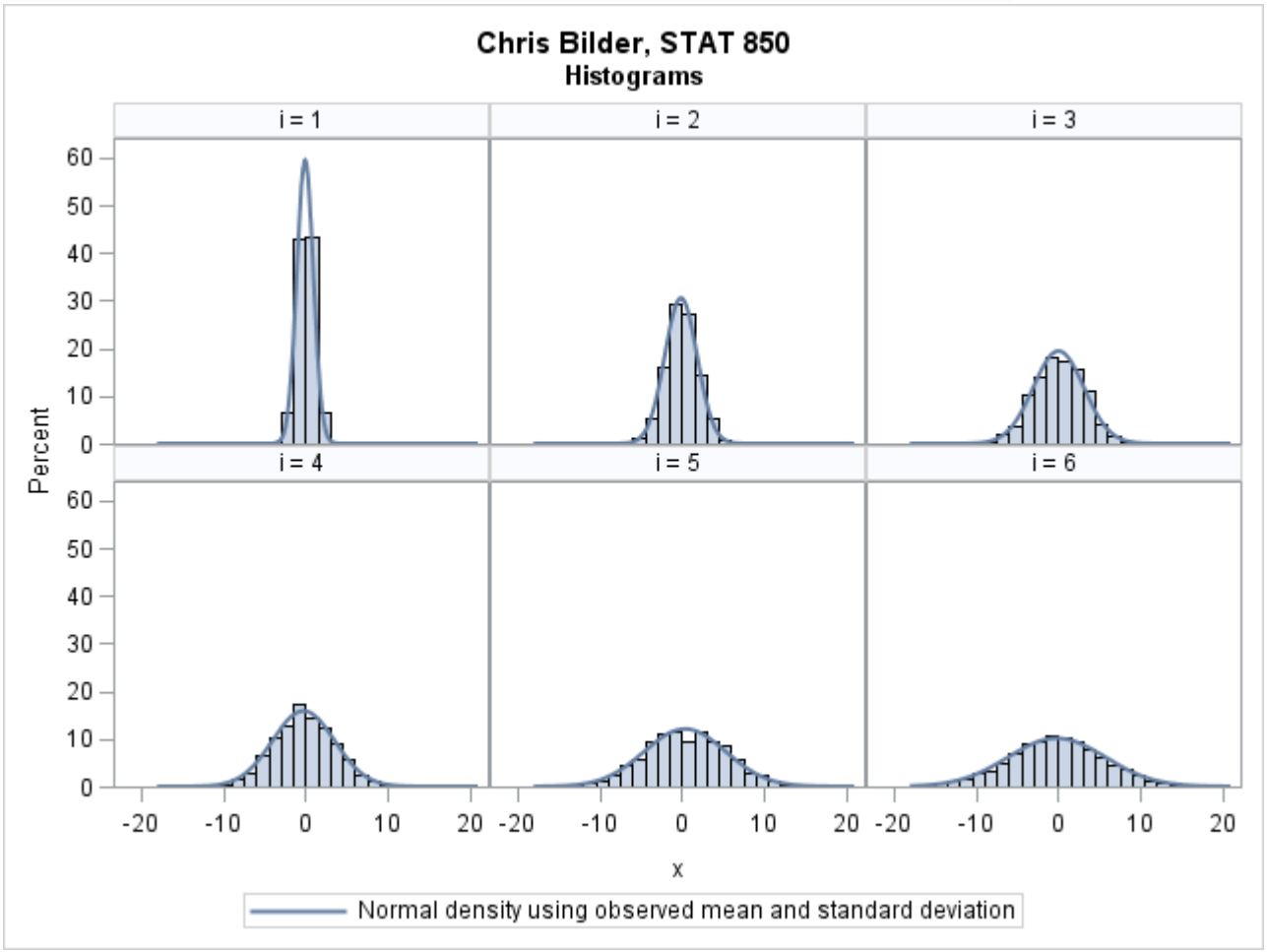
```
  *create an empty data set to save the results in;
  data save;
    set _null_;
  run;

  %do i = 1 %to 6;

    data set&i;
      call streaminit(%eval(1221 + &i));
      do j = 1 to 1000;
        x = rand("normal", 0, &i);
        i = &i;
        output;
      end;
    run;

    *adds set&i to the end of the data set;
    data save;
      set save set&i;
    run;

  %end;

%mend example1;

%example1;

title2 "Summary statistics for data simulated from a normal
   distribution";
proc means data=save mean std;
  class i;
  var x;
run;

title2 "Histograms";
proc sgpanel data=save;
  panelby i;
  histogram x;
  density x / type=normal legendlabel="Normal density using
    observed mean and standard deviation";
```

```
run;
```

Chris Bilder, STAT 850
Summary statistics for data simulated from a normal distribution

The MEANS Procedure

Analysis Variable : x			
i	N Obs	Mean	Std Dev
1	1000	-0.0136031	1.0066057
2	1000	-0.0819543	1.9584700
3	1000	0.0703464	3.0831272
4	1000	-0.2207414	3.7736468
5	1000	0.4097500	4.9606690
6	1000	-0.1643165	5.9118945



Comments:

- Six different data sets are created and then vertically concatenated. For this to work, an initial, empty data set needed to

be created. This was done by using the `_null_` statement in the first datastep of `%example1`. The concatenation occurred in the last datastep of the macro. While this strategy works well here, note that the save data set is written over multiple times. If the data set was large, this could result in poor time efficiency due to the time it takes to write the file to the hard drive. Options to avoid this potential problem involve using `proc append` or `proc datasets`.

- Notice the use of `set&i` when creating the six different data sets. This represents a convenient way to use a macro variable in combination with similarly named items. For instances where two macro variables need to be used in this format, one could use a syntax like `set&i.&k` where `k` is the second macro variable. The period is not included in how SAS recognizes the resolved name.

Conditional execution is performed using a `%if`, `%then`, and `%else` code set. Very often, a `%do` and `%end` code set needs to be included so that a set of code can be executed depending on the conditions. Below is the standard syntax.

```
%if <condition> %then %do;

  <put code here>

%end;
%else %do;

  <put code here>

%end;
```

A very simple setting where conditional execution can be helpful is when keeping track of how far along a Monte Carlo simulation is. For example, suppose one has a do loop that repeats a set of code 10,000 times. One could use conditional execution to print to the Log window when iteration 1,000, 2,000, ..., and

9,000 is reached. The `%example2` macro in my program provides a simple illustration of this type of code in the context of the last example.

Binary variable example

This example shows how SAS can be used to write its own code! I developed this example through my own correlated binary data research. In summary, binary variables with values of 0's and 1's can occur as a response (e.g., pass/fail, yes/no, ...). If more than one of these binary variables is observed on an experimental unit, the responses are likely to be correlated. In one part of my research, I needed to created all possible combinations of binary responses when there may be c of these variables. For example, when $c = 4$, there are $2^4 = 16$ possible combinations of the 0-1 values:

	Y_1	Y_2	Y_3	Y_4
1	0	0	0	0
2	0	0	0	1
3	0	0	1	0
4	0	0	1	1
⋮				
16	1	1	1	1

If c was always the same (say, $c = 4$), one could use a datastep as follows:

```
data temp;
  do y1 = 0 to 1;
    do y2 = 0 to 1;
      do y3 = 0 to 1;
        do y4 = 0 to 1;
          output;
        end;
      end;
    end;
```

```
  end;
run;
```

Below is a more general way using macros that works for any value of c:

```
%macro binary(numb, set);

  %let X = do y1 = 0 to 1%str(;);
  %let endit = end%str(;);

  *Generalize for number of binary variables;
  %do j = 2 %to &numb;
    %let X =  &X.do y&j=0 to 1%str(;);
    %let endit = &endit end%str(;);
   %end;

  *Create set of all possible;
  data &set;
    &X;
      output;
    &endit;
   run;

  title2 "All possible combinations of &numb binary variables";
  proc print data=&set;
  run;

%mend binary;

%binary(4, set4);
```

Chris Bilder, STAT 850
All possible combinations of 4 binary variables

Obs	y1	y2	y3	y4
1	0	0	0	0
2	0	0	0	1
3	0	0	1	0
4	0	0	1	1
5	0	1	0	0
6	0	1	0	1
7	0	1	1	0
8	0	1	1	1
9	1	0	0	0
10	1	0	0	1
11	1	0	1	0
12	1	0	1	1
13	1	1	0	0
14	1	1	0	1
15	1	1	1	0
16	1	1	1	1

Comments:

- The Log window is helpful to see how `&X` and `&endit` values are created though the do loop.
- The `%str` macro function helps SAS recognize that a semicolon is wanted without ending the SAS line of code.
- Notice the use of a period after &X within the do loop. This was done to distinguish the macro variable from the remaining text after it. The period is not included in how SAS recognizes the resolved name.

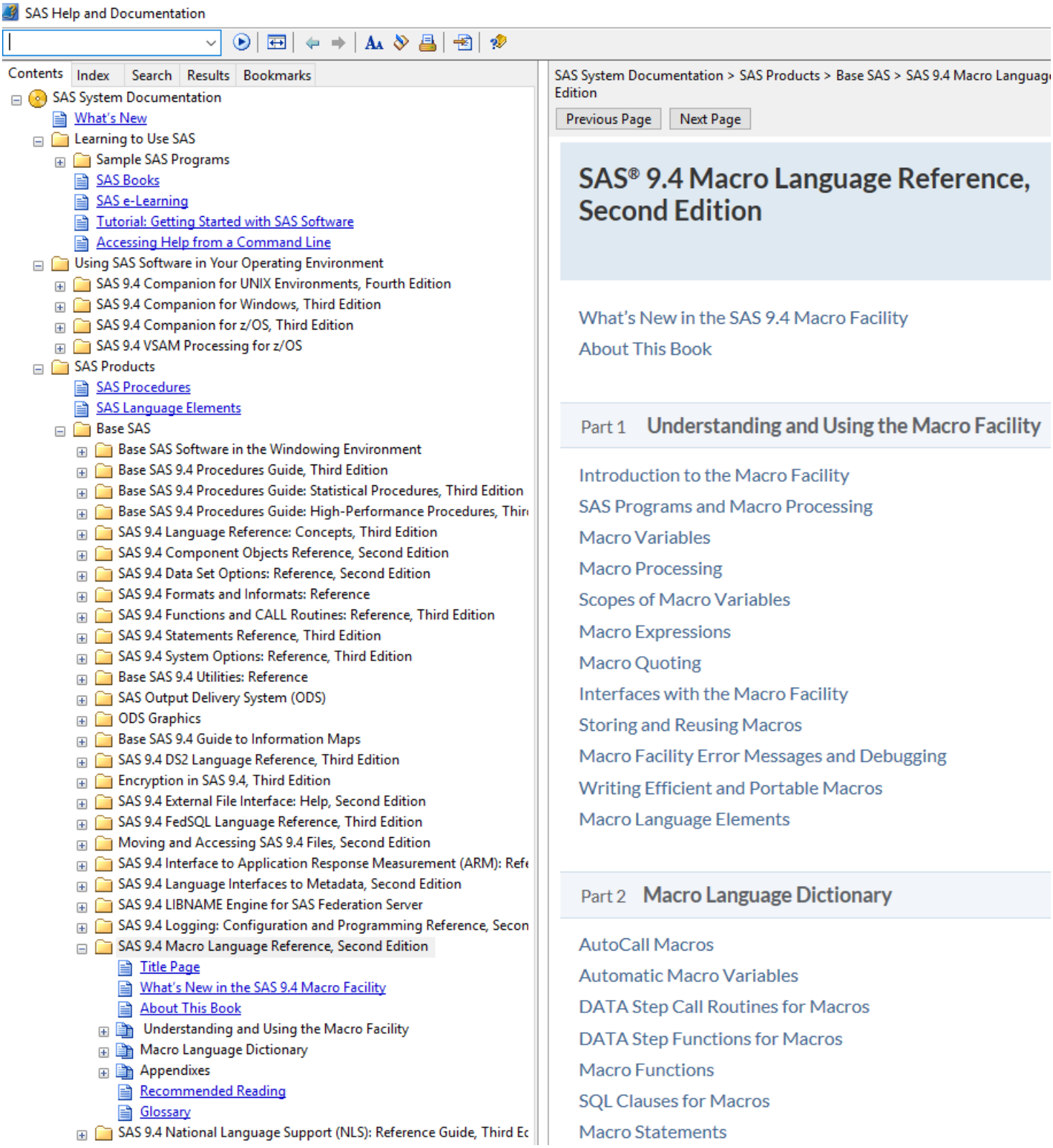
Final comments

- Not all of the Enhanced Editor's syntax highlighting works for code within a macro.

- The macro function code in the Enhanced Editor window can be folded into one line.
- For large SAS programs used in research, I have typically arranged most of my SAS code into macro functions at the beginning of my program. At the end of my program, I have a "main program" area where I call each macro function as needed. My boot_2mrcv_final.sas program at `http://www.chrisbilder.com/mrcv/Comm` provides an example of this organization structure, where the main program is actually in the last macro. Furthermore, this program is included via `%include` in control_2mrcv.final.sas.
- Macro variables can be created from values within a data set. Below is code from my pre-defined.sas program which prints the current time to the Log window.

```
*This data set will not exist;
data _null_;
  set time;
  call symput("hour", time);
  call symput("minute", minute);
  call symput("second", second);
run;

%put &hour &minute &second;
```

- SAS Help for macros:


SAS Help and Documentation
Contents
Index
Search
Results
Bookmarks
SAS System Documentation
What's New
Learning to Use SAS
Sample SAS Programs
SAS Books
SAS e-Learning
Tutorial: Getting Started with SAS Software
Accessing Help from a Command Line
Using SAS Software in Your Operating Environment
SAS 9.4 Companion for UNIX Environments, Fourth Edition
SAS 9.4 Companion for Windows, Third Edition
SAS 9.4 Companion for z/OS, Third Edition
SAS 9.4 VSAM Processing for z/OS
SAS Products
SAS Procedures
SAS Language Elements
Base SAS
Base SAS Software in the Windowing Environment
Base SAS 9.4 Procedures Guide, Third Edition
Base SAS 9.4 Procedures Guide: Statistical Procedures, Third Edition
Base SAS 9.4 Procedures Guide: High-Performance Procedures, Thir
SAS 9.4 Language Reference: Concepts, Third Edition
SAS 9.4 Component Objects Reference, Second Edition
SAS 9.4 Data Set Options: Reference, Second Edition
SAS 9.4 Formats and Informats: Reference
SAS 9.4 Functions and CALL Routines: Reference, Third Edition
SAS 9.4 Statements Reference, Third Edition
SAS 9.4 System Options: Reference, Third Edition
Base SAS 9.4 Utilities: Reference
SAS Output Delivery System (ODS)
ODS Graphics
Base SAS 9.4 Guide to Information Maps
SAS 9.4 DS2 Language Reference, Third Edition
Encryption in SAS 9.4, Third Edition
SAS 9.4 External File Interface: Help, Second Edition
SAS 9.4 FedSQL Language Reference, Third Edition
Moving and Accessing SAS 9.4 Files, Second Edition
SAS 9.4 Interface to Application Response Measurement (ARM): Refe
SAS 9.4 Language Interfaces to Metadata, Second Edition
SAS 9.4 LIBNAME Engine for SAS Federation Server
SAS 9.4 Logging: Configuration and Programming Reference, Secon
SAS 9.4 Macro Language Reference, Second Edition
Title Page
What's New in the SAS 9.4 Macro Facility
About This Book
Understanding and Using the Macro Facility
Macro Language Dictionary
Appendixes
Recommended Reading
Glossary
SAS 9.4 National Language Support (NLS): Reference Guide, Third Ec
SAS System Documentation > SAS Products > Base SAS > SAS 9.4 Macro Languag
Edition
Previous Page
Next Page
SAS® 9.4 Macro Language Reference, Second Edition
What's New in the SAS 9.4 Macro Facility
About This Book
Part 1 Understanding and Using the Macro Facility
Introduction to the Macro Facility
SAS Programs and Macro Processing
Macro Variables
Macro Processing
Scopes of Macro Variables
Macro Expressions
Macro Quoting
Interfaces with the Macro Facility
Storing and Reusing Macros
Macro Facility Error Messages and Debugging
Writing Efficient and Portable Macros
Macro Language Elements
Part 2 Macro Language Dictionary
AutoCall Macros
Automatic Macro Variables
DATA Step Call Routines for Macros
DATA Step Functions for Macros
Macro Functions
SQL Clauses for Macros
Macro Statements