Programming with SPSS Syntax and Macros

SPSS Inc.
233 S Wacker Drive, 11th Floor
Chicago, Illinois 60606
312.651.3000

Training Department
800.543.6607

v10.0 Revised 12/31/99 ss

SPSS Neural Connection, SPSS QI Analyst, SPSS for Windows, SPSS Data Entry II, SPSS-X, SCSS, SPSS/PC, SPSS/PC+, SPSS Categories, SPSS Graphics, SPSS Professional Statistics, SPSS Advanced Statistics, SPSS Tables, SPSS Trends, SPSS Exact Tests, and SPSS Missing Value are the trademarks of SPSS Inc. for its proprietary computer software. CHAID for Windows is the trademark of SPSS Inc. and Statistical Innovations Inc. for its proprietary computer software. Excel for Windows and Word for Windows are trademarks of Microsoft; dBase is a trademark of Borland; Lotus 1-2-3 is a trademark of Lotus Development Corp. No material describing such software may be produced or distributed without the written permission of the owners of the trademark and license rights in the software and the copyrights in the published materials.

General notice: Other product names mentioned herein are used for identification purposes only and may be trademarks of their respective companies.

Copyright(c) 2000 by SPSS Inc.

All rights reserved.

Printed in the United States of America.

No part of this publication may be reproduced or distributed in any form or by any means, or stored on a database or retrieval system, without the prior written permission of the publisher, except as permitted under the United States Copyright Act of 1976.

Programming with SPSS Syntax and Macros
Table of Contents

Chapter 1 **Introduction and Syntax Review**

A Data Manipulation Example	1 - 1
A Macro Example	1 - 3
Rules and Aids for SPSS Syntax	1 - 6
Advice for Those Working with Syntax	1 - 9
Summary	1-10

Chapter 2 **Basic SPSS Programming Concepts**

Command Types in SPSS	2 - 2
The Three Types of SPSS Programming	2 - 3
SPSS Data Definition	2 - 5
SPSS Programming Constructs	2 - 6
Do If & End If	2 - 6
Do Repeat & End Repeat	2 - 7
Loop & End Loop	2 - 9
Scratch Variables	2-11
Vector	2-12
Summary	2-16

Chapter 3 **Complex File Types**

ASCII Data and Records	3 - 2
File Types	3 - 2
Syntax Basics	3 - 3
Data File Structure	3 - 4
Grouped Data	3 - 4
Mixed Data	3 - 5
Nested Data	3 - 6
Reading a Mixed File	3 - 7
Errors in the Data	3-10
Grouped File Type Without Record Information	3-12
Summary	3-17

Chapter 4 **Input Programs**

Syntax Components	4 - 2
Example 1: Change the Case Base of a File	4 - 2
End of Case Processing	4 - 7
End of File Processing	4 - 9
Checking Input Programs	4-10
Incomplete Input Programs	4-11

Example 2: Reading Files with Missing Identifiers 4-14
When Things Go Wrong 4-20
Summary 4-21

Chapter 5 **Advanced Data Manipulation**
Reading a Comma-Delimited File 5 - 2
Reading Multiple Cases on the Same Record 5 - 7
An Existing SPSS Data File with Repeating Data 5-11
Print Command for Diagnostics 5-15
Practical Example: Consolidating Transactions 5-17
Summary 5-24
Appendix: Identifying Missing Values by Case 5-25

Chapter 6 **Introduction to Macros**
Macro Basics 6 - 2
Macro Arguments 6 - 3
Macro Tokens 6 - 3
Viewing a Macro Expansion 6 - 7
Keyword Arguments 6 - 8
Using a Varying Number of Tokens 6-10
When Things Go Wrong 6-15
Summary 6-18

Chapter 7 **Advanced Macros**
Looping in Macros 7 - 2
Producing Several Clustered Bar Charts 7 - 2
Double Loops in Macros 7 - 5
String Manipulation Functions 7 - 7
Direct Assignment of Macro Variables 7 - 7
Conditional Processing 7 - 7
Creating Concatenated Stub and Banner Tables 7 - 8
Additional Recommendations 7-13
Summary 7-13

Chapter 8 **Macro Tricks**
Combining Input Programs and Macros 8 - 2
Ordering Tables and Charts 8 - 6
The Case of the Disappearing Command 8 - 9
Summary 8-14

Exercises **Exercises**
Exercises E - 1

Chapter 1 Introduction and Syntax Review

Topics

A Data Manipulation Example
A Macro Example
Rules and Aids for SPSS Syntax
Advice for Those Working with Syntax

INTRODUCTION

This course has two major topical areas. We will review how to use SPSS Syntax to perform complex data manipulations that are not available under the SPSS menu system. This will be of interest to those who need to read complex data files from legacy computer systems (for example, legacy health care data, transaction oriented sales systems) and those who find they need to reorganize their data in order to perform a desired analysis. Examples of the latter include marketing and customer relationship studies in which a number of products (or services of an company) are rated on each of many attributes. All information from a respondent is typically stored in a single record, but needs to be spread across multiple records in order for factor analysis and perceptual mapping to be performed. When preparing data for churn (customer retention – for telecoms, credit card issuers, insurance companies) studies, comparisons might need to be made across transactional records sorted by customer ID and date. SPSS Syntax permits a richer array of data manipulations in this content than would the menu system. In short, we will examine uses of SPSS Syntax to facilitate analysis of files with complex structures or files that must be restructured for a desired analysis.

The second topical area concerns automation in SPSS through the SPSS macro language. SPSS macros can generate SPSS Syntax, which is then executed. For this reason, macros are very handy in situations where SPSS Syntax needs to be run repeatedly, but with minor and systematic changes each time. For example, you might wish to produce thirty Interactive graphs, each a clustered bar chart containing a demographic variable and one of thirty rating scale variables. Within the SPSS menu system, changes would have to be made in the Interactive graph dialog box for each graph. Instead, an SPSS macro could generate the SPSS Syntax for each interactive graph within a loop, substituting a new rating scale variable name per iteration. In this way, the SPSS macro language can automate what would otherwise be time-consuming tasks for the analyst.

Since these topics involve SPSS Syntax, we will use the dialog boxes within SPSS infrequently. A prerequisite for this course is familiarity with SPSS Syntax at the level of our *Introduction to SPSS Syntax* training course. In this chapter, we will present a sample of data manipulation with SPSS syntax and a macro example, and provide a brief review of and some recommendations for SPSS Syntax.

A DATA MANIPULATION EXAMPLE

To illustrate the type of data manipulation that can be performed with SPSS Syntax, we will display the beginning and final form of a data file recording SPSS training course purchases. Within the Training department, there was interest in examining patterns of training courses taken by SPSS customers, and an analysis was performed using SPSS Clementine. However, a requirement of the analysis was a data set in which all courses taken by a customer (an SPSS ID) were contained in a single customer record.

The original data file, extracted from a transaction database, contained one record per course taken, since an instance of a course being taken by a customer constituted a sales transaction. We show this below.

Figure 1.1 Training Sales Data - Transaction File

TrainTransact - SPSS Data Editor

File Edit View Data Transform Analyze Graphs Utilities Window Help

1 : spss_id 1499

	spss_id	course	var	var	var	var	var	var
1	1499	I02108						
2	20034	C21205						
3	20034	C22305						
4	20034	C50108						
5	20034	C50208						
6	30366	E02409						
7	30366	E05629						
8	30366	E05614						
9	32382	W22205						
10	32382	W30409						
11	32382	W33614						
12	32439	C10105						
13	32439	C10205						

Data View / Variable View

SPSS Processor is ready

Each record in this file is a sales transaction involving a specific training course. The two fields displayed are customer ID and course taken (which contains city, sequence within the year, and training course code information). Additional fields, such as date and price, were previously removed since they were not needed for this analysis. Here different courses taken by an individual SPSS customer are scattered throughout the file. Even if the file were sorted by customer ID, the fact that the training course history for a single customer is spread across a

number of records, that varies from customer to customer, would create difficulties for the analysis procedures.

Figure 1.2 Training Sales Data - One Record Per Customer

Untitled - SPSS Data Editor

File Edit View Data Transform Analyze Graphs Utilities Window Help

1 : spss_id 1499

	spss_id	basics61	de	basics75	basics80	inter61	inter75	inter80
1	1499	0	0	0	1	0	0	
2	20034	0	0	0	1	0	1	
3	30366	0	0	0	0	0	0	
4	32382	0	0	0	0	0	1	
5	32439	0	0	1	0	0	1	
6	33871	0	0	0	1	0	0	
7	105302	0	0	0	0	0	1	
8	105796	0	0	0	0	0	0	
9	106133	0	0	0	0	0	0	
10	106524	0	1	0	0	0	0	
11	107136	0	0	0	0	0	0	
12	107479	0	0	0	0	0	1	
13	107803	0	0	0	1	0	0	

Data View / Variable View

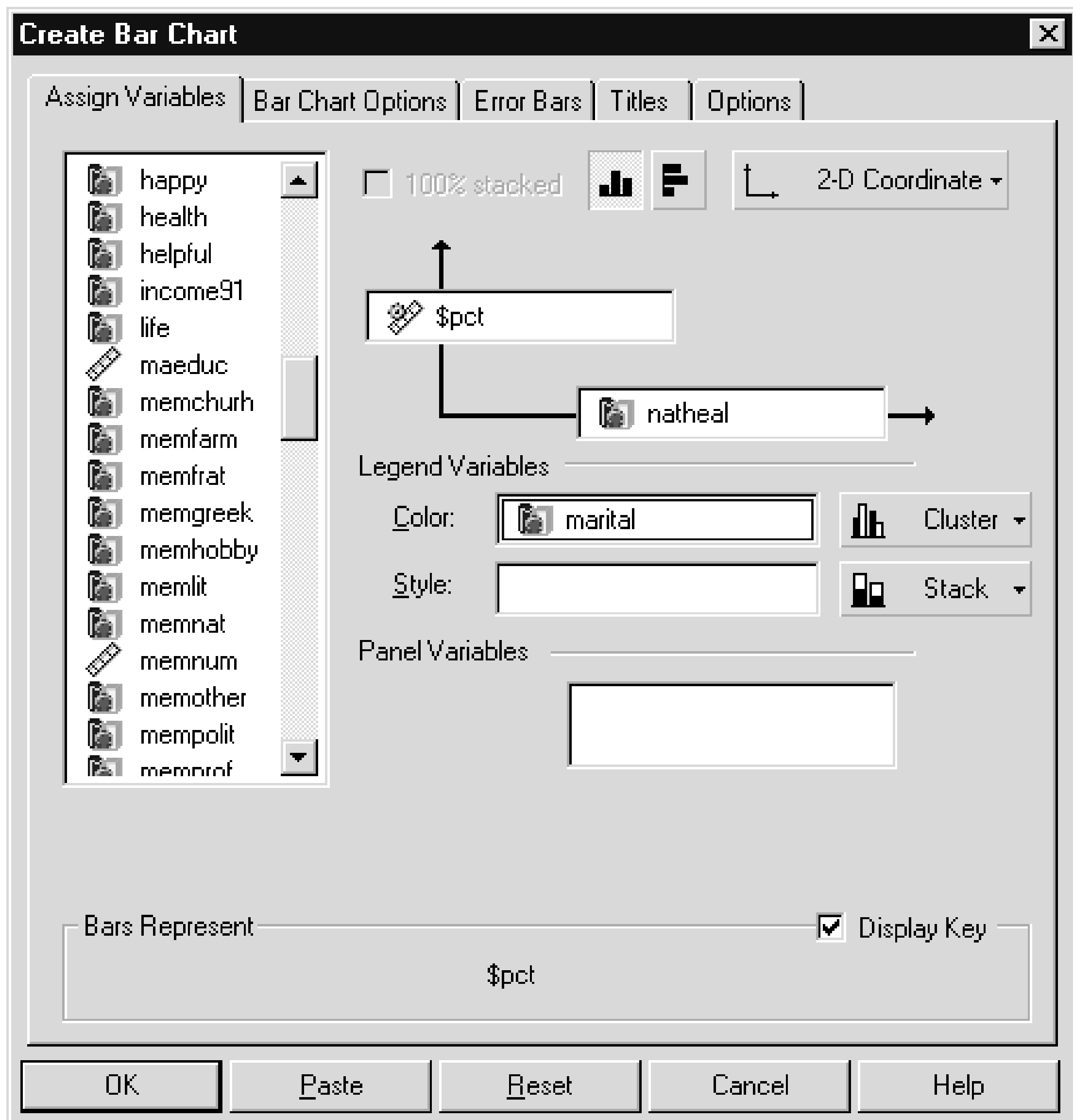
SPSS Processor is ready

The training data has been reorganized so there is a single record per customer ID and a separate variable for each training course. These course variables are coded 1 if a customer signed up for the course and 0 if not. This structure makes it easy to explore associations among training courses taken by customers. The SPSS syntax to perform the data reorganization involved two steps: creating a vector of variables in which each variable represented a specific course, and aggregating this file to the customer ID level. The logic behind these operations is reviewed in Chapter 5.

A MACRO EXAMPLE

We mentioned earlier that SPSS macros generate SPSS Syntax. A common use of macros is to produce a series of syntax commands that vary in specific ways, for example, a set of Interactive Graph or Tables commands in which each command runs an analysis based on a different variable. Thus one macro produces the same result as many syntax commands (which it creates) or interactions with a dialog box. To demonstrate, we will display results from a macro that produces a set of Interactive graphs, substituting different variables in clustered bar charts (this macro is discussed in Chapter 7).

Figure 1.3 Create Bar Chart Dialog Box



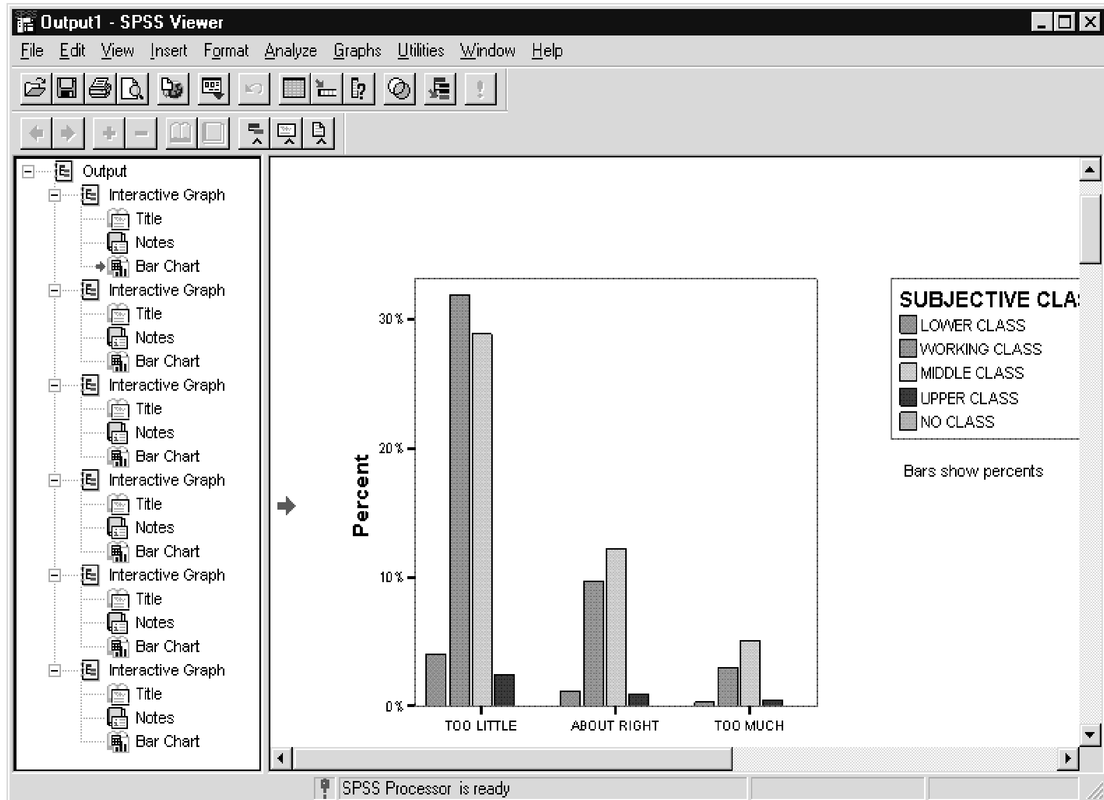
The dialog above will create a clustered bar chart displaying attitude toward government action on health for different marital status groups. Note that only a single variable can be placed in horizontal and Color boxes. (Note: actually multiple variables can be placed in a single box, but this action will not produce multiple charts.) Thus creating a series of charts, in which either the horizontal axis or Color variables change, would require repeated visits to this dialog, substituting one variable at a time. However, the macro below can build many clustered bar charts.

Figure 1.4 Macro to Produce Multiple Bar Charts (Interactive Graphs)

```
* The  CLUS argument contains a list of Cluster variable names.
* Macro invocation assumes Gss94.sav SPSS data file is active.

DEFINE Clu2IBar (CAT = !CHAREND("/") /
                 CLUS = !CMDEND).

!DO !I !IN (!CAT)
!DO !J !IN (!CLUS)

IGRAPH /VIEWNAME='Bar Chart' /X1 = VAR(!I) TYPE = CATEGORICAL /Y = $pct
 /COLOR = VAR(!J) TYPE = CATEGORICAL CLUSTER /COORDINATE = VERTICAL
 /X1LENGTH = 3.0 /YLENGTH = 3.0 /X2LENGTH = 3.0 /CATORDER VAR(!I)
  (ASCENDING VALUES SHOWEMPTY) /CATORDER VAR(!J) (ASCENDING VALUES
  SHOWEMPTY) /BAR KEY=ON SHAPE = RECTANGLE BASELINE = 0.00.

!DOEND
!DOEND


!ENDDEFINE.

Clu2IBar CAT natheal natenvir  /clus class marital sex .
```

The details of this macro (Clu2IBar) will be discussed in Chapter 7. However, we point out that the IGRAPH command, which was pasted by clicking the Paste pushbutton in the Create Bar Chart dialog box (see Figure 1.3), is nested within two loops, each of which iterates over a list of variables supplied by the user. The invocation of the macro (last line in program), supplies two variable names for the horizontal axis variable and three variable names for the cluster variable. Thus six IGRAPH commands will be generated, resulting in the six bar charts shown below.

Figure 1.5 Bar Charts Produced from Clu2IBar Macro



The six Interactive Graphs in the Outline pane were produced from the macro. In this way, macros can automate the running of sets of similar analyses. The second section of this course reviews SPSS macros in detail.

RULES AND AIDS FOR SPSS SYNTAX

Since this course involves either the writing or generation of SPSS Syntax, we begin be reviewing the rules of SPSS Syntax and how to obtain syntax help.

The syntax rules for editing and writing SPSS commands are as follows:

1. Each new command must begin on a new line and end with a period (.) or a blank line.
2. *Each command must begin in the first column of a new line.
3. *Continuation lines of a command must be indented at least one space.
4. Variable names must be spelled out fully.
5. Subcommands must be separated with a forward slash (/). The slash before the first subcommand is usually optional.
6. *Each line of command syntax cannot exceed 80 characters.

*Not required when running from a Syntax window, but required when using the INCLUDE command or the SPSS Production Facility

Syntax commands produced by clicking the Paste pushbutton from an SPSS dialog box will conform to these rules, so the important issue is to remember them when editing or entering syntax.

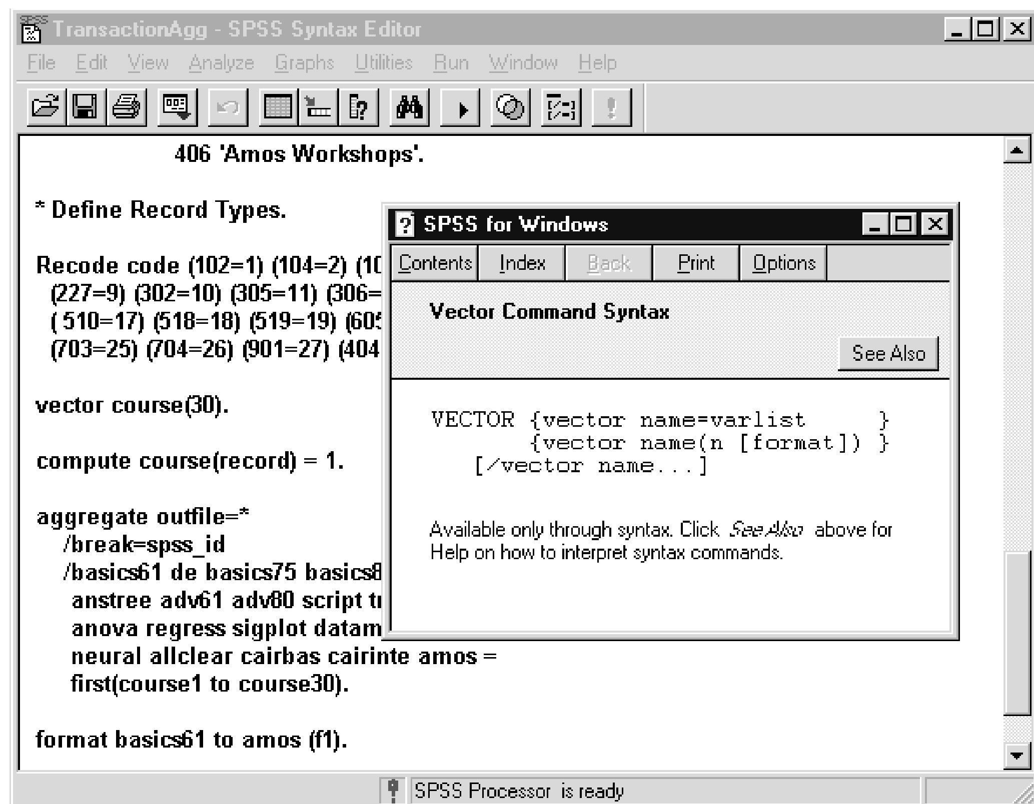
There are several useful sources of help when writing SPSS syntax. A quick reminder of the keywords and requirements for an SPSS command are only a tool-button click away. To demonstrate:

From within SPSS:

Click **File..Open..Syntax**
Move to the **c:\Train\ProgSynMac** directory
Double click on **TransactionAgg**
Scroll down to the **Vector** command
Click on the **Vector** command (so the insertion pointer touches it)

Click the **Syntax Help** tool

Figure 1.6 Syntax Help



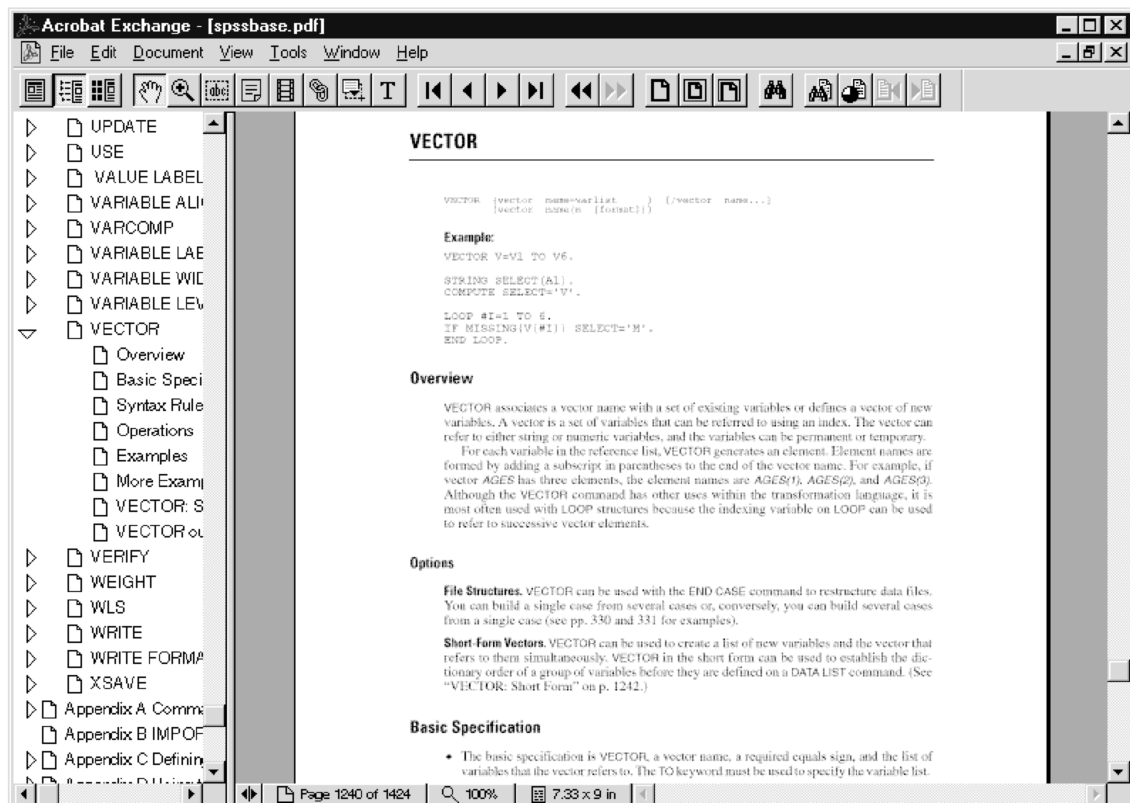
In this syntax summary for the Vector command, subcommand names and keywords are shown in upper case (some simple commands, like Vector, have no subcommands); lower case elements describe specifications that you supply (e.g. varlist indicates a list of variable names that you provide). Sections of the command enclosed in square brackets [ ] are optional, while those in braces indicate sets from which a single choice can be made. To focus on the required elements of the command, scan only sections not enclosed within square brackets.

While the syntax information about VECTOR is complete, there is no explanation about what each specification does. Although some might be obvious from their names, many are not. Complete documentation about SPSS for Windows syntax commands can be found in the *SPSS 10.0 Syntax Reference Guide* (included on the CD-ROM containing the SPSS 10.0 program). If copied to your hard drive during SPSS installation, you can access the guide from the main menu by clicking Help..Syntax Guide..Base (or one of the optional modules). Commands are listed alphabetically and the subcommand options are fully explained. Experienced syntax command users, needing only reminders, can work from the Syntax Help windows. For others, the Syntax Reference Guide is necessary.

Note The sequence below assumes the *SPSS 10.0 Syntax Reference Guide* has been installed on your machine. If not, it can be installed from the SPSS for Windows 10.0 CD-ROM.

Click **Help..Syntax Guide..Base**
Click the arrow ▷ beside **Commands** in the Outline pane
Scroll down to **VECTOR**
Click the arrow ▷ beside **VECTOR** in the Outline pane
Click on **VECTOR** in the Outline pane

Figure 1.7 SPSS Base 10.0 Syntax Reference for Vector Command



In addition to the syntax summary accessed through the Syntax Help tool, the *SPSS 10.0 Syntax Reference Guide* contains discussion, explanation and examples. All are useful when investigating the possibilities of an SPSS command. For those working often with SPSS Syntax, we strongly recommend installing the reference guide on your machine or purchasing a copy of the *SPSS 10.0 Syntax Reference Guide* in book form.

Click **File..Exit** to exit Adobe Acrobat and the *SPSS 10.0 Syntax Reference Guide*

ADVICE FOR THOSE WORKING WITH SYNTAX

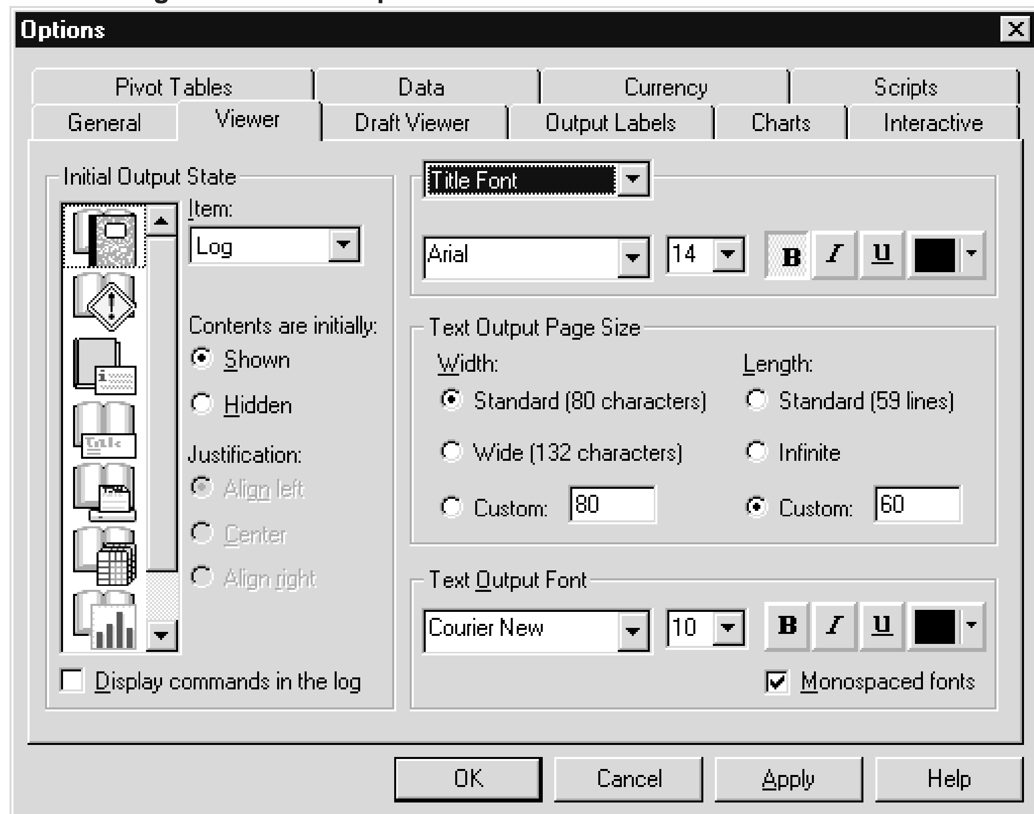
Finally, it is worth mentioning, at the risk of being obvious, several recommendations for those working with SPSS Syntax.

Display Syntax commands as Log Items

By default, SPSS does not display syntax in the Viewer window, although it is written to the SPSS journal file. If SPSS issues any error or warning messages, it is useful to see which command they follow. For this reason, while writing, editing and testing SPSS syntax, we recommend you set on the option to display syntax as a log item in the Viewer window. We will do this explicitly in the next chapter, but view the Options dialog here.

Click **Edit..Options**
Click the **Viewer** tab

Figure 1.8 Viewer Options



The checkbox in the lower left corner of Viewer tab within the Options dialog controls whether SPSS syntax commands display in the log.

Develop Basic Syntax Using Dialog Boxes

Although this course proves the exception to the rule (discussing Input Programs, Vectors and Loops), most SPSS commands can be generated by clicking the Paste pushbutton of the relevant dialog box. It is to your advantage to use dialogs, when possible, to construct the basic SPSS syntax and then edit is as needed. This will minimize errors by reducing your opportunity to make typing mistakes.

Use File New to Clear Data

If errors lead to complex SPSS data operations (Input Program) not completing, SPSS can be left in a waiting state. That is, it will not properly process new instructions until it has closure on the interrupted sequence. To clear the current data state of SPSS, you can run the NEW DATA command or click File..New..Data. When writing complex Input Programs (see Chapter 4 and 5), you might consider beginning the program with a NEW DATA command to insure that any problem data state has been cleared prior to running your program. We illustrate this in Chapter 4.

Test After Each Step

It is a difficult challenge to foresee all possible data problems that a program might face and it rarely the case that any program, for that matter, runs correctly the first time. For these reasons it is important to systematically test and check results at each stage of the process. Full programming methodologies have been developed to this end. Here we merely wish to recommend that, during development, you include displays or procedures to check the results of each set of operations, so that when something goes awry you have a way of isolating and identifying the problem. The Data Editor display in SPSS is useful for this purpose, as are the Frequencies, Crosstabs, Case Summaries and List procedures, and the Print transformation. These will be used repeatedly in the examples we present in this course, and we can assure you that the consultants in the SPSS Consulting group use them heavily.

Delete Items in Viewer Window

If an error occurs, and you have read and understood the warning messages in the Viewer window, it is often a good idea to delete the results in the Viewer window before rerunning your program. This is because syntax commands may be appended to the last Log item in the Outline pane, making it difficult to distinguish the old warning messages from the new results.

SUMMARY

In this chapter we introduced, with examples, the major focus areas of this course: Syntax for complex data manipulation and SPSS Macros. We also briefly reviewed the available help for SPSS Syntax and offered some advice for those working with SPSS Syntax.

Chapter 2 Basic SPSS Programming Concepts

Topics

Introduction
Command Types in SPSS
The Three Types of SPSS Programming
SPSS Data Definition
SPSS Programming Constructs
A Note About Program Execution
Analysis Tip: Reordering Variables

INTRODUCTION

All SPSS procedures are built upon a powerful programming language that has been consistent, though greatly extended, since SPSS was first developed as a mainframe statistical software program. This course will teach you how to use this language and other features for file and data input and manipulation, and for overall control of SPSS execution.

The SPSS language, called *syntax*, is generated by the program every time a user clicks on the OK button in a dialog box to execute a procedure. Behind the scenes, SPSS builds syntax to send to the SPSS central engine to execute a particular procedure or transformation. Using the Paste button in a dialog box places a copy of that syntax in a Syntax window so that it can be edited or saved and used again.

SPSS syntax is also often called a *command* or set of commands. The grammar or rules associated with commands are fairly simple, and we will review them as necessary throughout the chapters.

Note about Data File Access When Running SPSS from a Remote Server

SPSS for Windows 10.0 can run entirely on your desktop machine. Alternatively, an SPSS Client, through which you request analyses and view results, can run on your desktop, while the analyses are run by the SPSS Server, possibly located on a different machine. In this course, except for the directory you use to access the training data files, it makes no difference whether the SPSS Server is located on your desktop or a different computer. The SPSS Server Login dialog (click File..Switch Server) allows you to connect to a remote SPSS Server (if installed on your network).

If you are running SPSS from a Remote (not Local) server, then to use the data files accompanying this course, they must be copied either to the server running SPSS or to a directory that can be accessed by (mapped from) the server. The directory references in this guide assume you are running SPSS as a local server and can thus directly access files stored on your hard drive.

COMMAND TYPES IN SPSS

Most users successfully program in SPSS without a complete understanding of the various command and program states of SPSS, and you can too. Nevertheless, it helps to know a little about this subject, particularly to help put the various capabilities of SPSS in context. Many users know the difference between *transformations* and *procedures*, the two main types of commands, but there are a few others:

File Definition Commands: As their name implies, all these commands are used to input data into SPSS. They include familiar commands such as GET, DATA LIST, or GET CAPTURE ODBC, but also others like FILE TYPE, IMPORT, or MATCH FILES.

Input Program Commands: These are specialized commands, also used to input or create data in SPSS. These commands are more esoteric and include RECORD TYPE, REPEATING DATA, and END CASE. We have more to say about this below when discussing the data step in SPSS.

Transformation Commands: These commands are quite varied in their operation, but the key element they share in common is that they neither input data nor analyze data. Instead, they modify data (COMPUTE, RECODE), create new variables (VECTOR, NUMERIC), write out data (WRITE), or label data (VARIABLE LABELS). To be specific, transformations do not cause SPSS to read the data file.

Procedure Commands: Almost all of these commands analyze data. However, the actual definition of a procedure in SPSS is a command that causes data to be read. Thus, SAVE is also a procedure because it causes the data to be read and an SPSS system file (an .SAV extension) to be created.

Utility Commands: These commands handle a variety of chores. They include commands to add comments (COMMENT, DOCUMENT), to define a new file (NEW FILE), and to define macros (DEFINE--!END DEFINE).

Knowing about command types will be helpful in understanding how and why a program operates. For example, XSAVE, an alternative to the SAVE command, can be used within a loop because it is a transformation, not a procedure.

THE THREE TYPES OF SPSS PROGRAMMING

Logically, there are three general methods of programming in SPSS. Two of them involve syntax, while the third uses a version of the Basic programming language (Sax Basic).

Standard Syntax: These programs are the most common and simply involve writing a series of SPSS commands to accomplish a set of tasks. An example of a simple program is shown in the box (this program uses the FILE TYPE command to read a non-standard ASCII data file). In a standard syntax program, each command does one thing, and it does not refer to other SPSS syntax. Standard programs are executed either through the Run button, the INCLUDE command, or the SPSS Production Facility.

```
* SPSS Example to read a nested file * .
FILE TYPE NESTED FILE 'C:\TEST.DAT' / RECORD RECID 1
    (A) CASE 3 (F).
RECORD TYPE 'H'.
DATA LIST /H1 to H10 5-14.
RECORD TYPE 'F'.
DATA LIST /F1 to F5 15-19.
RECORD TYPE 'P'.
DATA LIST /CASEX 3 P1 to P3 20-22.
END FILE TYPE.

LIST
```

Macros: Many programs allow users to define *macros*, which are typically a series of commands grouped together as a single command to make everyday tasks easier and more convenient. Macros can often be assigned to a toolbar or menu to make them readily accessible. Normally, macros are saved as a series of instructions in a special macro language.

SPSS Macros are a bit different and not exactly parallel to the more common definition of a macro in other programs. First, they are written in SPSS syntax (plus a few special macro commands) and are essentially executed like any other syntax file. Second, they generate customized SPSS command syntax, i.e., standard syntax, to reduce the time and effort needed by the program writer to perform complex and repetitive tasks. There is no special macro editor in SPSS or macro facility to execute a macro; again, a macro is simply a specialized syntax file. Below is an example of a macro that automates the production of a bar graph and the insertion of today's date into the title of a graph (this program actually defines two macros). The macro begins with DEFINE and ends with !ENDDEFINE.

```
DEFINE !GRAPHIT ( ARG1 !TOKENS(1) / ARG2 !TOKENS(1)/
     ARG3 !TOKENS(1)) .
GRAPH /BAR(SIMPLE)=COUNT BY !ARG1
  /TITLE=
 "EXAMPLE OF MACRO GRAPHING TITLES AND DATES"
  /SUBTITLE !QUOTE(!CONCAT(!UNQUOTE(!ARG2),
      "Something ",!UNQUOTE(!ARG3)))
  /FOOTNOTE= !CONCAT(""Today is ",!EVAL(@DATEIT),""" ).
!ENDDEFINE .
DATA LIST FREE / A.
BEGIN DATA
1 2 3 2 1 2 4
END DATA.
* The following defines a macro entity with today's date *.
DO IF $CASENUM=1 .
WRITE OUTFILE 'TMP'/
  'DEFINE @DATEIT()',$TIME(ADATE),'!ENDDEFINE .'.
END IF .
EXECUTE.
INCLUDE 'TMP' .
!GRAPHIT ARG1=A ARG2="ARGUMENT 2"
     ARG3="ARGUMENT 3".
```

SPSS Scripting Facility: The scripting facility also allows you to automate tasks in SPSS. It has far more power and capabilities than SPSS macros. You can accomplish the same tasks that you would with a macro, but can do much more, including the creation of dialog boxes, the customization of output in the Viewer window, or the writing out of selected portions of output to a separate file for use in other programs. Scripts can be set to run automatically or run at user choice.

Unlike syntax programs or macros, scripts are written in a special language, Sax Basic, that is similar to the macro language in other programs, such as Visual Basic for Applications. Of course, scripts can also use SPSS syntax in their definition, and they can be assigned to a menu choice like macros. Scripts are often somewhat lengthy compared to standard syntax, so we won't display an example in this chapter. SPSS Scripts are covered in the *Programming with SPSS Scripts* training course.

SPSS DATA DEFINITION

A substantial portion of this course is devoted to the manipulation of files and data with SPSS. As such it will be helpful to understand a bit about SPSS data file definition. More information is available in the Commands and Program States Appendix in the *SPSS Base 10.0 Syntax Reference Guide* (this guide is available on the CD-ROM containing SPSS and can be copied to your hard drive when SPSS is installed).

To do something in SPSS you need to define a working data file, possibly transform the data, and then analyze it. The first task is accomplished with a file definition command. A simple instance might be the GET command to open an SPSS data file, but a more complex one is INPUT PROGRAM to define a non-standard data file. In either case, file definition commands use an *input program state* to accomplish their job. In an input program state, SPSS must determine how to read a data file, what the definition is of a case, when to create a case, and when to create the data file.

Often these decisions are straightforward for SPSS. When you click on File…Open..Data, and name a file with an extension of SAV, SPSS knows how to read the file, that each logical record in the file is to be written to one row in the Data Editor, and that it should read the whole file. Or in the simple program below, the execution of the DATA LIST command (accessed by choosing File…Read Text Data; note that as of SPSS 10.0 a GET DATA command is pasted instead) tells SPSS that what follows is a normal file, where each line of data is to be written to a new row, or case, in the Data Editor, and that the last case should be written and the file created when the END DATA command is encountered.

```
DATA LIST FREE / X.
BEGIN DATA.
1 2 3
END DATA.
LIST.
```

However, even in these simple commands, SPSS enters an input program state that is more complex than it first appears. In fact, you can explicitly place SPSS into this state by using the INPUT PROGRAM command. Thus, the following program is equivalent to the first.

```
INPUT PROGRAM.
DATA LIST FREE / X.
END INPUT PROGRAM.
BEGIN DATA.
1 2 3
END DATA.
LIST.
```

The difference is that first, SPSS is explicitly put into the input program state, and second, SPSS is told when to quit the input program state (with END INPUT PROGRAM). There is, of course, no reason to use INPUT PROGRAM in this uncomplicated example, but for complex data, using an input program may be necessary to successfully read data into SPSS. We will see more of INPUT PROGRAM in Chapter 4.

The key to understanding and using complex file definition commands in SPSS is to understand that you, the user, are in charge of telling SPSS what constitutes a case or row in the Data Editor, when to create that case, and when to end the input program and create the working data file. As an illustration, it is possible through an input program to have SPSS read only a portion of a file rather than first creating a larger working data file which is then reduced by selecting certain cases to retain.

SPSS data files must be rectangular (denormalized, in database terminology). This means that there must be a value for every variable for every case. Or to put it another way, each row of the Data Editor defines a case to SPSS. Often, data come in a format that doesn't match this layout, and that is one of the most common uses for the input program capability. In addition, SPSS supplies several predefined complex file definitions that read common types of non-rectangular files (we will discuss these in Chapter 3). Changing the case definition of a file is a common technique to solve a variety of problems.

SPSS PROGRAMMING CONSTRUCTS

As with other programming languages, SPSS programs, whether they be standard syntax, macros, or scripts, all have several standard constructs that can be used to do many things. These include the ability to loop, to create an array of elements, to repeat actions, and to do actions only if some condition is true. We illustrate these concepts here with standard syntax; then later you will see these constructs used again in macro0s.

DO IF & END IF

A DO IF & END IF structure is used to execute transformations on subsets of cases based on some logical condition. It is often used to replace a long series of IF statements. The logical structure of a DO IF command sequence is:

```
DO IF (test for condition)
transformations
ELSE IF (test for another condition)
transformations
ELSE IF or ELSE
further transformations
END IF
```

The clear advantage is that not all statements are executed for each case, as is true for a series of IF statements. Consider regression analyses that found that the relationship between gross domestic product (GDP) and birth rate (BTHR) is not the same for first- and third-world countries (which is definitely true). The results of the separate regression analyses can be applied to a file of first- and third-world countries efficiently with these commands (where WORLD is the selection variable).

```
DO IF (WORLD = 1).
COMPUTE  BTHR = 10.872 + .0014 *GDP.
ELSE IF (WORLD = 3).
COMPUTE BTHR = 46.148 -.004 * GDP.
END IF.
```

Although we could have accomplished the same with two IF commands, the advantage is that the Else If and second Compute commands are not executed for first-world countries.

DO REPEAT & END REPEAT

A DO REPEAT construct allows you to repeat the same group of transformations on a set of variables, thereby reducing the number of commands that you must enter. SPSS must still execute the same number of commands; the efficiency comes for the user, not SPSS. To illustrate its use, let's access the 1994 General Social Survey file, stored in the c:\Train\ProgSynMac directory.

Displaying Variable Names in Dialog Boxes

First, to simplify the instructions in this course, we will request that variable names (and not the default variable labels) be displayed in dialog boxes. From within SPSS:

Click **Edit..Options**
Click the **Display Names** option button in the Variable Lists section of the General tab
Click the **Alphabetical** option button in the Variable Lists section of the General tab

In order to display SPSS commands in the Viewer window when we run analyses, we change one of the Viewer options.

Click **Viewer** tab in the SPSS Options dialog box
Click **checkbox** beside **Display commands in the log**
Click **OK**

Now to read the data.

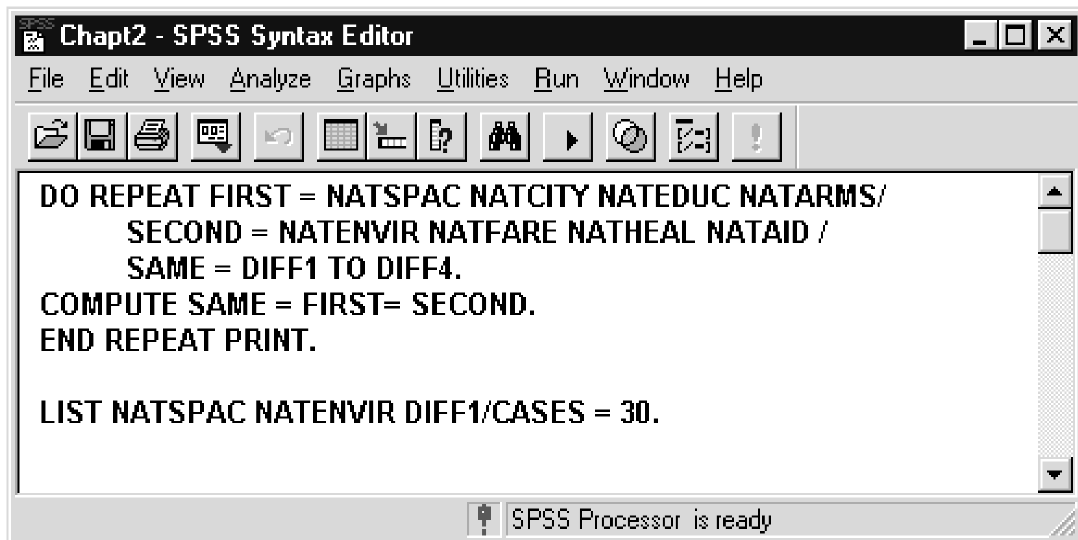
Click **File...Open..Data**
Move to the **c:\Train\ProgSynMac** directory (if necessary)
Double-click on **GSS94**

There are several questions in the file that ask about whether national spending on various programs or areas should be increased, stay the same, or be reduced. Imagine that we wish to compare pairs of

questions (urban problems and welfare) to see whether or not a respondent gave the same answer to each. The program in Figure 2.1 accomplishes that task. Open it by

> Clicking on **File...Open..Syntax** (move to c:\Train\ProgSynMac folder if necessary)
> Double-click on **CHAPT2**

Figure 2.1 DO REPEAT & END REPEAT Program



The Do Repeat structure requires that stand-in variable names be used to represent a list of variables or constants. The stand-in variables exist only within the DO REPEAT structure. Between the DO REPEAT and END REPEAT commands, transformation commands can be used, referencing the stand-in variables. The PRINT keyword on the END REPEAT command tells SPSS to list the commands generated by the DO REPEAT structure. (This is a good idea except when SPSS generates hundreds of commands.)

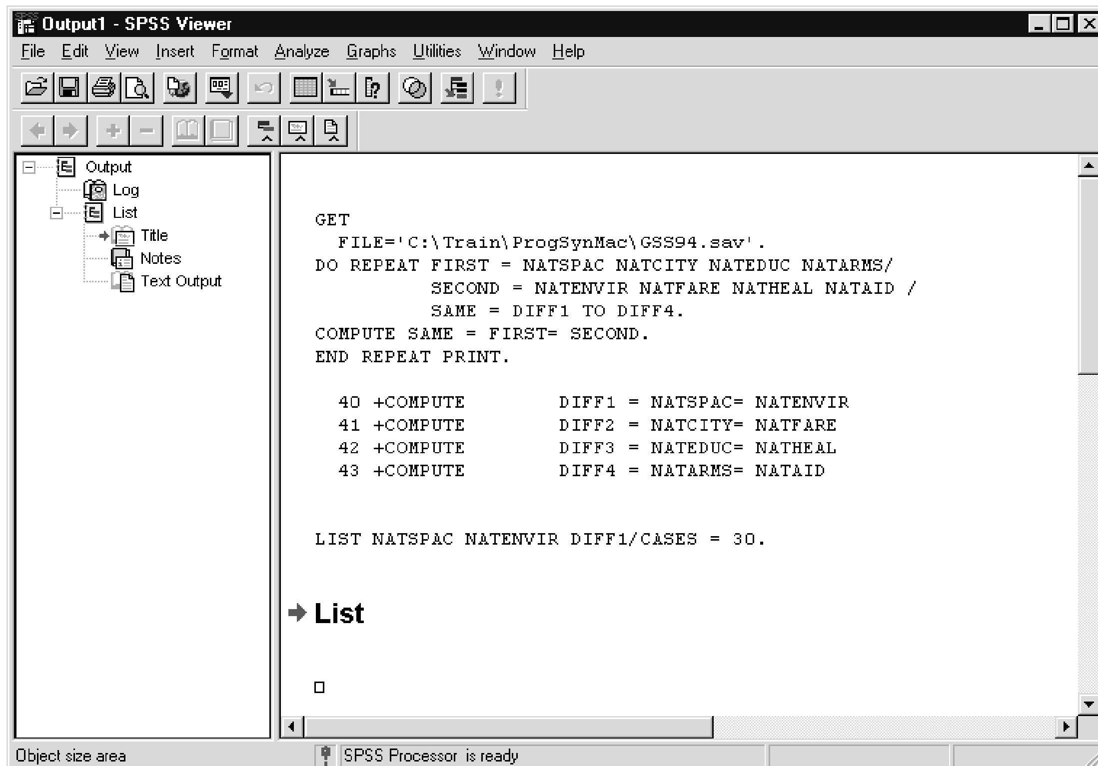
The COMPUTE command uses another feature of SPSS syntax, true/ false comparisons. The COMPUTE statement tells SPSS to compare the values of the elements in FIRST to SECOND, in pairs. When, for example, NATSPAC is equal to NATENVIR, the test is true and SPSS returns a “1” to the variable SAME. When the two responses are not equal, SPSS returns a false, or “0”, to SAME.

To see this in operation

> Highlight all the lines from **DO REPEAT to LIST**, then click on the **Run** button

After SPSS runs the commands the Viewer window opens, as shown in Figure 2.2. SPSS creates four COMPUTE commands based on the DO REPEAT structure. Scrolling down through the output from LIST (not shown) demonstrates that when NATSPAC is not equal to NATENVIR, DIFF1 is set equal to zero, and when the two responses are equal, DIFF1 is set to 1 (a value of 0 for either of the spending variables is defined as missing so the COMPUTE is not done).

Figure 2.2 Output from END REPEAT PRINT



Although in this instance little if any work was saved by the use of DO REPEAT, in many circumstances the savings can be substantial.

LOOP & END LOOP

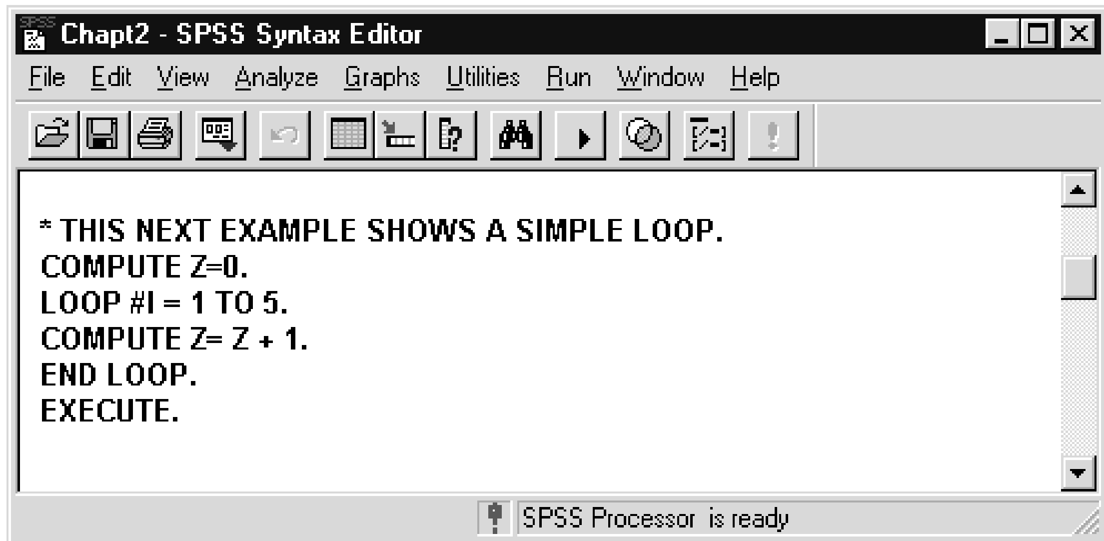
The DO REPEAT structure is an iterative construct because SPSS iterates over sets of elements to carry out the user instructions. A more generic form of iteration is provided by the looping facility in SPSS, represented by the LOOP & END LOOP commands. They can be used to perform repeated transformations on the same case until a specified cutoff is reached, which can be defined by an index on the LOOP command, an IF statement on the END LOOP command, or other options. By default, the maximum number of loops is 40, defined on the SET command. Almost any transformation can be used within a loop.

We begin with a very simple loop to illustrate its syntax.

Click on **Window...CHAPT2 - SPSS Syntax Editor** to return to the Syntax Editor window

Scroll down to the program shown in Figure 2.3

Figure 2.3 LOOP & END LOOP Example



On the LOOP command, we tell SPSS to loop five times with the index clause of #I=1 to 5. This tells SPSS to repeat the COMPUTE command five times for each person in the GSS file. Usually indices are increased by one, as in this example, but that is not always the case. Nor must they begin at 1.

The COMPUTE command itself tells SPSS to add one to the previous value of Z, which has initially been set to 0 before the loop. The loop then finishes with the required END LOOP command to tell SPSS the construct has finished.

A NOTE ABOUT PROGRAM EXECUTION

Notice that the program ends with an EXECUTE command. When running syntax from a Syntax window, SPSS does not immediately process transformations by reading the data file. Instead, it stores transformations in memory and waits until a command is encountered which forces a pass of the data. This is in comparison to running SPSS commands from a dialog box, where the command is executed immediately after the OK button is clicked. The EXECUTE command forces a pass of the data and executes any preceding transformations.

Highlight the commands from the first **COMPUTE to EXECUTE**

Click on the **Run** button

To see the effect of this program

Switch to the **Data Editor** window
Scroll to the **last column** in the Data View sheet

Figure 2.4 Data Editor with variable Z added

Gss94 - SPSS Data Editor

File Edit View Data Transform Analyze Graphs Utilities Window Help

1 : year 94

	god	agewedcr	diff1	diff2	diff3	diff4	z	var
1	9	0	.	.	.	.	5.00	
2	0	0	.00	.00	1.00	1.00	5.00	
3	0	0	.	.	.	.	5.00	
4	6	0	.00	.00	.00	.00	5.00	
5	0	0	1.00	.00	1.00	1.00	5.00	
6	0	0	.00	1.00	1.00	.	5.00	
7	6	0	.00	.00	1.00	1.00	5.00	
8	0	0	.	.	.	.	5.00	
9	0	0	.	.	.	.	5.00	
10	6	0	.00	.00	1.00	.00	5.00	
11	6	0	.	.	.	.	5.00	
12	2	0	.00	.00	1.00	1.00	5.00	

Data View / Variable View

SPSS Processor is ready

SPSS has added Z to itself plus 1 five times, and since Z initially was zero, Z is now 5 for every case in the file. To reiterate, the LOOP command works within a case rather than across cases. We will see many uses of looping in programs, and the concept of looping will be repeated in macros and scripts.

SCRATCH VARIABLES

The variable #I used to index the loop does not exist in the GSS file. If it did, we would see it next to Z in the Data Editor. It hasn't been created by SPSS because it was declared a *scratch variable*. This is done by specifying a variable name that begins with the # character. Scratch variables are used in transformations or data definition when there is no reason to retain them in the data file. They cannot be used in procedures.

VECTOR

A vector is a construct used to reference a set of existing variables or newly created variables with an index. The vector can reference either string or numeric variables.

Here is how to create a vector from existing variables.

```
VECTOR SAT = SATCITY TO SATHEALT.
```

The vector SAT is created from the five questions in the General Social Survey that ask about a respondent's satisfaction with various aspects of his/her life. This vector is not visible in the Data Editor as a separate variable or set of variables because it is a logical construct from existing variables. These variables must be contiguous in the file; that is, they must be located next to each other when viewed in the Data Editor.

Conversely, the syntax

```
VECTOR X(10).
```

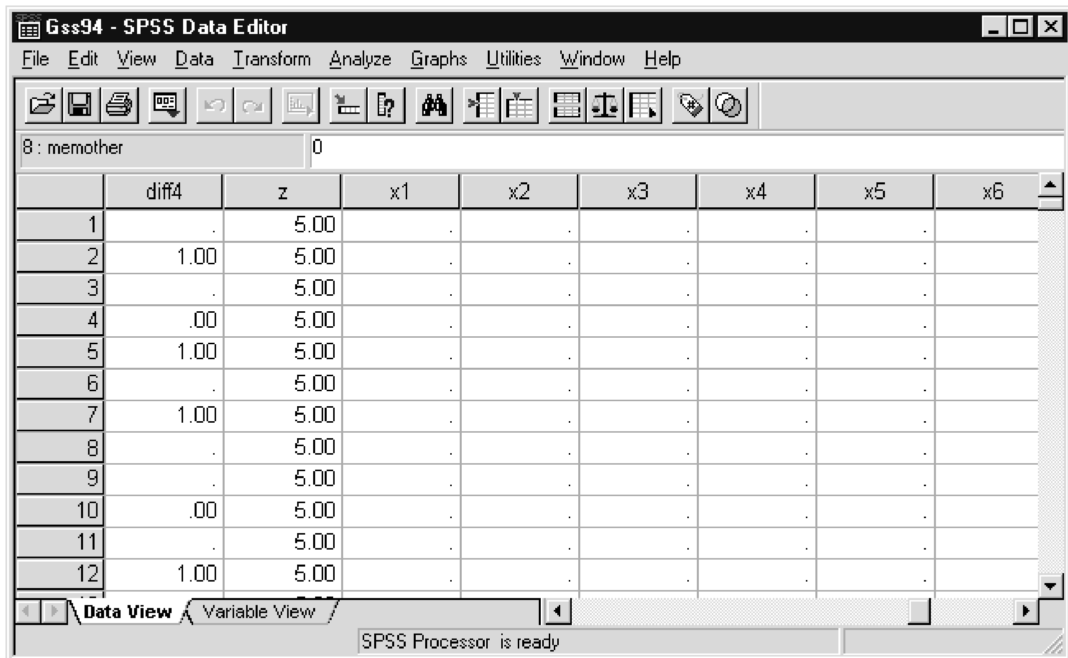
will create 10 new variables with names from X1 to X10, all initialized to system-missing. To illustrate this point

Switch to the **CHAPT2 - SPSS Syntax Editor** window

Click **VECTOR X(10).**, then click the **Run** button

Go to the **Data Editor** (click Goto Data tool)and scroll to the **last columns**

Figure 2. 5 Data Editor with Variables X1 to X10



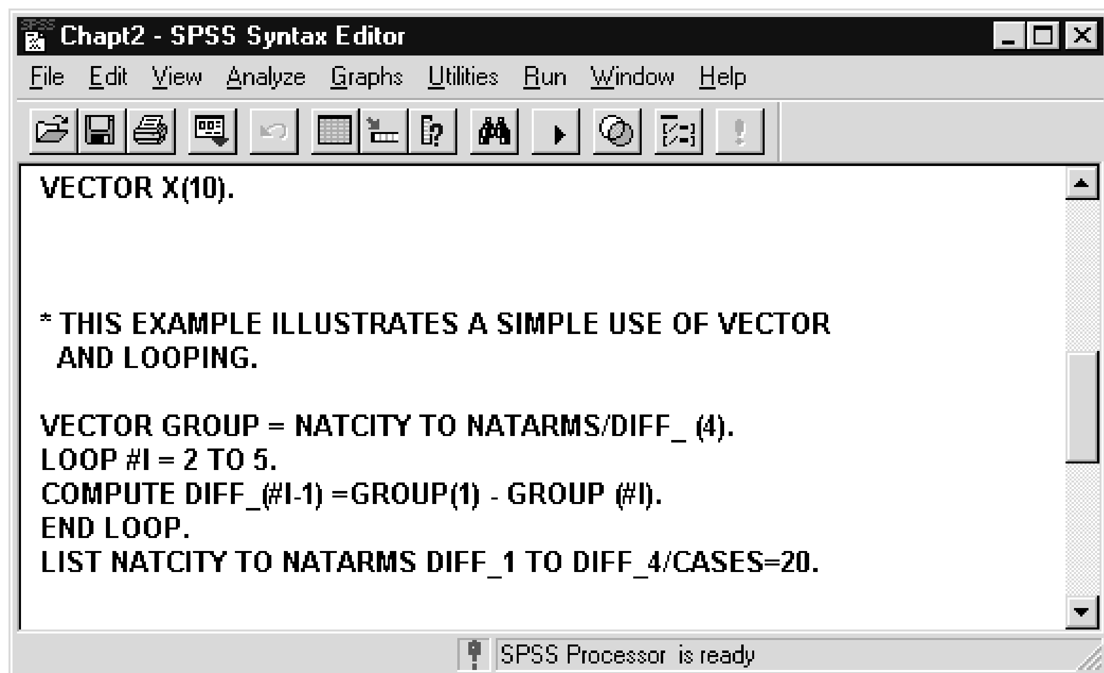
	diff4	z	x1	x2	x3	x4	x5	x6
1	.	5.00	.	.	.	.	.	
2	1.00	5.00	.	.	.	.	.	
3	.	5.00	.	.	.	.	.	
4	.00	5.00	.	.	.	.	.	
5	1.00	5.00	.	.	.	.	.	
6	.	5.00	.	.	.	.	.	
7	1.00	5.00	.	.	.	.	.	
8	.	5.00	.	.	.	.	.	
9	.	5.00	.	.	.	.	.	
10	.00	5.00	.	.	.	.	.	
11	.	5.00	.	.	.	.	.	
12	1.00	5.00	.	.	.	.	.	

The ten new variables all have system-missing values for each case.

A more interesting use of a vector is illustrated by the syntax shown in Figure 2.6. In the DO REPEAT example we compared the value of one spending variable to another to see if responses were identical. We can accomplish a similar task with vectors and loops. In this instance we wish to compare the responses on the variable NATCITY to responses on four other variables (NATCRIME, NATEDUC, NATRACE, AND NATARMS). And instead of creating a new variable that indicates whether the response on NATCITY is identical or not to the other four variables, we will compute the difference.

Switch to the **CHAPT2 - SPSS Syntax Editor** window
Scroll down to the program shown in Figure 2.6

Figure 2.6 Program with Vector and Loop to Compute Differences Between Variables



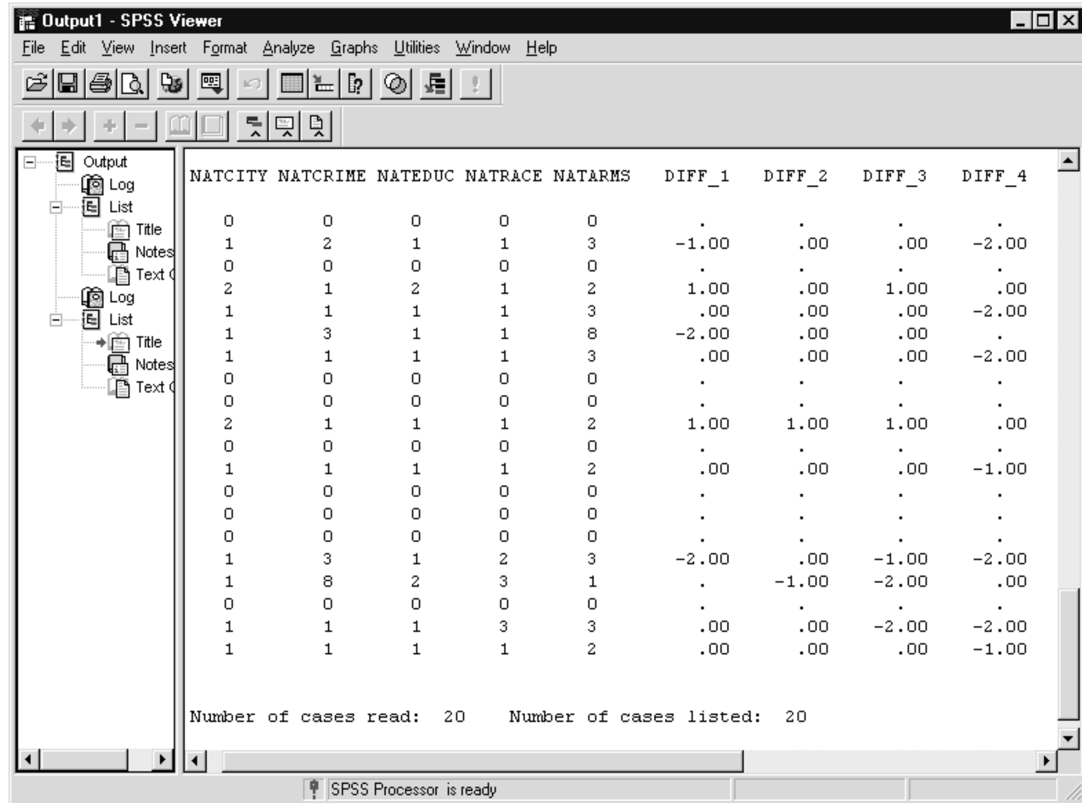
The VECTOR command creates two new vectors. GROUP is composed of the five variables from NATCITY to NATARMS (again, they must be contiguous). DIFF_ has four elements and so creates four new variables, DIFF_1, DIFF_2, DIFF_3, and DIFF_4. We will place the difference for each pair in this vector.

The loop increments by 1 but begins at 2 rather than 1. It loops until 5 (or a total of four times) because there are four variables to compare to NATCITY. On the first pass through the loop, the COMPUTE command compares NATCITY (the first element of GROUP) to the second element of GROUP (NATCRIME) and puts the difference in DIFF_(1). And so on for three other iterations.

Highlight all the lines from **VECTOR GROUP** to **LIST**

Click the **Run** button

Figure 2.7 List Output Showing DIFF_1 to DIFF_4



NATCITY	NATCRIME	NATEDUC	NATRACE	NATARMS	DIFF_1	DIFF_2	DIFF_3	DIFF_4
0	0	0	0	0	.	.	.	.
1	2	1	1	3	-1.00	.00	.00	-2.00
0	0	0	0	0	.	.	.	.
2	1	2	1	2	1.00	.00	1.00	.00
1	1	1	1	3	.00	.00	.00	-2.00
1	3	1	1	8	-2.00	.00	.00	.
1	1	1	1	3	.00	.00	.00	-2.00
0	0	0	0	0	.	.	.	.
0	0	0	0	0	.	.	.	.
2	1	1	1	2	1.00	1.00	1.00	.00
0	0	0	0	0	.	.	.	.
1	1	1	1	2	.00	.00	.00	-1.00
0	0	0	0	0	.	.	.	.
0	0	0	0	0	.	.	.	.
0	0	0	0	0	.	.	.	.
1	3	1	2	3	-2.00	.00	-1.00	-2.00
1	8	2	3	1	.	-1.00	-2.00	.00
0	0	0	0	0	.	.	.	.
1	1	1	3	3	.00	.00	-2.00	-2.00
1	1	1	1	2	.00	.00	.00	-1.00

Number of cases read: 20   Number of cases listed: 20

Where a case has valid values for the spending variables, we can see that SPSS created the four new DIFF_ variables measuring the difference between NATCITY and the other four spending items. It would be straightforward to create additional COMPUTE statements to compare all other possible pairs.

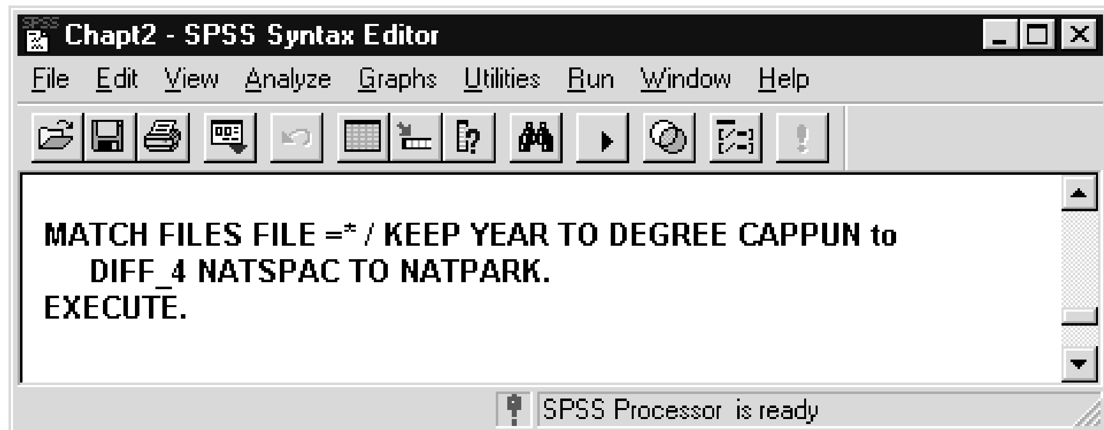
We have used the LIST command to check the operation of SPSS in two of the examples. Checking to see whether syntax has done what you expected it to do is very important when doing SPSS programming. The SUMMARIZE command, available through the menus, can do what LIST does and more, but LIST is easier to type and less complicated when using syntax.

ANALYSIS TIP: REORDERING VARIABLES

A set of variables must be contiguous when placing them into a vector. What can you do if that is not true in an existing file? Perhaps the easiest method to rearrange variables is to use the trick of matching a file to itself. Figure 2.8 displays syntax from CHAPT2.SPS that illustrates this technique.

Switch to the **CHAPT2 - SPSS Syntax Editor** window
Scroll down to the Match Files example

Figure 2. 8 Match Files Program



Normally, MATCH FILES is used to match one file to another. However, here the working data file (referenced by an asterisk on the FILE subcommand) is matched to itself because no other file is named. Usually files are matched using one or more link variables (for example, ID number), but here it is not necessary since we match one file to itself. The key portion of the MATCH FILES command is the KEEP subcommand, where we list the variables we wish to retain in the order we want them to appear in the Data Editor. The EXECUTE command is required because MATCH is a transformation, not a procedure.

Highlight the lines from **MATCH to EXECUTE**

Click the **Run** button

Switch to the **Data Editor** window and scroll to the **last columns**

Figure 2.9 Data Editor with Spending Variables moved to the End

Gss94 - SPSS Data Editor

File Edit View Data Transform Analyze Graphs Utilities Window Help

8 : x7

	diff_4	natspac	natenvir	natheal	natcity	natcrime	nateduc	natrace
1	.	0	0	0	0	0	0	
2	-2.00	3	1	1	1	2	1	
3	.	0	0	0	0	0	0	
4	.00	3	2	1	2	1	2	
5	-2.00	2	2	1	1	1	1	
6	.	3	1	1	1	3	1	
7	-2.00	3	1	1	1	1	1	
8	.	0	0	0	0	0	0	
9	.	0	0	0	0	0	0	
10	.00	3	1	1	2	1	1	
11	.	0	0	0	0	0	0	
12	-1.00	3	1	1	1	1	1	

Data View / Variable View

SPSS Processor is ready

The KEEP subcommand named the spending variables last, so they have been moved to the last columns in the Data Editor.

SUMMARY

We reviewed the types of SPSS commands, the three types of SPSS programs, and briefly reviewed data definition. We then discussed some of the key programming techniques in SPSS, including the use of loops, the creation of vectors, the processing of conditional statements (DO IF), and the creation of repeating elements (DO REPEAT). These techniques will be used repeatedly in SPSS programming. There are a few other important programming techniques that you will see in later chapters when the need arises. We turn in Chapter 3 to the handling of complex data files.

Chapter 3 Complex File Types

Topics

Introduction
ASCII Data and Records
File Types
Syntax Basics
Data File Structure
Reading a Mixed File
Errors in the Data
Grouped File Type Without Record Information

INTRODUCTION

Most SPSS users find that the standard DATA LIST command is sufficient to read the great majority of the data files they normally encounter. This is because most data files are rectangular, i.e., they contain the same number of records per case, the definition of a case is consistent throughout the file, and the variables to be defined are identical for each case. There are, however, situations in which the above conditions do not hold. One example is a file at a medical center with two types of records, one for inpatients and one for outpatients, with identical variables located in different column positions on each type of record, and some variables unique to each type of patient. A standard DATA LIST cannot correctly read such a file and create a separate case for each patient type.

To handle such a data file and any other that is non-rectangular, SPSS supplies two general solutions. The first is to use a FILE TYPE command, which allows the use of predefined file types that read grouped, mixed, or nested files. The second solution is to allow the user to take complete control of the process of reading data with an INPUT PROGRAM command. This chapter discusses the use of file types; the next chapter covers input programs. A third solution is to read the file with a standard DATA LIST command, then use other programming techniques to restructure the file, such as VECTOR and LOOP. We will also illustrate this approach in subsequent chapters.

Before reviewing the various file types, we need to discuss some background information.

ASCII DATA AND RECORDS

SPSS assumes that complex files are in ASCII format so that they can be read with a DATA LIST command (within the complex file types). Files that are stored in a spreadsheet or database format cannot be read directly by SPSS with these techniques. In that case, you have two options. You can write out an ASCII file from the other software and then read it into SPSS with a complex file definition. Or you can read the file into SPSS as you normally would, temporarily creating a working file with an incorrect format for analysis. You can then use various programming techniques to restructure the file.

The use of complex file types requires an understanding of a *record*. For SPSS, a record refers to a physical line of data in an ASCII data file. Technically speaking, a record ends with a carriage return and a line feed (these are invisible to users in most software). In practice, if you open a data file in a text editor, such as Notepad, each line will correspond to a record in the data file. However, this is not always the case in word processing software that wraps lengthy lines, so be careful when dealing with a file for which you do not have a codebook that lists record length.

It is common to have several records for each case you plan to create in the final SPSS data file or to have several cases on one physical record. Understanding what constitutes a record and what the case definition should be in the final SPSS file is part of the art of successful data input programming.

In general, ASCII text data files can be in either fixed or delimited format. Because complex file types must be able to locate case and/or record variables, though, complex data should be stored in a fixed-format ASCII text file.

FILE TYPES

The three available file types within the FILE TYPE command are:

Grouped: This is a file in which all records for a single case are located physically together. Each case usually has one record of each type. Each record should have a case identification variable. This type of file is often identical to a standard rectangular file, the difference being that a grouped file type allows additional checking for errors because of missing and out-of-sequence records, since SPSS normally assumes that the records are in the same sequence within each case.

Mixed: This is a file in which each record type defines a case. Some information may be the same for all record types but can be recorded in different locations. Other information may be recorded only for specific record types. Not all record types need be defined, so this is often a very efficient method to read only part of a data file.

Nested: This is a file in which the record types are related to each other hierarchically. An example is a file containing school records and student records, where all the students attending one school have their records placed together after the school record. Usually the lowest level of the hierarchy, the student in this example, defines a case. Information from the higher-level records, perhaps overall GPA at the school, is usually spread to the lower-level record when the case is defined. All record types that form a complete set should be physically grouped together, with an optional case identifier on each record. It is worth noting that record types can be skipped when reading the data, resulting in the creation of cases at a higher level in the hierarchy.

SYNTAX BASICS

Complex file type programs are begun by the command FILE TYPE and closed with the command END FILE TYPE. These two commands enclose all definitional statements. One of the three keywords GROUPED, MIXED, or NESTED must be placed on the FILE TYPE command. The commands that define the data must include at least one RECORD TYPE and one DATA LIST command, though it is common to have several. One set of RECORD TYPE and DATA LIST commands is used to define each type of record in any data file. The definition of a case, again, depends upon which FILE TYPE is specified.

The RECORD subcommand is required and names the column location of the record identification information and, optionally, the variable that will store this information. A CASE subcommand is also available (and required for a grouped file) that specifies the name and location of the case identification information.

This syntax illustrates the basic structure,

```
FILE TYPE  (Grouped, Mixed, or Nested)  FILE = 'Your File' /
     RECORD = RECID  4    CASE = ID 1-3.
RECORD TYPE 1.
DATA LIST / your variables and column locations here.
RECORD TYPE 2.
DATA LIST / more variables here.
etc.
END FILE TYPE.
```

All three file types have subcommands available that warn the user when records and cases are encountered that don't meet the definitions of the file type, record, and case. This warning can include situations when records are missing.

After the FILE TYPE--END FILE TYPE structure is processed, a rectangular active file is created, no matter the original structure of the raw data file.

DATA FILE STRUCTURE

To further illustrate the three types of files, we display samples of data files that can be read as grouped, mixed, or nested files.

GROUPED DATA

A grouped data file often looks identical or very similar to a standard rectangular data file. However, a grouped file often has one or more of the following problems:

1. A different number of records for each case
2. Records out of order
3. Records with the wrong record number
4. Duplicate records

These situations all mean that SPSS will not read the file successfully with a simple DATA LIST command. The structure of a simple grouped data file is shown in Figure 3.1.

Figure 3.1 Grouped Data File for Hospital Patients

```
Case ID   Record ID

1  1  John Smith Male   45  Evanston  Blue Cross
1  2  X-Ray   3/04/97   456.34
1  3  Blood   3/05/97    67.40
2  1  Mary Doe   Female 33  Lockport  Blue Cross
2  2  X-Ray   5/23/97   435.12
```

These data are from a hospital and contain information on tests and procedures administered to each patient. Each patient's data begins with a record that lists identifying information. The second and subsequent records include information on a test that was given, the date of the test, and the cost. Each record after the first defines a test, but we would like the case definition to be a patient. The problem is that a different number of tests is given to each patient, so we cannot specify the same number of records for each patient.

A CASE subcommand is required on the FILE TYPE command in addition to the RECORD subcommand. Records with a missing or incorrect case identification information cannot be corrected and placed with the correct case, but SPSS will warn you about the problem.

All defined variable names for a grouped file must be unique because multiple records will be put together to form one case. By default all

instances of missing, duplicate, out of range (called "wild") or out-of-order records will result in warnings from SPSS.

MIXED DATA

A mixed raw data file looks quite different than a rectangular data file. Again, a MIXED file type is used when each record type defines a separate case (though not all record types need be defined). Figure 3.2 depicts a portion of the file MIXED.DAT that contains job information on employees from a large company.

Figure 3.2 Mixed Data File For Employees

Record ID

Salary

```
189 1 54 1 34488 88 2 3 4
190 1 35 1 50647 20 1 0 4
191 1 34 0 74095 25 6 2 3
198 1 45 1 25043 56 9 3 2
200 1 64 1 82600 97 5 1 1
201 1 31 1 88490 50 3 3 3
210 1 35 0 93527 57 2 4 4
  1 2 65 1 2 7 2 21 97236
  2 2 28 1 4 9 0 56 24177
  6 2 61 1 2 3 7  6 79711
  9 2 49 1 4 6 1 80 37350
 10 2 42 0 4 8 8 87 50995
 14 2 23 1 3 1 9 23 44761
 19 2 53 1 3 9 2 38 99635
 21 2 23 0 1 9 3 83 42432
 23 2 29 0 1 8 6 23 19647
```

Columns 2-4 contain an identification number for each employee. This is necessary for the company but not important for a FILE TYPE MIXED definition. Column 6 contains the required record identification information, in this case either a 1 or 2 (there is also a record number 3 not shown). The company had three separate record-keeping systems for employee information for each division. The data from all three have recently been placed in one file for reporting purposes.

A standard DATA LIST cannot be used to read this file because some of the same information is in a different location for each record type. Salary is recorded in a different location for each record type, and other variables are not recorded in each system. We will attempt to read this file in the first example.

NESTED DATA

A nested data file also looks quite different than a rectangular data file. A FILE TYPE NESTED command is used when the records in a file are hierarchically related. One example is a file with two types of records, one for each department in a company, and one for each of the employees in that department. All the employee records for one department are placed consecutively together, after the record for the department in which they are located and before the next department record.

The variable names on all the records must be unique because one record of each type will be grouped together to form a case. Since not all record types need be mentioned on the RECORD TYPE command, it is possible to define a case at a higher level in the hierarchy, e.g., a department rather than an employee. In fact, a case can be defined at any level in the hierarchy of record types. Figure 3.3 depicts a nested data file for a school district.

Figure 3.3 Nested Data File for School District

Record ID

```
1 101 Jackson      2.89 485 501
2  3 4  2.78
3 2512.57  442 448 1 0
3 2432.94  515 484 1 0
3 2471.28  453 298 1 0
3 1792.83  336 768 0 0
3  443.33  212 465 0 0
3  492.84  601 476 1 1
3  994.00  343 367 0 0
3 1873.47  598 431 0 0
3  433.23  428 517 0 1
3 1853.39  459 369 1 1
3  582.70  415 524 0 0
2  5 1  2.97
3  752.30  422 433 1 0
3 1672.42  229 314 1 1
3   22.44  264 313 0 0
```

Student Records

The file contains data on the performance of high school students organized by homeroom and school. It contains three types of records:

Record 1: The high school record contains identifying information on the school plus the school's overall GPA, SAT verbal and SAT math scores.

Record 2: The homeroom record contains identifying information on the homeroom plus the average GPA for all students in the homeroom.

Record 3: The student record contains identifying information including the student's sex and academic track, plus GPA, SAT verbal and SAT math scores.

There is only one variable in common for all three records, the record identifying information in the first column.

The data file has no case identifying information, which is typical of real-life situations. That isn't a problem since SPSS simply stores the higher-level record information, spreading it to each record 3 when it creates a case for each student, retaining this information until another record type 1 is encountered. SPSS can still successfully create the cases even when intermediate-level records are missing (a homeroom record, in this instance).

READING A MIXED FILE

We will read the employee mixed data file from Figure 3.2 (named MIXED.DAT) into SPSS and create a rectangular file with a case for each employee.

Here is a codebook table for the three types of employee record systems.

VARIABLE	RECORD 1 LOCATION	RECORD 2 LOCATION	RECORD 3 LOCATION
ID	2-4	2-4	2-4
RECORD ID	6	6	6
AGE	8-9	8-9	8-9
SEX	11	11	11
SALARY	13-17	22-26	13-17
TENURE	19-20	19-20	19-20
JOBCODE	22	17	not recorded
LOCATION	24	15	22
JOBRATE	26	13	not recorded

Not only are variables like SALARY recorded in different locations, but two variables, JOBCODE and JOBRATE, were not recorded on record 3, which was the oldest record-keeping system. When the same variable, e.g., AGE, is defined by more than one record type, the format type and length should be the same on all records. SPSS uses the first appearance of the variable for the active file dictionary.

The appropriate commands to read this file are included in Figure 3.4 and are in the file MIXED.SPS.

Click on **File..Open..Syntax**
If necessary, switch directories to **c:\Train\ProgSynMac**
Double-click on **MIXED**

Figure 3.4 Mixed File Type Program

```
FILE TYPE MIXED FILE ='C:\Train\ProgSynMac\MIXED.DAT' RECORD=SYSTEM 6.


RECORD TYPE 1.
DATA LIST  / ID 2-4 AGE 8-9 SEX 11 SALARY 13-17 TENURE 19-20
          JOBCODE 22 LOCATION 24 JOBRATE 26.

RECORD TYPE 2.
DATA LIST / ID 2-4 AGE 8-9 SEX 11 JOBRATE 13 LOCATION 15 JOBCODE 17
         TENURE 19-20 SALARY 22-26.

RECORD TYPE 3.
DATA LIST / ID 2-4 AGE 8-9 SEX 11 SALARY 13-17 TENURE 19-20  LOCATION 22.

END FILE TYPE.

VARIABLE LABELS  SYSTEM 'System of Record Keeping'
             JOBRATE 'Overall Job Performance'
             LOCATION 'Department'
             TENURE 'Months in current job'.

VALUE LABELS SYSTEM 1 'Latest version' 2 'Middle version'
               3 'Earliest version'/
          SEX 0 'Male' 1 'Female'/
          JOBRATE 1 'Poor' 2 'Fair' 3 'Good' 4 'Excellent'.

FREQUENCIES SYSTEM.
```

The command FILE TYPE begins the file definition and puts SPSS into an input program state. The MIXED subcommand tells SPSS that this is a mixed data file. The data file is named here, not on the DATA LIST commands that follow. The only other required subcommand is RECORD to specify the record identification variable. The equal sign is not required following RECORD or FILE. The record variable is in column 6 and will be named SYSTEM. For the employee data it is important to retain information that tells us under what record system the data were created because of duplicate IDs under each system; often, though, the record variable doesn't need to be retained in the final file. In that case, it can be declared a scratch variable by beginning its name with "#".

Each employee data system, corresponding to a type of record in the data file, gets its own RECORD TYPE command. The value specified on the command (1, 2, or 3) refers to an actual value in the file MIXED.DAT in the record identification position, here column 6 (refer to Figure 3.2). The DATA LIST command following each RECORD TYPE command defines the variables for that record type. Notice how SALARY is in columns 13-17 for record types 1 and 3 but in columns 22-26 for record type 2.

An optional subcommand on the FILE TYPE command is WILD, which tells SPSS to issue a warning when it encounters undefined record types in the data file. The default is NOWARN, so SPSS simply skips all record types not mentioned and does not display warning messages.

The input program state ends with the END FILE TYPE command, which is followed by labeling commands and then a Frequencies command. FILE TYPE--END FILE TYPE are not procedures and do not cause the raw data file to be read, so they must be followed by either an EXECUTE command or another procedure.

To run all the syntax

Click **Run..All**

SPSS displays the commands in the Viewer window (not shown) and then the frequency table for SYSTEM. We can see that there are 212 employees in the file, created from 212 records, and that there are 52 of record type 1, 122 of record type 2, and 38 of record type 3 in the data file. All the information on, for example, salary, has now been placed in one column despite its two different locations in the data (if you wish, switch to the Data Editor to verify this).

Figure 3.5 Frequency Table for System

System of Record Keeping

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Latest version	52	24.5	24.5	24.5
	Middle version	122	57.5	57.5	82.1
	Earliest version	38	17.9	17.9	100.0
	Total	212	100.0	100.0	

ERRORS IN THE DATA

We will illustrate what occurs with undefined record types by once again reading the file MIXED.DAT. Because warning messages are turned off by default, we were unaware that there are in fact 213 records, or employees, in the file. However, the 213th case has an error in its record type, as shown in Figure 3.6.

Its record type should be a “3” but is instead a “4.”

Figure 3.6 Error in MIXED.DAT

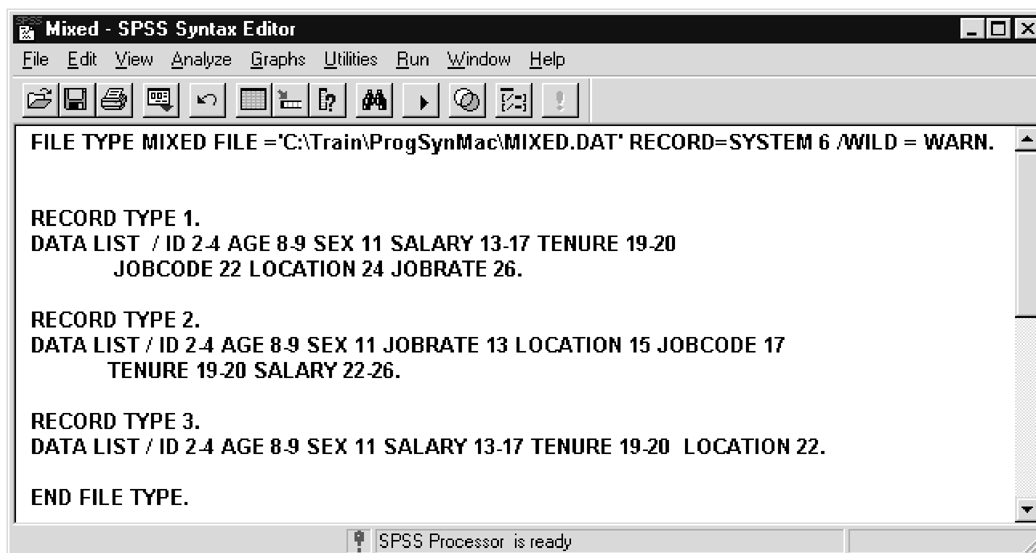
```
171 3 33 1 18768 85 1
177 3 28 0 89638 50 1
178 3 32 1 39237 45 6
183 3 36 0 39253 96 7
185 3 34 0 64370 83 3
196 3 27 1 65352 80 5
202 3 60 0 24010 55 6
204 3 27 1 65352 80 5
211 3 30 0 82358  5 7
319 4 43 1 45678  7 5
```

SPSS skipped this employee’s record because it was not defined on a RECORD TYPE command, but didn’t warn us. Let’s tell SPSS to do that, then reread the file.

Click **Window..MIXED - SPSS Syntax Editor** to return to the syntax file

Add the subcommand **/WILD=WARN** to the end of the FILE TYPE command, but before the period (.)

Figure 3.7 Modified File Type Command to Add Warnings From SPSS



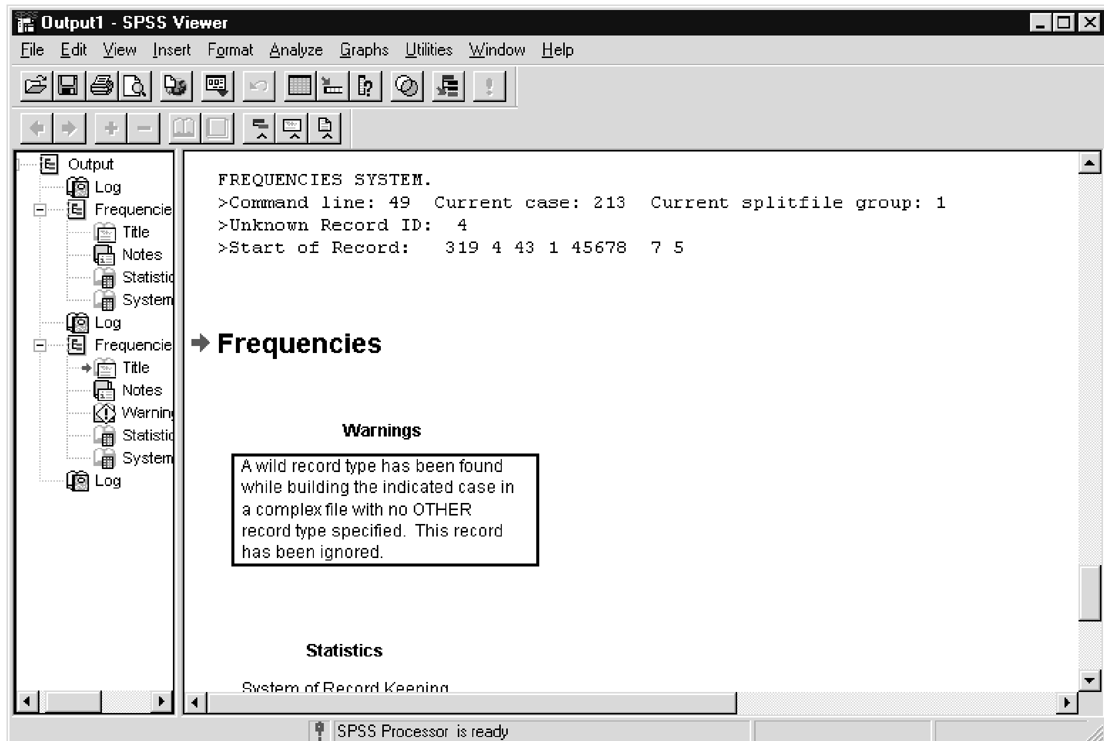
After you have carefully added this subcommand, rerun all the commands by

Clicking **Run..All**

When SPSS switches to the Viewer, you will now see a warning message and a note in the log under the FREQUENCIES command, as shown in Figure 3.8. The warning message is clear, telling us that the record type (4) was ignored when building the file. You can verify this by looking at the frequencies output, which lists only 212 cases.

The exact position of the problem is noted in the message that begins with “Command line:”. The critical information is that it was on case 213 that SPSS encountered an unknown record ID, whose value is 4. SPSS also conveniently lists the actual line of data from the file MIXED.DAT for reference. Obviously, warnings can be helpful in finding and fixing errors in data entry or definition.

Figure 3.8 Warning Messages for Undefined Record Type 4



It is now possible to use the file created via the FILE TYPE MIXED command to report on either the total group of employees, or differences between employees across record-keeping systems.

Analysis Tip

If you plan to read a file with known errors, you might think that you want to be warned every time there is a problem defining an SPSS data file. However, this is not always the case. If it is a large data file with many errors, SPSS could possibly generate hundreds, even thousands, of warning messages. It is unlikely that you will care to scroll through all that output. In recognition of this, the maximum number of warnings SPSS will display has been set relatively low, to a value of 10. When you do want to see more warnings, use the SET command with this syntax:

SET MXWARNS = 100. (or to whatever value is appropriate)

GROUPED FILE TYPE WITHOUT RECORD INFORMATION

Reading either a grouped or nested file into SPSS isn't much different than reading a mixed file in terms of the syntax. However, one situation that causes problems yet is still relatively common, and therefore worth exploring, is when you wish to use FILE TYPE GROUPED but don't have a record identification variable. This is fairly common, especially because any rectangular data file can be read with either a standard DATA LIST or via a FILE TYPE GROUPED format. The advantage of the latter is that SPSS will fix any problems with out-of-order records and read the file correctly if there are missing records. However, a record type variable is needed in each instance, and you may not have created one for what you knew was a standard rectangular file format.

Our example here is a little more complicated. Figure 3.9 displays a small data file that stores information on students in a statistics class and their scores on each of three assignments. There is an identification variable for the student in column 2, but no numeric record identifier to tell SPSS that the first record has information on the first quiz, the second record on the first homework assignment, and the third record on the first test. Moreover, student 2 did not complete the first homework assignment and so only has two records, which is the real problem.

If this file had been created with one record for each student, and each assignment in a separate column, it would be a straightforward task to read it into SPSS.

Figure 3.9 Grouped Data File

```
 1  quiz1 23
 1  hmwk1 34
 1  test1 84
 2  quiz1 18
 2  test1 55
 3  quiz1 25
 3  hmwk1 36
 3  test1 91
```

Note

We should point out that the score type (quiz1, etc.) field can be used as a record type identifier, although it is a string field. However, we will ignore this in order to demonstrate another method of reading the file.

What we want to do is read this data file and create three cases, one for each student. We also want to create three variables, one for each type of assignment, and just as important, we want SPSS to realize that the second student's second record has his score for the test, not the homework, assignment.

There are two methods to approach this problem without using a more complex INPUT PROGRAM command sequence.

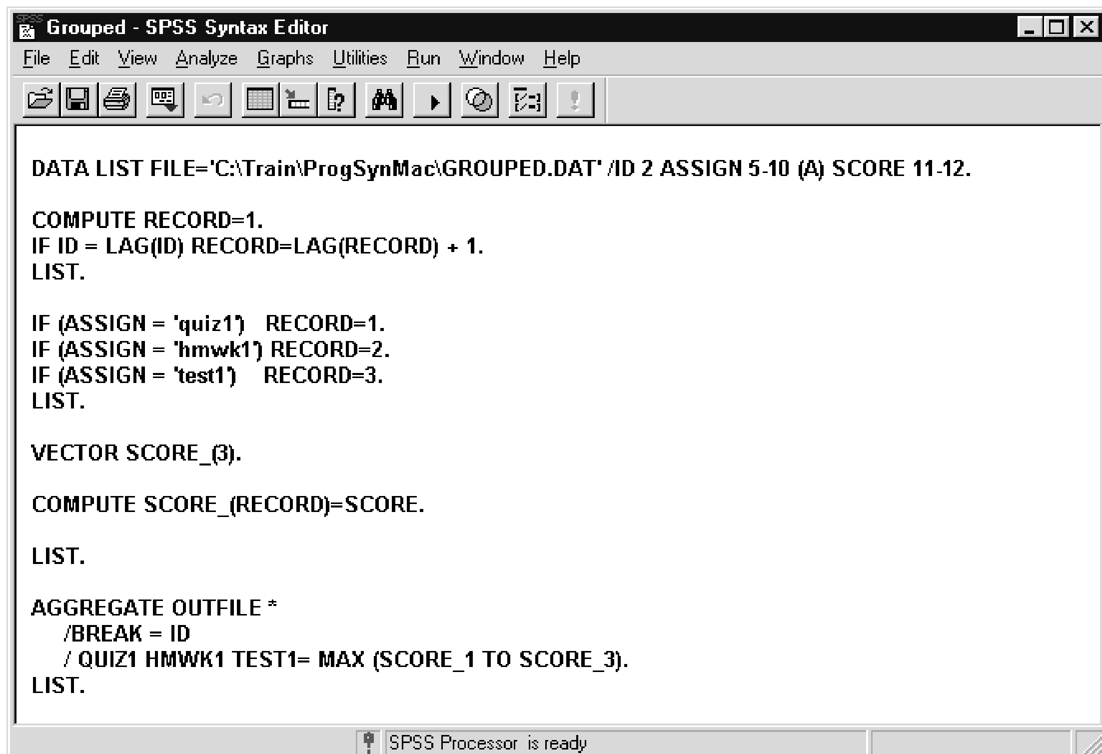
1) Read the data into SPSS, create a record identifier, write the data back out as an ASCII file, then read it back in using FILE TYPE GROUPED.

2) Read the data into SPSS, create a record identifier, then manipulate the data to create the necessary variables.

The latter choice is clearly preferable because it requires fewer passes of the data file. The file GROUPED.SPS contains the commands to implement the second method.

Click on **File..Open..Syntax** (move to **c:\Train\ProgSynMac** folder)
Double-click on **GROUPED**

Figure 3.10 Program to Read a Grouped File



The DATA LIST command reads GROUPED.DAT as if it were a rectangular file. The interesting techniques in this program are what happens after the file is read.

First, we create a RECORD variable with the LAG function. Initially, RECORD is set to 1 for each case. Then, when the current case has the same ID as the previous case, RECORD is set equal to the value of RECORD for the previous case plus one. When the previous case was a different student, then this statement is not executed and RECORD remains at 1, i.e., it is reset to 1 for each student's first record. This creates the desired record identification variable. To see this

Highlight the lines from **DATA LIST** to the first **LIST** command

Click on the **Run** button

Figure 3.11 List Output Showing Record ID

```
ID ASSIGN SCORE   RECORD

 1 quiz1    23      1.00
 1 hmwk1    34      2.00
 1 test1    84      3.00
 2 quiz1    18      1.00
 2 test1    55      2.00
 3 quiz1    25      1.00
 3 hmwk1    36      2.00
 3 test1    91      3.00
```

Analysis Tip

When creating programs that read or transform data, it is very helpful in debugging your programs to list out the data after an action or series of actions is executed. If you don't do this, then it will be very difficult to figure out where things went wrong. Following this advice, the LIST command has been placed at four spots in the program. It is better to err on the side of excess here.

Our problems are not yet solved, though. Student 2 didn't complete the homework, but the value of RECORD for his test1 score is 2, not 3. The set of IF statements fix this problem. They assign the correct value of RECORD for each type of assignment, so that student 2's test1 score will now be listed as record type 3.

Switch to the **GROUPED - SPSS Syntax Editor** window
Highlight the lines from **IF** to the next **LIST** command

Click on the **Run button**

In the Viewer (not shown) we can now see that the value of RECORD for the second line (or case) for student 2 has been changed to a 3.

One task remains, and that is to take these 8 cases in the Data Editor and turn them into 3 cases. As mentioned above, we could write this file out and then read it back into SPSS, but that is cumbersome. A better method is to use the AGGREGATE command. For analysis purposes, it is best to create a separate variable or column for each assignment. If we simply aggregate the current working file by student ID, that won't happen. Why?

To create these new variables (three in this instance), we can take advantage of the VECTOR command we saw in Chapter 2. The VECTOR command creates a new vector, SCORE_, with three elements. The COMPUTE statement then assigns the value of SCORE for a case to an element of SCORE_ based on the value of RECORD. In other words, for the first record type (quiz1), SCORE_1 gets the value of SCORE for that case, the quiz1 score. The values of SCORE_2 and SCORE_3 *for that case* are system-missing. For the second record type (for hmwk1) SCORE_2 gets the value of SCORE for that case, and SCORE_1 and SCORE_3 are system-missing.

It's probably easier to see this in action.

Switch to the **GROUPED - SPSS Syntax Editor** window
Highlight the lines from **VECTOR** to the next **LIST** command

Click on the **Run button**

Figure 3.12 List Output Showing Vector SCORE_

```
ID ASSIGN SCORE   RECORD  SCORE_1  SCORE_2  SCORE_3

 1 quiz1    23      1.00    23.00      .        .
 1 hmwk1    34      2.00      .      34.00      .
 1 test1    84      3.00      .        .      84.00
 2 quiz1    18      1.00    18.00      .        .
 2 test1    55      3.00      .        .      55.00
 3 quiz1    25      1.00    25.00      .        .
 3 hmwk1    36      2.00      .      36.00      .
 3 test1    91      3.00      .        .      91.00


Number of cases read:  8    Number of cases listed:  8
```

The values for each assignment or test have been placed in separate variables, with the quiz1 score in SCORE_1, the hmwk1 score in SCORE_2, and the test1 score in SCORE_3. However, there are still three cases for each student, with lots of missing data, and all we need is one case for each student to calculate appropriate statistics.

A file with that structure can be created using the AGGREGATE command, which changes the case base in a file and calculates summary statistics for the new case base. For example, in a file of customers that buy five different products, AGGREGATE can create an SPSS data file where the case base is product (and so there are only five cases), with information such as the number of customers who bought each product, the mean age of customers who bought that product, and so forth.

There are three necessary subcommands for AGGREGATE, as shown in Figure 3.10. The OUTFILE subcommand tells SPSS whether to save the new data file to disk or to make it the current working file in the Data Editor. The asterisk means to replace the current working file with the new one. The BREAK subcommand defines the case base for the new file. By breaking on the variable ID, which here has only three unique values, we will create a file with three cases. The aggregated variables subcommand creates the summary variables in the new file. Its format is

LIST OF NEW VARIABLES = FUNCTION (LIST OF
EXISTING VARIABLES)

where you define new variable names on the left of the expression, an aggregate function on the right, followed by the same number of existing variables that will be used to create the new variables. In our example, we use the MAX (maximum) function to create three variables called QUIZ1, HMWK1, and TEST1, based on the maximum value of SCORE_1 to SCORE_3 for each ID value. Why does this accomplish our task? Could we have used a different function?

To finish the program

Highlight the **AGGREGATE** and **LIST** commands

Click on the **Run button**

Figure 3.13 List Output Showing New Assignment Variables

```
ID    QUIZ1    HMWK1    TEST1

 1    23.00    34.00    84.00
 2    18.00      .      55.00
 3    25.00    36.00    91.00


Number of cases read:  3    Number of cases listed:  3
```

The output from LIST demonstrates that there are only three cases in the new file, one for each student. Three new variables have been created, one for each assignment. This makes it easy to calculate statistics for each assignment. And student 2 has been correctly assigned a missing score for HMWK1. Since AGGREGATE only creates the new summary

variables we define, the variables ASSIGN, SCORE, RECORD, and SCORE_1 to SCORE_3 are gone, which is fine. You may also want to look at the Data Editor (not shown) to see the file format.

Analysis Tip

Unlike the definition of complex file types, every command in this program could have been created from the dialog boxes except VECTOR. As this course is generally concerned with SPSS programming, we instead worked from a Syntax file. Either approach is acceptable, although seeing the syntax often helps your understanding, and it certainly lets you apply the same type of program in the future to another data file.

SUMMARY

We reviewed the types of complex files, their structure, and the SPSS syntax used to read these data files. We illustrated the use of complex file types by reading a mixed data file, then discussed how data errors are handled by SPSS. We then showed how to read a grouped file with an odd structure and no numeric record type information.

Chapter 4 Input Programs

Topics

Introduction
Syntax Basics
Changing the Case Base of a File
End of Case Processing
End of File Processing
Checking Input Programs
Incomplete Input Programs
Reading Files with Missing Identifiers
When Things Go Wrong

INTRODUCTION

There are situations where you encounter non-rectangular raw data files that cannot be read directly with the complex file types provided by SPSS. For those situations, SPSS offers an input program facility, as mentioned in Chapter 2, that has the capability to read essentially any type of ASCII data file. The ability to read a file will at times depend upon the cleverness of the user, as really odd files may require creative solutions.

An input program can also be used to create data that match a target distribution, often for purposes of teaching or illustration. In other words, an input program can create data from nothing (this is the one time that SPSS provides a free lunch, so to speak).

For very large files, input programs also offer great efficiencies, even if the file is a standard rectangular data file. An input program can be used to select only certain cases as the file is read, saving one pass of the data. Or it can concatenate raw data files, saving on having to create SPSS data files of each. And input programs can perform the equivalent functions of a grouped, mixed, or nested file type, but with added flexibility.

The user is in charge of case definition when writing an input program, so careful attention must often be paid to where in the program stream a case should be created. At times, you may also need to tell SPSS when to stop reading the data and create a working data file.

SYNTAX COMPONENTS

The commands INPUT PROGRAM and END INPUT PROGRAM enclose data definition and transformation commands that build cases from input data. At least one file definition command, such as a DATA LIST, must be included in the structure. Essentially any transformation commands can be placed within an input program structure, but no procedures. This means that you can use COMPUTE, IF, DO IF, REPEATING DATA, LOOP, or any of the other transformation commands that may help to create a working data file.

It is very important to understand that SPSS processes the input program commands on a case-by-case basis. This may be hard to grasp intuitively, given that an input program creates the definition of a case as it is executed, but we will illustrate this concept below to provide a concrete example. Without comprehending this point, input programs may remain somewhat mysterious in their operation.

EXAMPLE 1: CHANGING THE CASE BASE OF A FILE

We begin our exploration of input programs with an example that includes vectors and looping. Figure 4.1 displays a small data file. This data file is to be used in an experiment on the effect of a drug on hypertension. Each person's data consist of 15 measurements, recorded in three sets of five numbers. Five different characteristics of each person were recorded at three time points (hence the three records). Thus for the first person, the first five numbers are 1, 2, 3, 4, and 5, for the five measurements at time 1. In this small test file, there are six lines of data for two persons.

Figure 4.1 Data File with Three Records Per Case

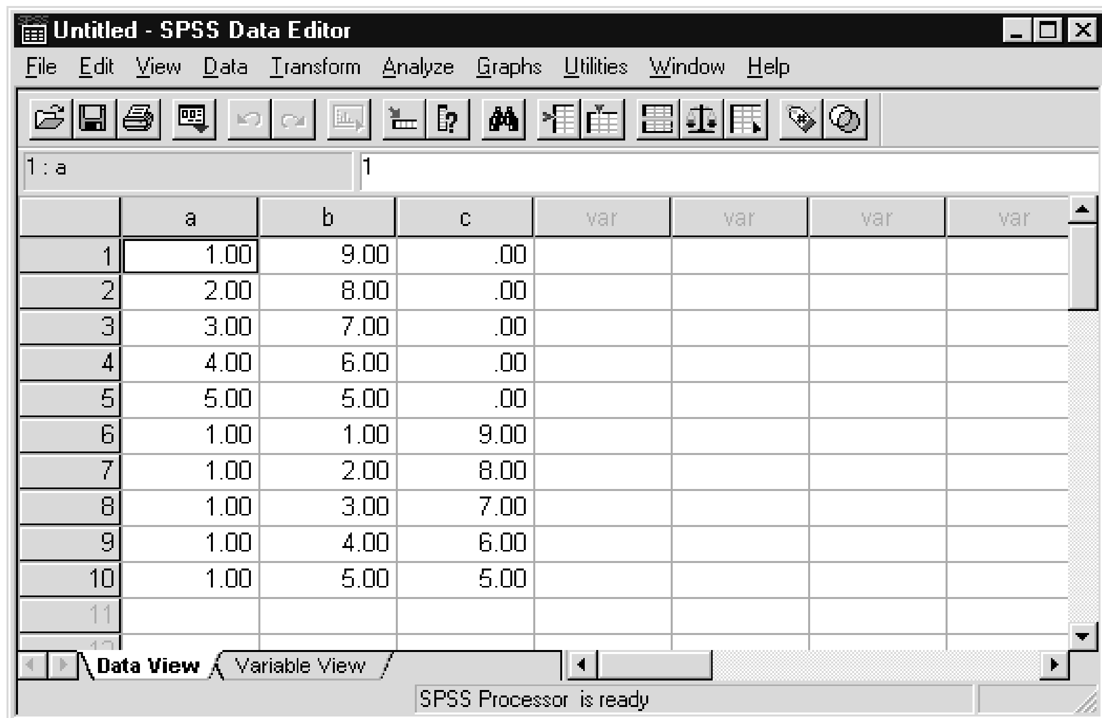
```
12345 ┐
98765 │   One case
00000 ┘
11111
12345
98765
```

However—and this is the problem with the file structure—we wish to flip the data because measurements of each characteristic came in sets of three that logically should be grouped together. Thus, the three measurements for the first characteristic of the first person are not 1, 2, and 3, but 1, 9, and 0. That is, they are the first numbers in each record. The three measurements for the second characteristic are 2, 8, and 0, and so forth for the remaining three characteristics.

When we are done, the SPSS data file should look like Figure 4.2. In the restructured file, the rows have become columns and the columns have become rows, separately for each person. That is our goal. It is usually very important when using an input program to literally sketch out the equivalent of Figure 4.2 to have a clear goal in mind.

Note that typically such files would contain subject and characteristic identifiers, and additional information. We drop these variables to better focus on the data reorganization. However, without such identifiers you cannot tell what is what within the file.

Figure 4.2 The Goal: A Restructured Data File



	a	b	c	var	var	var	var
1	1.00	9.00	.00				
2	2.00	8.00	.00				
3	3.00	7.00	.00				
4	4.00	6.00	.00				
5	5.00	5.00	.00				
6	1.00	1.00	9.00				
7	1.00	2.00	8.00				
8	1.00	3.00	7.00				
9	1.00	4.00	6.00				
10	1.00	5.00	5.00				
11							

To open the program that carries out this task

Click on **File..Open..Syntax**
If necessary, switch directories to **c:\Train\ProgSynMac**
Double-click on **INPUT1**

Figure 4.3 Input Program to Invert a File

Input1 - SPSS Syntax Editor

File Edit View Analyze Graphs Utilities Run Window Help

```
INPUT PROGRAM.

DATA LIST FILE = 'C:\Train\ProgSynMac\INPUT1.DAT' RECORDS=3
                  /#V1 TO #V5 1-5
                  /#V6 TO #V10 1-5
                  /#V11 TO #V15 1-5.

VECTOR VARS_=#V1 TO #V15.

LOOP #I = 1 TO 5.
COMPUTE A=VARS_(#I).
COMPUTE B=VARS_(#I + 5).
COMPUTE C=VARS_(#I + 10).

END CASE.
END LOOP.
END INPUT PROGRAM.

LIST.
```

SPSS Processor is ready

The key points of the program are these:

1) Read the file as it currently exists to create fifteen variables for each person;
2) Use a vector and looping to read the first, sixth, and eleventh variable for each person;
3) Put these three bits of information into three variables in a new case;
4) Do this four more times for each person, to create five cases for the five characteristics being measured of each person.

The case basis can be changed as the file is read because we are still in an input program structure. We will walk through each command of this program to show how it operates.

INPUT PROGRAM: This command begins the input program definition. No other specification is necessary or can be added.

DATA LIST: The command reads the file INPUT1.DAT. This is a standard DATA LIST command, but because there are three records per person, a RECORDS subcommand is used to specify this to SPSS. Fifteen new variables are created, one for each measurement. They are declared scratch variables so that they are not retained on the new file that SPSS will construct. The syntax for the DATA LIST uses a "/" to indicate that SPSS should read the next record in INPUT1.DAT.

VECTOR: This command creates a vector from the fifteen variables.

LOOP: We loop from 1 to 5 (using the scratch variable #I) because we want to create five new cases for each person.

COMPUTES: Each pass through the loop creates three new variables, A, B, and C. For the first value of #I, which is 1, A is equal to the value of #V1. B is equal to #V6, and C is equal to #V11. It is important that you understand why this happens and that this operation puts the values of 1, 9, and 0 in these three variables.

At this point, SPSS has 19 variables created, #V1 to #V15, #I, and A, B, and C.

END CASE: This command tells SPSS to create a case at this point and pass it on to the commands immediately following the input program structure, when the data have all been read. When the case is created, all 16 scratch variables are ignored, so SPSS creates a case with only A, B, and C, as desired.

When SPSS first encounters the END LOOP command, it resets the value of #I to 2 and passes through the three COMPUTE statements and the END CASE command again with that value. Recall from Chapter 3 that looping occurs within a case in SPSS, not across cases. Since the initial DATA LIST statement essentially created a (temporary) case from each set of three records, the five passes through the loop create five new cases from these three records, or five cases per person.

When looping is done for the first person, SPSS goes on to read the next set of three records and do the same for this person, and this will continue until the end of the data file is encountered.

END INPUT PROGRAM: When the physical end of the file INPUT1.DAT is encountered, SPSS then writes out a final version of a working data file to the Data Editor, with only the variables A, B, and C. However, an input program is a transformation and so is not executed until SPSS encounters a procedure or the EXECUTE command. Here, we force execution with the LIST command (an SPSS procedure).

Let's run the complete program.

Highlight all the lines in the program.

Click the **Run Button**

The commands are echoed in the Viewer, and the output from LIST shows that the file has been inverted as desired. The variables A, B, and C for the first case have values of 1, 9, and 0.

Figure 4.4 List Output Showing New File Structure

```
        A        B        C

     1.00     9.00      .00
     2.00     8.00      .00
     3.00     7.00      .00
     4.00     6.00      .00
     5.00     5.00      .00
     1.00     1.00     9.00
     1.00     2.00     8.00
     1.00     3.00     7.00
     1.00     4.00     6.00
     1.00     5.00     5.00


Number of cases read:  10    Number of cases listed:  10
```

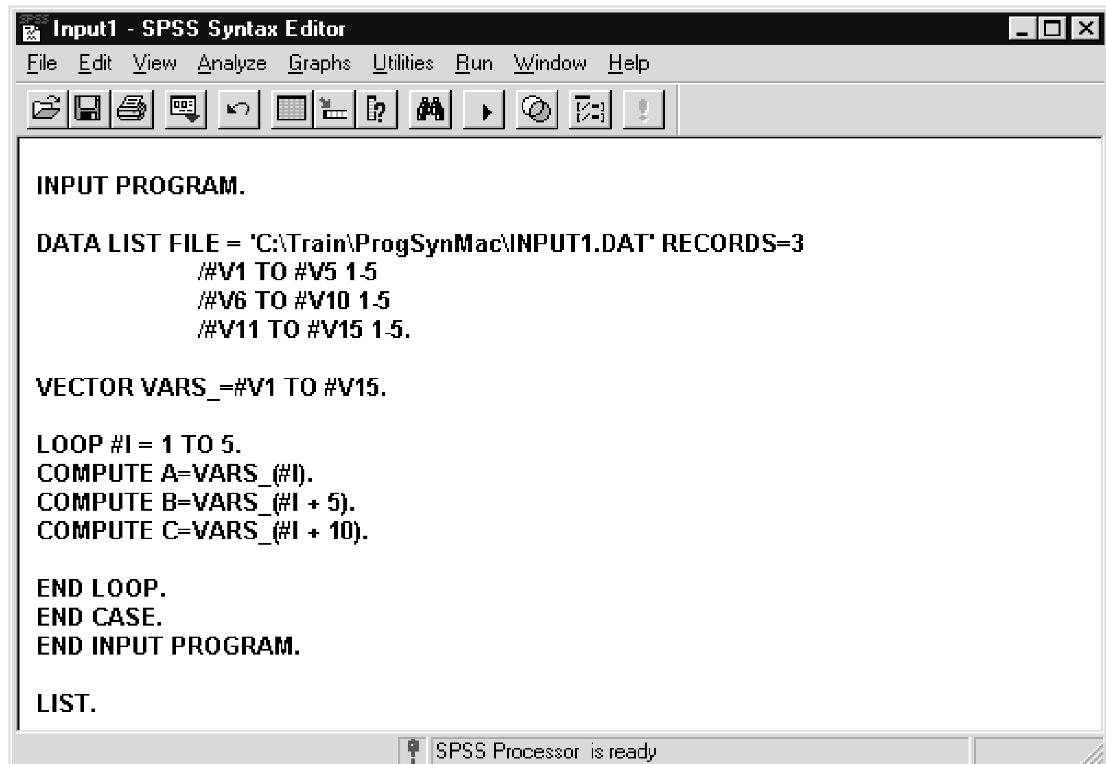
Given the file restructuring capabilities of SPSS, we certainly could have read the data file without an input program, then changed the case base with such commands as AGGREGATE afterwards. But an input program is more efficient than such techniques, especially as a file grows larger. When you have the choice between these two approaches, an input program is usually the better choice.

END OF CASE PROCESSING

A basic problem that many people have when writing input programs is how to properly handle end of case and end of file processing. We can use the program INPUT1.SPS to illustrate the complexities of these decisions.

In Figure 4.5, the program has been slightly modified by switching the order of the END CASE and END LOOP statements.

Figure 4.5 Program With End Loop Before End Case

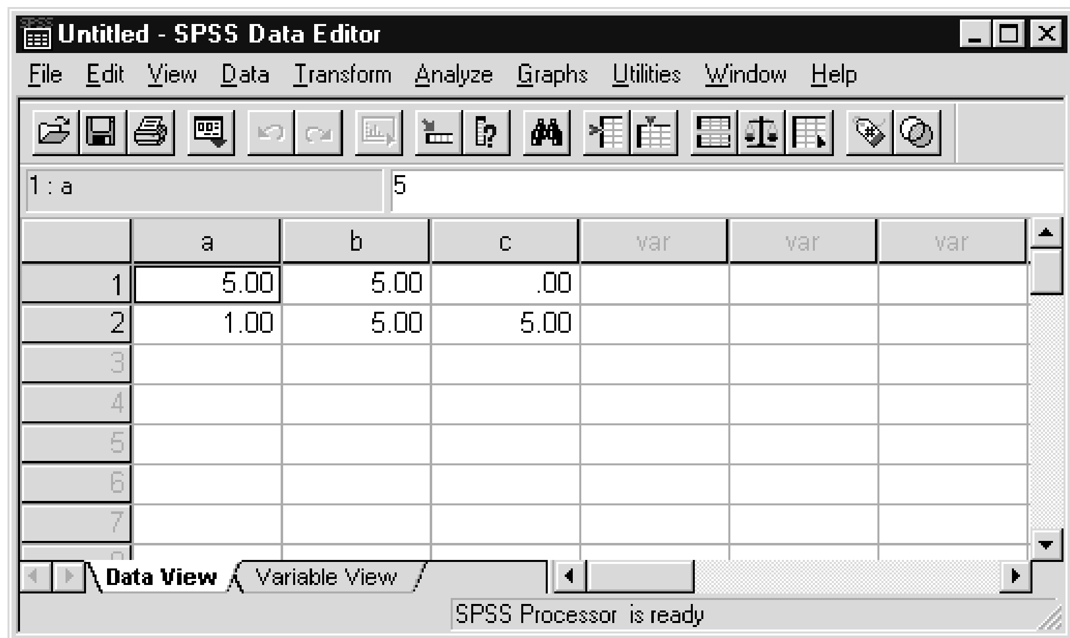


```
INPUT PROGRAM.

DATA LIST FILE = 'C:\Train\ProgSynMac\INPUT1.DAT' RECORDS=3
                 /#V1 TO #V5 1-5
                 /#V6 TO #V10 1-5
                 /#V11 TO #V15 1-5.

VECTOR VARS_=#V1 TO #V15.

LOOP #I = 1 TO 5.
COMPUTE A=VARS_(#I).
COMPUTE B=VARS_(#I + 5).
COMPUTE C=VARS_(#I + 10).

END LOOP.
END CASE.
END INPUT PROGRAM.

LIST.
```

What type of data file will this structure create? How many cases will be created? Why?

The answer is shown in Figure 4.6.

Figure 4.6 New SPSS Data File



What happens is that the new program still tells SPSS to loop five times within a case, but the second loop replaces the values of A, B, and C with new values, the third does the same, and so forth. After the fifth and last loop, the values of A, B, and C for the first case are 5, 5, and 0—the values for the last number in each of the three records. At this point, we then tell SPSS to create a case. Since we effectively didn't change the case basis from what the DATA LIST command created, we end up with only two cases, one for each person. This is not what we intended, and it illustrates the importance of correctly placing the END CASE command.

Fortunately, it's relatively easy to experiment in this fashion and usually straightforward to see the effect of command placement.

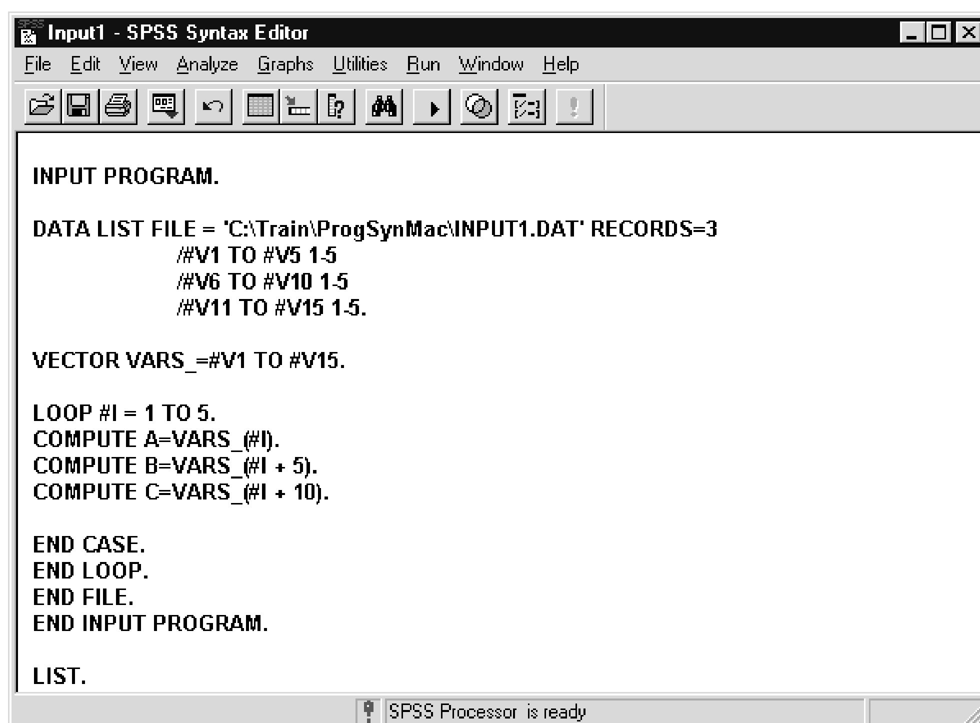
END OF FILE PROCESSING

When you write an input program, you can control all aspects of case and file creation. Often, though, file creation is determined by SPSS automatically when it runs out of raw data after reading to the end of an ASCII data file. At that point, it creates the working data file. However, you can control this decision, and we illustrate this capability by modifying INPUT1.SPS slightly.

Click **Window...INPUT1 - SPSS Syntax Editor**
Add a **blank line after END LOOP**
Type the command **END FILE.** in this line
Make sure **END CASE** precedes **END LOOP**

The syntax editor should appear as Figure 4.7

Figure 4.7 Modified Program with End File Command Added



At first glance, it might appear that this END FILE command is redundant. After all, it occurs after both the END CASE and END LOOP commands. What effect might this command have? Run the program by

Highlighting all the lines

Clicking on the **Run** Button

The output from LIST says that SPSS created five cases, not ten. Inspection of the output shows that only data from the first person in the file was used to create the working data file. The program operated correctly for the first person, creating five rows with three variables, but it dropped the second person (and it would have dropped any subsequent records).

The END FILE command was encountered before END INPUT PROGRAM. This means that SPSS passed the decision of when to finish reading the data file to the user. Our program operates as follows:

1) Read the first three records;
2) Loop five times for the first person and create five cases with three variables each;
3) Don't read the next set of three records. Instead, stop reading the data file and write out a working file.

Figure 4.8 List Output Showing Data From Only One Person

```
       A        B        C

    1.00     9.00      .00
    2.00     8.00      .00
    3.00     7.00      .00
    4.00     6.00      .00
    5.00     5.00      .00


Number of cases read:  5    Number of cases listed:  5
```

Like END CASE, END FILE takes effect immediately, which is why it produces this result. It also has this effect because SPSS processes files by case. Logically, it is true that the end of file comes just after the end of the looping, but in SPSS programming the entire input program is read and processed for each case before the next case is read and defined. Thus, the END FILE command is encountered for the first case, and SPSS stops reading INPUT1.DAT.

CHECKING INPUT PROGRAMS

In Chapter 3 we strongly advised you to place LIST commands liberally throughout your programs when creating and debugging them. Nevertheless, an input program is a transformation, and so procedures cannot be placed within its structure.

This makes it imperative to work with small but representative files when writing an input program, as we did in the first example. The larger the file, the more that can go wrong and the harder to figure out what went wrong.

INCOMPLETE INPUT PROGRAMS

There are two types of mistakes to make in writing input programs. The first is a logical mistake, such as placing the END CASE command in the wrong location. The second, just as common, is to make a typing error when creating commands. In either instance, strange things can happen that are directly related to the fact that an error occurred in the middle of an input program structure.

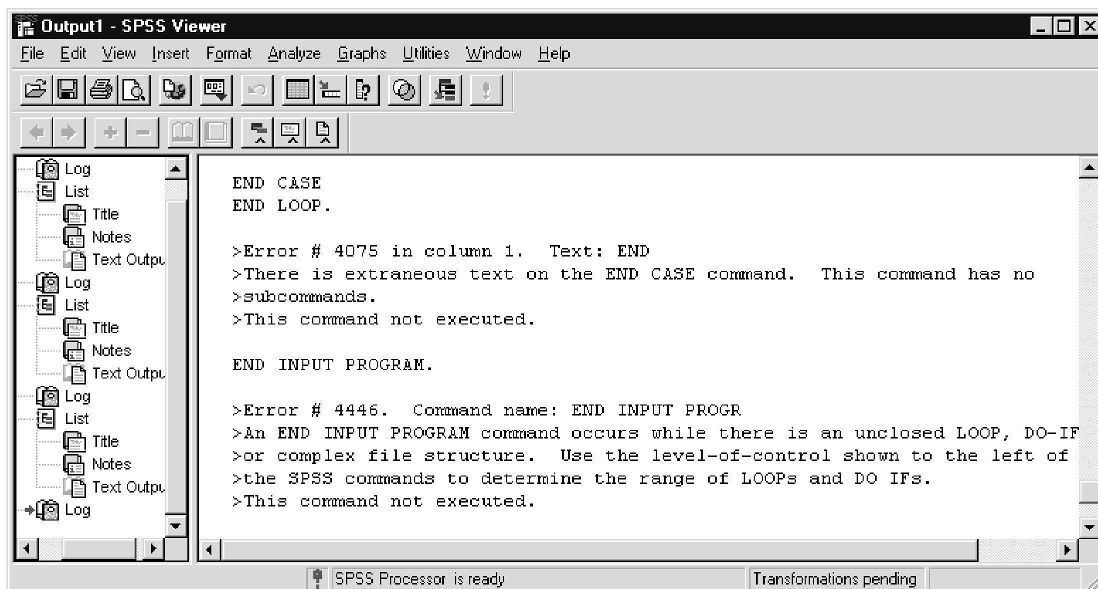
When SPSS is in the input program state, it hasn't yet created a working data file. Moreover, SPSS will be in the middle of various definitions, which *are not necessarily reset* when the program fails. In other words, when there is an error in an input program, SPSS will often remain in the input program state. This isn't good, because when you fix an error and rerun your program, SPSS will become confused when it sees an INPUT PROGRAM command when it is already in that state.

To illustrate this problem, we will make the most common type of syntax error and see what it causes.

> Click **Window..INPUT1 - SPSS Syntax Editor**
> Remove the line **END FILE.**
> **Remove the period** after the **END CASE** command
> **Highlight** all the lines
> Click on the **Run** button
> If necessary, switch to the **Viewer** window

When this program is run, SPSS understandably complains, although it doesn't actually state that the END CASE command is missing a period. Instead, it becomes confused about the text "END LOOP" which it thinks the user meant to add to the END CASE command (which has no subcommands).

Figure 4.9 Errors Created in an Input Program



Notice that the second error, on END INPUT PROGRAM, explicitly states that there is an error created while there is an unclosed complex file structure. If you take a look at the Data Editor (not shown), you will see that it is empty, although it does have the three variables A, B, and C created.

Now let's fix the error and run the program again.

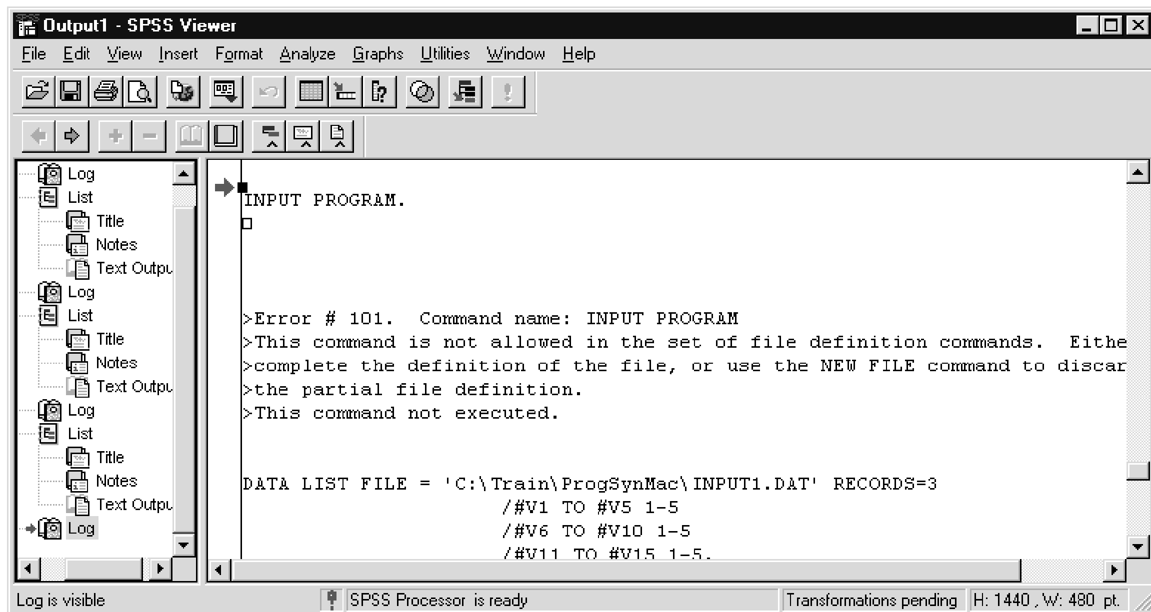
Click **Window...INPUT1 - SPSS Syntax Editor**
Add a period after the **END CASE** command
Highlight all the lines

Click on the **Run Button**

If necessary, switch to the **Viewer**

In Figure 4.10, you can see what occurred. After processing the first command, INPUT PROGRAM, SPSS complained with an error that such a command is not allowed in the set of file definition commands, i.e., those commands that normally follow an INPUT PROGRAM command.

Figure 4.10 Errors Created When Rerunning the Input Program



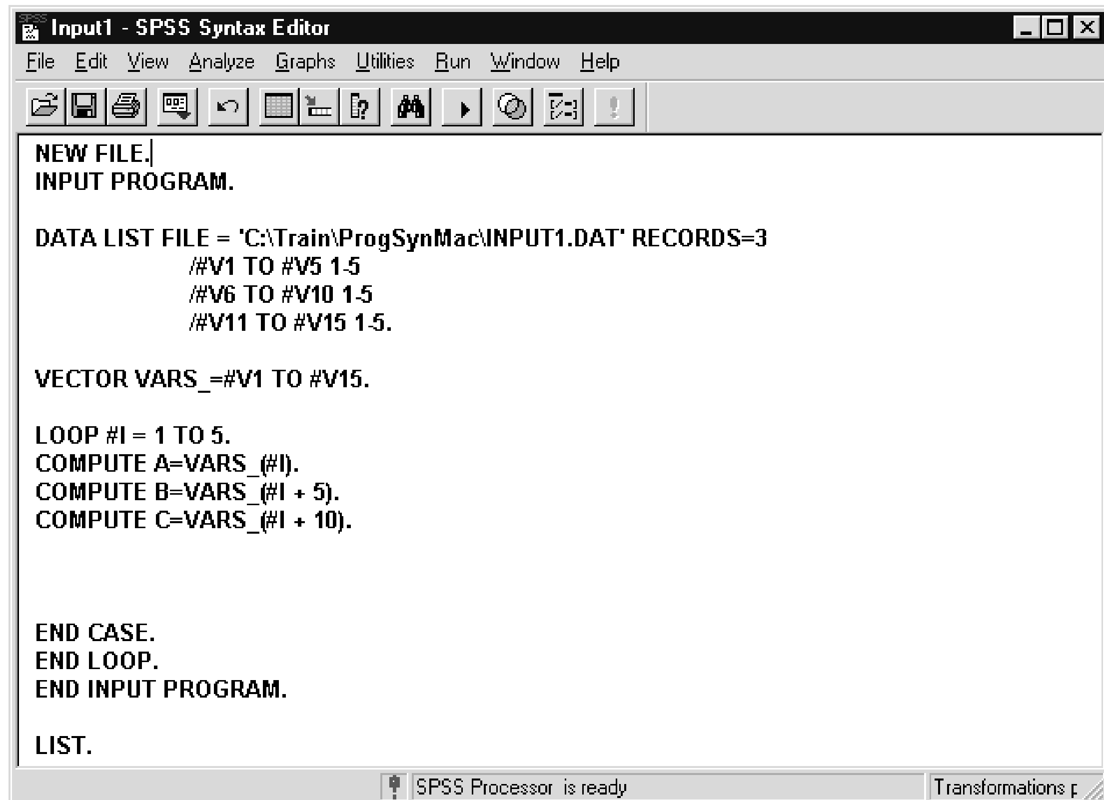
The remainder of the program is not correctly processed, so once again the Data Editor is empty of cases.

Fortunately, there is an easy solution to this problem, which SPSS mentioned in Error Message 101: Use the NEW FILE command. This command clears the working data file and removes any unclosed loops or file structures from memory. Let's add it to the program.

Click **Window...INPUT1 - SPSS Syntax Editor**
Add the command **NEW FILE.** on its own line before INPUT PROGRAM (Alternatively, click File..New..Data)

Your screen should appear similar to Figure 4.11.

Figure 4.11 New File Command Added

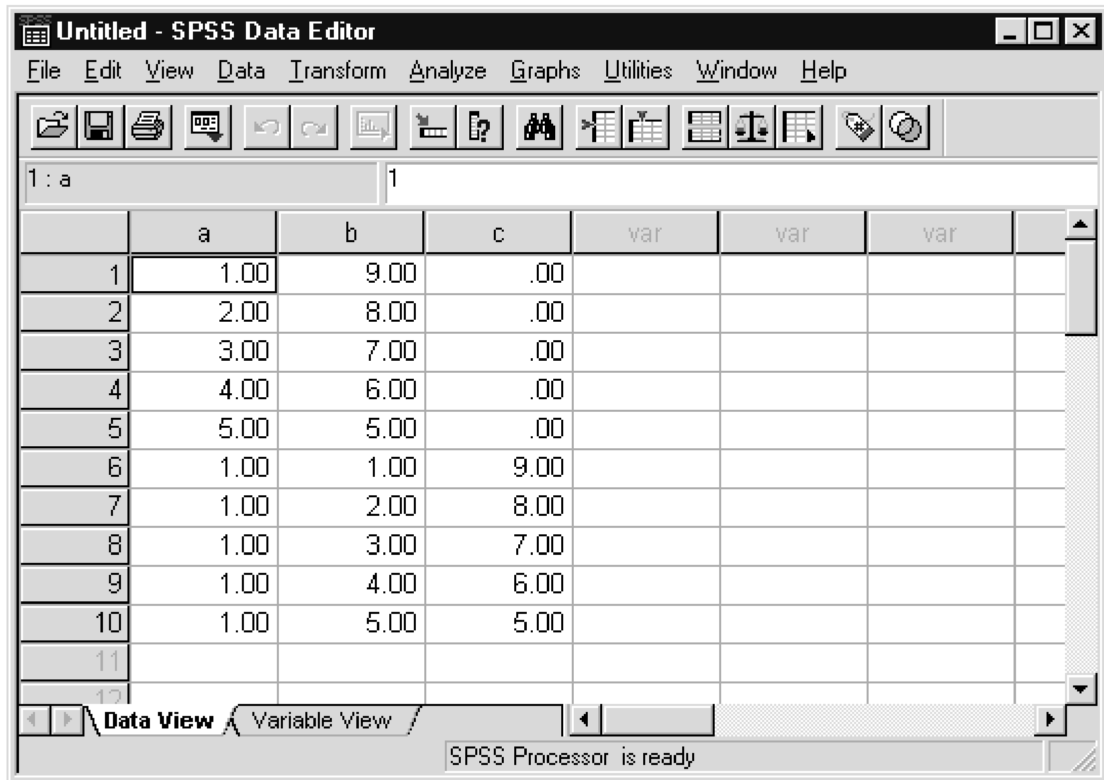


Highlight all the lines

Click on the **Run Button**

The program has now worked flawlessly with no errors. The output from LIST shows the correct file structure of 10 cases. Figure 4.12 provides a look at the file in the Data Editor. The NEW FILE command is so important that, until you become accomplished at writing input programs, and whenever you first start writing such a program, we recommend you include it as the first command and run it each time along with the other commands.

Figure 4.12 Data Editor With Five Cases Per Person



EXAMPLE 2: READING FILES WITH MISSING IDENTIFIERS

After beginning with a relatively uncomplicated example, we next attempt to read a deceptively simple file. This example will illustrate several features of input programming, including the ability to reread lines of data.

Occasionally you may encounter files which require special treatment because they have no case and/or record identifiers on every record. Figure 4.13 is an example of such a data file, recording order information for a consumer products company. Each customer's data begins with a header record that contains information on billing ID, the location of the customer, and the general product type. In this small data file, a customer can make up to three orders, represented by the records following the header record. The order records contain the price and the type of order, 1, 2, or 3.

Figure 4.13 INPUT2.DAT File

Header record

Order records

```
006 129484 30
        1              45.60
        3              44.22
        2              77.93
002 938405 34
        1              89.42
        3              32.12
003 473521 33
        1              77.52
        2              19.89
```

In this file, there are multiple records per customer, and an unequal number per customer, since not every customer has made three orders. The goal is to create an SPSS data file with one case per customer. In addition, the three order types can come out of order—for the first customer, the third order appears second—and we also want to record these correctly in sequence in the SPSS data file. As always, it is important to have a picture of the goal in mind, so Figure 4.14 displays the SPSS Data Editor with the target file structure. Each customer, identified by the billing number, represents one case, and the orders are now in the correct sequence, so that order number 3 for the first customer is placed in the variable VALUE3, even though it appeared on the second record.

Figure 4.14 Target SPSS File Structure

Untitled - SPSS Data Editor

File Edit View Data Transform Analyze Graphs Utilities Window Help

1 : region 6

	region	value1	value2	value3	billnum	prodtype
1	6.00	45.60	77.93	44.22	129484	30
2	2.00	89.42	.	32.12	938405	34
3	3.00	77.52	19.89	.	473521	33

Data View / Variable View

SPSS Processor is ready

Would it be possible to read this file with a FILE TYPE GROUPED? If not, what are we missing? What about FILE TYPE MIXED? Or FILE TYPE NESTED?

The complications in the file are that the header record has no record information, and the order records have no ID variable. We must take this into account when reading the file.

Our task will be made much simpler by using the REREAD command. To determine the type of record, it is necessary to read it. That sounds so obvious it's not worth stating, but here is the difficulty: once SPSS has read a record, it normally goes to the next record in the data file. However, that's no good here, because once we've read a record and determined whether it is a header or order record, we then want to reread it to store the necessary information in the SPSS data file. REREAD tells SPSS to do exactly that.

Let's open the program that accomplishes the task of reading this data file.

Click on **File..Open..Syntax**
If necessary, switch directories to **c:\Train\ProgSynMac**
Double-click on **INPUT2**

Figure 4.15 displays the program in the Syntax Editor window. The general method that will read this file is to:

1) Read a field from each record to test for record type.

2) If the record is a header record, build a case with the END CASE command from the information currently stored. In other words, when a header record is encountered, we know a new customer's records are being read, so we create a case for the *previous customer* from the records just above this header record. And if the record is the last record in the file, build a case and end the file with the END FILE command.

3) Read the header record with the appropriate DATA LIST.

4) If the record is not a header record, read the order type and cost as scratch variables so they are not retained in the final file.

5) Store the results into a vector created earlier in the program.

Figure 4.15 INPUT2.SPS File

Input2 - SPSS Syntax Editor

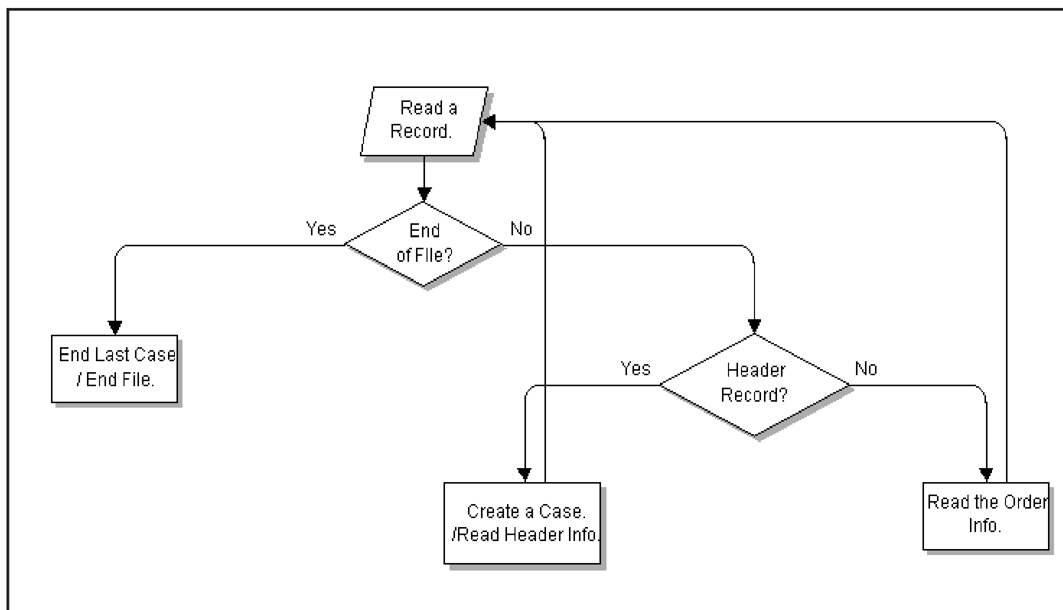
File Edit View Analyze Graphs Utilities Run Window Help

```
INPUT PROGRAM.

NUMERIC REGION VALUE1 VALUE2 VALUE3.
VECTOR ORDER=VALUE1 TO VALUE3.

DATA LIST FILE='C:\Train\ProgSynMac\INPUT2.DAT' END=#LAST /#CHECK 3.

DO IF #LAST.
.  END CASE.
.  END FILE.
END IF.

COMPUTE #TESTREC=SYSMIS(#CHECK).

DO IF #TESTREC=0.
.  DO IF NOT(SYSMIS(REGION)).
.  END CASE.
.  END IF.

REREAD.
DATA LIST  / REGION 1-3 BILLNUM 5-10 PRODTYPE 12-13.

ELSE IF #TESTREC=1.
.  REREAD.
.  DATA LIST / #TYPE 9 #COST 23-27 (2).

COMPUTE ORDER(#TYPE)=#COST.
END IF.

END INPUT PROGRAM.
LIST.
```

SPSS Processor is ready

This logic is displayed graphically in Figure 4.16. As before, we will work through the program one line at a time, explaining the logic behind each command and the structure of the syntax.

Figure 4.16 Flow Chart For INPUT2.SPS



The program begins with the INPUT PROGRAM command.

NUMERIC: This command initializes the variable REGION and the three VALUE variables that will be used to create a vector for the orders. These variables are initialized to system-missing, an important feature, because the value of REGION will be used to determine when the first header is being read for the first case.

VECTOR: This command defines the vector ORDER from the three VALUE variables.

DATA LIST: This is the first of three DATA LIST commands. It defines the file to be read, and it includes the END subcommand, which defines the variable #LAST (a scratch variable so it is not retained). #LAST is a logical variable which takes on only two values. It has a value of 0 except when the end of file is reached (after the last record), where it is set to 1. This will allow us to tell SPSS to create the last case and to end the file. We read only the variable #CHECK from each record, from column 3. That column has non-missing information only for header records. We could have chosen any column that differentiated a header record from an order record.

DO IF #LAST: This first DO IF checks for the value of #LAST. Since #LAST is a logical variable, the DO IF is executed when this condition is true, i.e., when #LAST is equal to 1. And that occurs when no records remain and the end of file is reached.

END CASE: Notice that this command is indented four spaces. It is good programming practice to make programs more readable by indenting lines that fall within a DO IF structure (or DO REPEAT or LOOP). SPSS commands can be indented without the period in column 1 when running interactively from the Syntax Editor, but programs run in the production facility or with the INCLUDE command should include the period (or, alternatively, a plus sign or dash) to indent a command.

The END CASE command, for the last record in the file, tells SPSS to create a case. We must do this check because the last record is not followed by another header record, which is the normal method used to tell SPSS when to build a case.

END FILE: The command tells SPSS to create the working data file, again because this is the last record.

COMPUTE: The COMPUTE command creates a logical variable that allows us to check the record type, using the variable #CHECK from the first DATA LIST. When #CHECK is system missing, #TESTREC is set to true, or 1. This occurs for order records. Why? When #CHECK is not system–missing, #TESTREC is set to false, or zero.

DO IF #TESTREC=0: This second DO IF is the heart of the program. It checks to see whether #TESTREC is 0, which means we are reading a header record, and so must create a case.

DO IF NOT(SYSMIS(REGION)): The third DO IF, nested in the second, creates a case when REGION is not system missing. This is true for every header record except the first, where REGION was initialized to system missing by the NUMERIC command. So for all header records except the first, this DO IF command creates a case.

REREAD: After creating a case, it's time to get on with building the next case, beginning with the header record we just read. Thus, the REREAD command tells SPSS to go back to the beginning of this record.

DATA LIST: This second DATA LIST rereads the record with a format appropriate for a header record.

ELSE IF #TESTREC=1: This block handles order records. The header test variable takes on values of 0 and 1. When it is equal to 1, that means that #CHECK is system missing, so we are reading an order record, which has no information in column 3.

REREAD: Once again, we must tell SPSS to reread the record because SPSS had to read it once to see whether it was a header or order record.

DATA LIST: The third DATA LIST command reads the two variables that are on order records. It uses scratch variables because this information will either be passed to other permanent variables or is unnecessary to retain.

COMPUTE: This command plugs the cost of each order into the correct element of the vector ORDER. For example, if #TYPE=2, then the second element of ORDER (VALUE2) is set to the value of #COST for that record.

END IF: This closes the last DO IF.

The program ends, as always, with an END INPUT PROGRAM command, then a LIST command to force program execution and check our work.

To see this in operation:

Highlight all the lines in the program

Click on the **Run** button

When an input program works correctly, it all seems so deceptively simple. In Figure 4.17 we see the output from LIST. We have created

three cases as planned, retaining region, billing number, and product type, and we have placed the costs in the correct VALUE variables. The second customer has a missing value for VALUE2 because he made no order of this type.

Figure 4.17 List Output With Three Cases

```
  REGION   VALUE1   VALUE2   VALUE3 BILLNUM PRODTYPE

    6.00    45.60    77.93    44.22  129484    30
    2.00    89.42      .      32.12  938405    34
    3.00    77.52    19.89      .    473521    33


Number of cases read:  3    Number of cases listed:  3
```

WHEN THINGS GO WRONG

Creating a program such as INPUT2.SPS is not easy, though, without lots of experience at programming with SPSS or with other programming languages. Expect to make several errors when trying to create this program, which, as noted previously, can be difficult to check because procedures can't be placed in the middle of the program to check your work. One important thing to do is to make sure you are working in an empty Viewer window, and to empty it out on a regular basis. Otherwise, as you execute the program, syntax errors from one run of the program will blend with errors from subsequent runs, and it will be more difficult to recognize what needs to be fixed.

One error that is sometimes made is to ignore the need to tell SPSS to stop reading the data file. What makes the decision to use an END FILE command tricky is that, in general, there is no requirement within an input program to tell SPSS when to stop reading the file. So in our first example, no END FILE command was necessary. This is because normally SPSS reads the last line of data, inputs it as instructed, sees an end–of–file marker in the raw data file, and writes out a working data file automatically.

In this program, however, we took control of the process with the END subcommand on DATA LIST. We did this so we would be able to create the last case, but SPSS then passed end of file processing to the user.

SUMMARY

This chapter provided a brief look at input programs via the use of a simple example. We discussed the basics of writing such programs, and the importance of controlling end of case and end of file processing. We also discussed how to check and correct input programs. We then attempted a complex input program that read a data file with missing record and identification fields.

Chapter 5 Advanced Data Manipulation

Topics

Introduction
Splitting a String Variable on a Delimiter
Reading Multiple Cases on the Same Record
An Existing SPSS Data File with Repeating Data
Print Command for Diagnostics
Practical Example: Consolidating Transactions
Appendix: Identify Missing Values by Case

INTRODUCTION

We have discussed how to read various non-rectangular files in SPSS, using either input programs or complex file types. In this chapter, we consider some examples of how to further manipulate data in SPSS, and we discuss a few additional commands often used in data transformation. The examples were chosen to both illustrate potentially useful applications and to show the operation of these commands.

SPLITTING A STRING VARIABLE ON A DELIMITER

Often users encounter ASCII data files that are comma delimited, rather than in fixed or freefield format. Comma delimited means that the values of variables are separated by commas. Many database software programs create such files. An example of this type of file is displayed in Figure 5.1.

Figure 5.1 Comma–Delimited Data

```
Missing Data

1,alpha,23453,,43553,,,sadfs,5435
3452,dsff,3,1,3545,3,2,ssss,34
,,,3,543,,4,dadf,56,78
```

SPSS does have an option to automatically read this type of file, under freefield input, but this example serves to illustrate methods used when working with string fields. However, there should be ten values per record, but the first and second records have only 9 (because there are only 8 commas). Standard freefield input will misread such a data file. (Note the LIST option on the DATA LIST command and the default Text Wizard dialog choices (click File..Read Text Data) will read the data correctly.)

Also, on occasion a list of string values, separated by a delimiter character, are stored as a single field in a data file. For example, radio stations listened to, problems encountered with a product, products used in the last month, etc.

Such files can be read without too much fuss. The method is based on these steps:

1) Read all of each record as a long string variable;
2) Search through this string for a comma, extracting the text before this position as a variable;
3) Replace the string with the contents following the comma, and continue the process;
4) Repeat until there are no more commas, which allows for variable-length input fields.

To accomplish these tasks we need to utilize not only the VECTOR and LOOP commands, but also the string transformation capability of SPSS. The program that illustrates this procedure is named COMMA.SPS.

Click on **File...Open..Syntax**
If necessary, switch directories to **c:\Train\ProgSynMac**
Double-click on **COMMA**

Figure 5.2 Comma.sps Syntax File

Comma - SPSS Syntax Editor

File Edit View Analyze Graphs Utilities Run Window Help

```
DATA LIST / VAR 1-80 (A).
BEGIN DATA
1,alpha,23453,,43553,,,sadfs,5435
3452,dsff,3,1,3545,3,2,ssss,34
,,,3,543,,4,dadf,56,78
END DATA .

VECTOR SLOTS (10,A5).
COMPUTE #I=0.
LOOP.
.  COMPUTE #comma=INDEX(VAR,',').
.  COMPUTE #I=#I+1.
DO IF #comma > 0 .
.  COMPUTE SLOTS(#I)=(SUBSTR(VAR,1,#comma-1)).
.  COMPUTE VAR=SUBSTR(VAR,#comma+1).
END IF.
END LOOP IF #comma=0.
COMPUTE SLOTS(#I)=VAR.

LIST slots1 to slots10.

DO REPEAT NS=NS1, NS3, NS4, NS5 NS6 NS7 NS9, NS10
      / SL=SLOTS1 SLOTS3 TO SLOTS7, SLOTS9, SLOTS10.
COMPUTE NS=NUMBER(SL,F5).
END REPEAT.
LIST.
```

SPSS Processor is ready

We are going to read the small dataset from Figure 5.1, which is included as inline data in the program. This method will easily generalize to any size file, and can be used to split an existing string variable into new variables. We now discuss program execution, step by step.

DATA LIST: Each record of data is read in as one long string variable, 80 characters in width, with the name VAR. Doing so makes it possible to easily search through all the data and avoids the problem of missing data between commas. This trick—reading all the data as a long string—is a very common technique to read troublesome files.

VECTOR: The command creates a vector to store the actual variables. 10 elements of SLOTS are created because there are ten variables for each case. Each is given a format of A5 (alpha field 5 columns wide) because a number of the fields contain string data.

COMPUTE: For each case, #I is initialized to 0, but note that this is outside the loop that will be created next.

LOOP: This Loop command has no index value, so it might appear as if it will not loop at all, or loop forever, but that is not true. Looping will continue until the maximum number of loops is reached (40), or until a condition is met on the END LOOP command.

The two COMPUTE commands find the next comma and increment #I. The INDEX function in SPSS as used in the first COMPUTE has this format:

INDEX(variable,test string)

It returns an integer that indicates the starting position of the first occurrence of the test string in the variable. It returns 0 if the test string is not found within the variable.

COMPUTE #COMMA: The INDEX function searches through VAR for a comma. The syntax places the test string value in apostrophes (','). It sets #COMMA equal to the position of the first comma it finds. Thus for the first case and the first iteration through the loop, the value of #COMMA is 2.

COMPUTE #I: The value of #I is incremented by 1. The reason will become apparent in a moment.

We next encounter a DO IF–END IF structure. The two COMPUTE commands within this structure are executed when #COMMA is greater than 0. When does this occur? Answer: when the value that the INDEX function returns is greater than 0, which occurs when a comma can be found in VAR. In other words, the COMPUTE commands are executed when there is a value to record from VAR.

The first COMPUTE uses the SUBSTR function. SUBSTR has this format:

SUBSTR(variable,position,length)

It returns the substring of the variable beginning at the position indicated and running for the length specified.

COMPUTE: SLOTS(#I): Let's see how this operates for the first iteration of the loop. #COMMA is equal to 2, so this says to take a substring of VAR, beginning in column 1, for one column. This means to read the first column of information (a "1") and place it in the first element of SLOTS, because #I =1 (now you can see why we increment #I once each loop). So now SLOTS1=1 and the first variable for the first case has been created.

COMPUTE VAR: This command replaces the full 80-character VAR with a shorter version, taking a substring beginning at column 3, thus leaving out the "1" and the comma that follows it for the next iteration.

END LOOP: The looping only ends when #COMMA is equal to zero. This occurs when the INDEX function can't find any more commas, which means we don't need to read any more of VAR.

If you've followed the program to this point, it might appear that our data file reading problem has been solved, but that's not quite the case. A line of data does not end with a comma, but this means that the last field will not be stored by SPSS because the program as written requires a comma at the end of a field (so that the preceding characters can be stored in the vector SLOTS). The last field must be handled separately.

COMPUTE SLOTS(#I): This command takes the current value of VAR and puts it in the element for SLOTS equal to the current value of #I. Thus if the last bit of data is the tenth variable, as for the third case, this command places 78 into SLOTS10.

The LIST command then executes the preceding transformation statements and checks our work. Before executing the program, let's run through one more example. After the program places the value of "1" in SLOTS1, it does the following:

1) Computes VAR to be equal to "**alpha,23453,,43553,,,sadfs,5435**"
2) Loops again, since #COMMA is equal to 2
3) Computes #COMMA=6
4) Computes #I=2
5) Computes SLOTS2= substring of VAR from position 1 to 5 = "alpha"
6) Recomputes VAR, etc.

To see this in action:

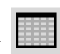
Highlight all the lines from **DATA LIST** to the **first LIST** command

Click the **Run** tool button

Figure 5. 3 Ten New Variables Created

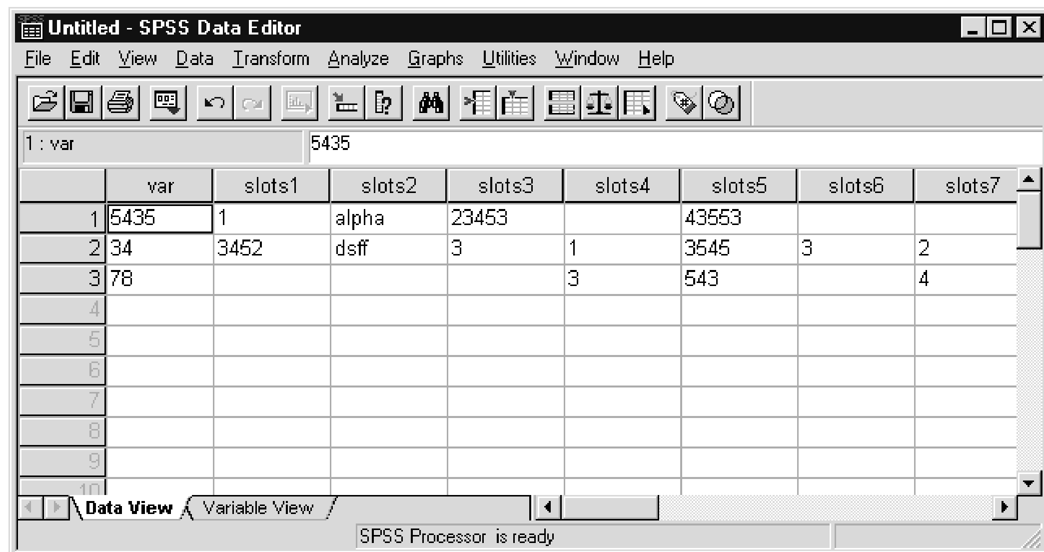
```
SLOTS1 SLOTS2 SLOTS3 SLOTS4 SLOTS5 SLOTS6 SLOTS7 SLOTS8 SLOTS9 SLOTS10

1      alpha  23453         43553                sadfs  5435
3452   dsff   3      1      3545   3      2      ssss   34
                     3      543           4      dadf   56     78


Number of cases read:  3    Number of cases listed:  3
```

The output from LIST shows that the values from the raw data have been correctly placed in the appropriate variables, with missing data (a blank for string variables) inserted in the right spots. The tenth variable value is missing for both the first and second cases. To further illustrate how the program is functioning, switch to the Data Editor window.

Click on the **Goto Data** tool 

The variable VAR is retained in the file and has the value of 5435 for the first case, 34 for the second case, and 78 for the third case. It stores the last valid value in each record because that was the last field that did not end with a comma.

Figure 5.4 Data Editor With New Variables and VAR



	var	slots1	slots2	slots3	slots4	slots5	slots6	slots7
1	5435	1	alpha	23453		43553		
2	34	3452	dsff	3	1	3545	3	2
3	78				3	543		4

Note that the VAR column is narrowed in the Data Editor so both it and the SLOT variables are visible.

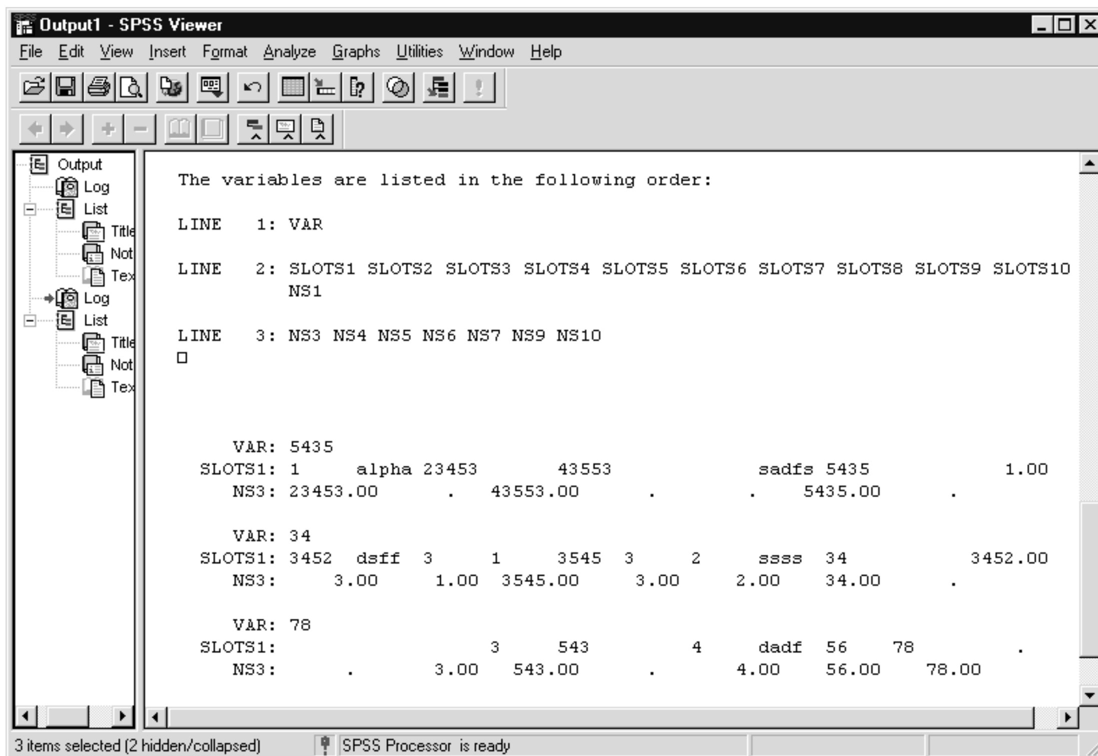
This file is now a regular SPSS data file and ready to be used for analysis, but all the variables are strings. To make the variables that are numbers into numeric variables, we use the NUMBER function. The DO REPEAT command simply affords an easy means to create a series of eight COMPUTE statements. The NUMBER function requires two arguments, the variable to be converted, and a format by which to read the variable (here F5 for 5 characters, no decimal digits).

Click **Window..COMMA - SPSS Syntax Editor**
Highlight the commands from **DO REPEAT** to **LIST**

Click on the **Run** tool button

After these commands are executed, the Viewer window displays all eight COMPUTE commands created from the DO REPEAT (not shown). Figure 5.5 displays a portion of the output from LIST with the numeric versions of eight of the SLOTS variables. Why do these variables all have two decimal digits displayed?

Figure 5.5 List Output with Numeric Variables



This program can be readily generalized by increasing the width of the long string variable VAR on the DATA LIST, by using more than one of these variables if the record length is more than 256 characters (the maximum width for a long string variable), and by setting the number of elements for SLOTS as high as necessary to create as many fields as exist in the data. MXLOOPS may have to be set higher than the default of 40 as well (use SET MXLOOPS= value.).

READING MULTIPLE CASES ON THE SAME RECORD

In Chapter 4 we discussed how to rearrange a file and change the columns into rows. At times you may be faced with the reverse situation: many cases stored in one record. This can occur whenever all information for one person or organization is stored in one record of data during data entry for efficiency's sake. You may later wish to place each occurrence within an ID into a separate case for analysis purposes (such as to calculate aggregate statistics across cases). An example of such a file is in Figure 5.6.

Figure 5.6 A File With Multiple Cases Per Record

```
1 1 117 837 947 594
2 1 352 039 373
3 2 152 838 939
4 2 422 234 098
5 3 198 578 274 302 987
6 3 723 512 609
```

Each case has identifying information in the first two fields. It then has up to five fields of valid data, which could be customer orders, procedures in a hospital, exam scores, and so forth. This type of structure can be thought of as *repeating data*, because each case contains repeating bits of information that are of the same type.

SPSS supplies a predefined command that will read these types of data. The REPEATING DATA command must be used within an input program structure, and it will build one new case for each repeating group. All the repeating groups must contain the same type of information, but there can be a different number of repeating groups for each record.

The user must tell SPSS how many repeating groups of data are in each record, but often files of this type do not include such information, as in Figure 5.6. This value can be created from the data, though, as we demonstrate below. The sample program that reads these types of data is named Repeat1.sps.

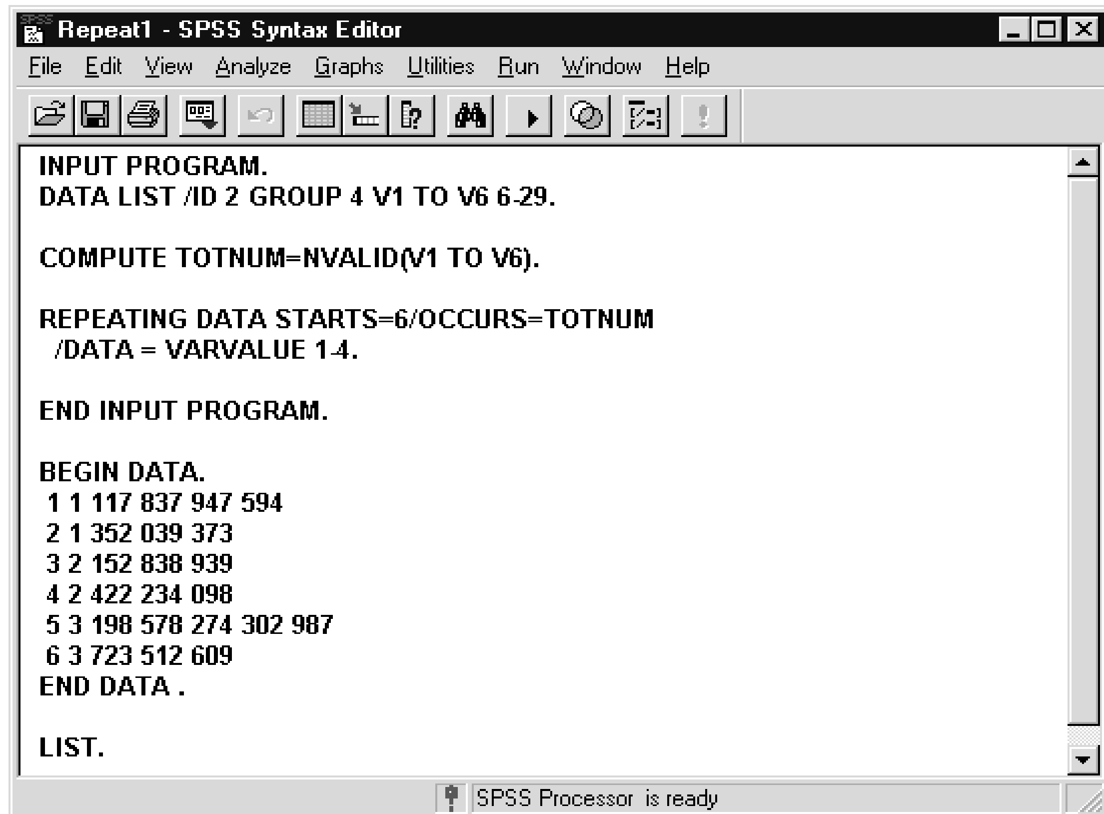
Click on **File..Open..Syntax** (move to c:\Train\ProgSynMac)
Double-click on **Repeat1**

The method of reading the file is not complicated.

1) Read the data with a standard DATA LIST command;
2) Compute a variable that records the number of repeating data groups;
3) Use the REPEATING DATA command to create the cases.

You must supply the number of repeating data groups for the REPEATING DATA command, but there is no penalty in initially estimating too many of these on the DATA LIST. We illustrate this by having SPSS read up to six variables for the critical information even though no more than five appear on any input record.

Figure 5.7 REPEAT1.SPS Syntax File



```
INPUT PROGRAM.
DATA LIST /ID 2 GROUP 4 V1 TO V6 6-29.

COMPUTE TOTNUM=NVALID(V1 TO V6).

REPEATING DATA STARTS=6/OCCURS=TOTNUM
  /DATA = VARVALUE 1-4.

END INPUT PROGRAM.

BEGIN DATA.
 1 1 117 837 947 594
 2 1 352 039 373
 3 2 152 838 939
 4 2 422 234 098
 5 3 198 578 274 302 987
 6 3 723 512 609
END DATA .

LIST.
```

Specifically, the program does these tasks:

DATA LIST: Read in the variables ID and GROUP, which appear once per record, and then read six variables for the repeating data.

COMPUTE: This command uses the NVALID function, which returns the number of nonmissing values in a set of variables for each case. This puts the number of occurrences of repeating data into the variable TOTNUM.

REPEATING DATA: The command has only three required subcommands. The STARTS subcommand specifies the beginning column of the repeating data segments, here column 6. The OCCURS subcommand tells SPSS how many repeating data groups are on each record, and the DATA subcommand tells SPSS the names, locations, and formats (if necessary) for each variable within the repeating groups. There can be many variables within each repeating group, but we've kept things simple in this example and have only one.

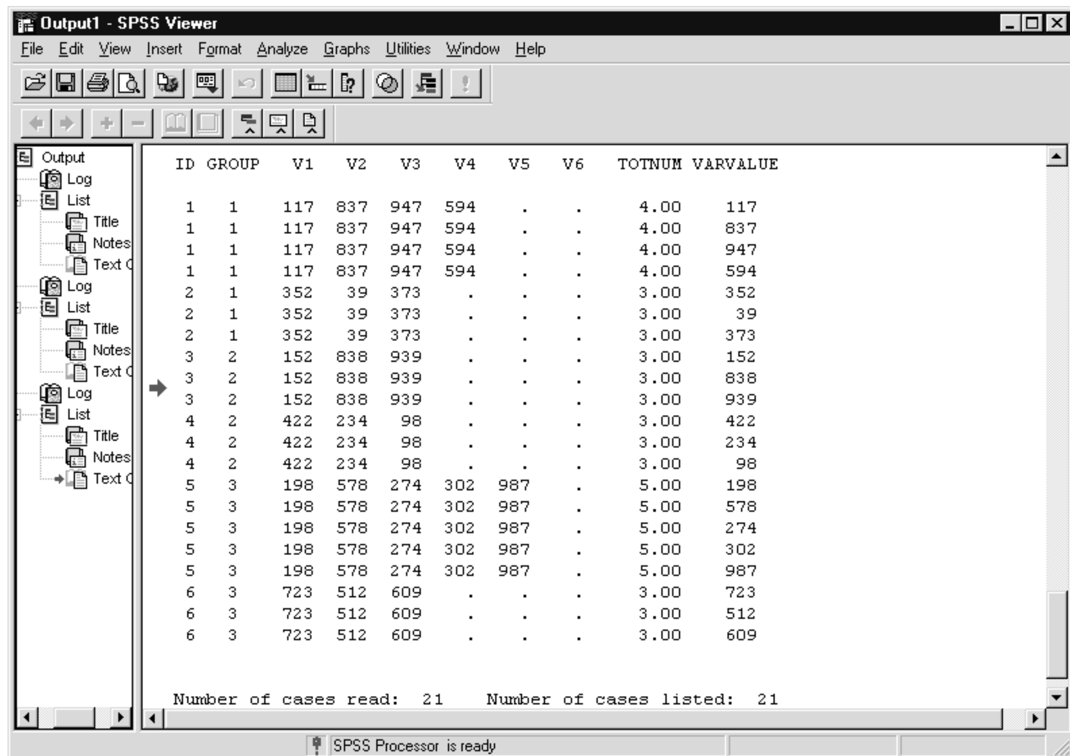
END INPUT PROGRAM closes the input program structure, and it is followed by the inline data.

Run the program by:

Highlighting all the lines

Clicking on the **Run** Button

Figure 5.8 List Output With All Variables



ID	GROUP	V1	V2	V3	V4	V5	V6	TOTNUM	VARVALUE
1	1	117	837	947	594	.	.	4.00	117
1	1	117	837	947	594	.	.	4.00	837
1	1	117	837	947	594	.	.	4.00	947
1	1	117	837	947	594	.	.	4.00	594
2	1	352	39	373	.	.	.	3.00	352
2	1	352	39	373	.	.	.	3.00	39
2	1	352	39	373	.	.	.	3.00	373
3	2	152	838	939	.	.	.	3.00	152
3	2	152	838	939	.	.	.	3.00	838
3	2	152	838	939	.	.	.	3.00	939
4	2	422	234	98	.	.	.	3.00	422
4	2	422	234	98	.	.	.	3.00	234
4	2	422	234	98	.	.	.	3.00	98
5	3	198	578	274	302	987	.	5.00	198
5	3	198	578	274	302	987	.	5.00	578
5	3	198	578	274	302	987	.	5.00	274
5	3	198	578	274	302	987	.	5.00	302
5	3	198	578	274	302	987	.	5.00	987
6	3	723	512	609	.	.	.	3.00	723
6	3	723	512	609	.	.	.	3.00	512
6	3	723	512	609	.	.	.	3.00	609

Number of cases read: 21 Number of cases listed: 21

Twenty one cases have been created by SPSS, for the 21 separate repeating data values in the input file. There are four repeating groups for the first record (117, 837, 947, and 594), so SPSS created four cases from the first input record. Although no input record had six repeating data groups, our specification of six input variables on the DATA LIST command didn't cause any difficulty. SPSS simply created a sixth variable, V6, that is missing for every case. The critical information, in addition to ID and GROUP, is the information stored in VARVALUE, which came from the repeating data.

At this point, we could delete the variables V1 to V6 and TOTNUM from the file (though the latter might be useful for analysis), and save the file as an SPSS data file. We then can compute such quantities as the mean of VARVALUE for the whole file, which would have been very difficult to calculate with the initial file structure.

A file in this format also allows us to study the data by customer by aggregating based on ID.

AN EXISTING SPSS DATA FILE WITH REPEATING DATA

What if an SPSS data file has already been created with the same format as the ASCII data file in Figure 5.6? That is, what if you are faced with an SPSS data file that has only six cases, with repeating data on each case? Such a file is REPEAT.SAV.

> Click on **File..Open..Data** (move to c:\Train\ProgSynMac)
> Double–click on **REPEAT**
> Click **No** when asked to save current data

What is certainly true is that you cannot use the REPEATING DATA command to restructure the file, because that command, and input programs in general, operate on raw ASCII data, not SPSS data files.

Figure 5.9 SPSS Data FIle, Repeat.sav, With Repeating Data

Repeat - SPSS Data Editor

File Edit View Data Transform Analyze Graphs Utilities Window Help

1 : id 1

	id	group	v1	v2	v3	v4	v5	v6
1	1	1	117	837	947	594	.	
2	2	1	352	39	373	.	.	
3	3	2	152	838	939	.	.	
4	4	2	422	234	98	.	.	
5	5	3	198	578	274	302	987	
6	6	3	723	512	609	.	.	
7								
8								
9								
10								

Data View / Variable View

SPSS Processor is ready

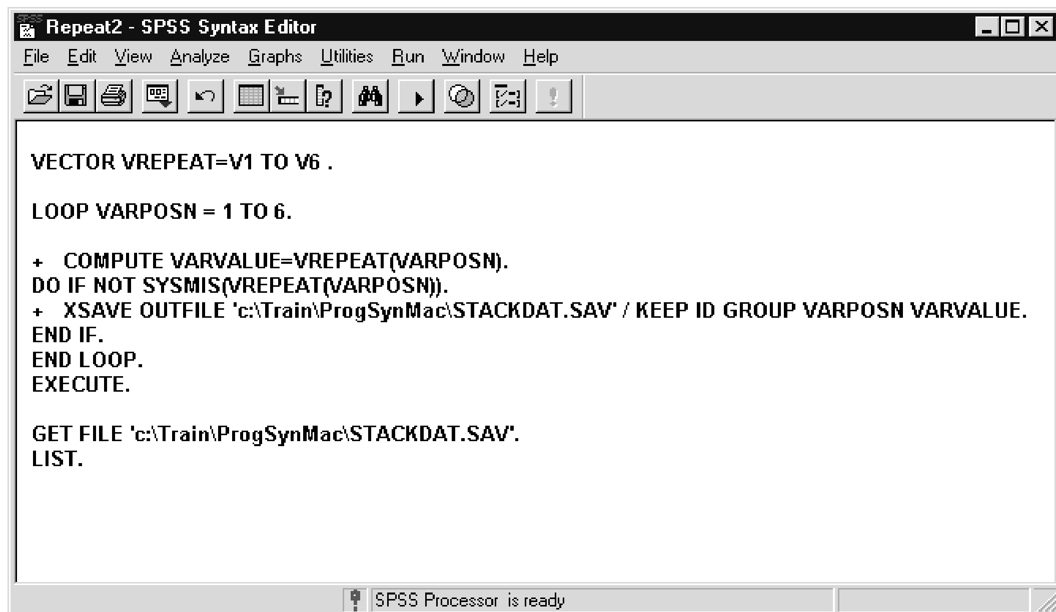
One option is to write the data out as an ASCII file and read it back in as above, but this is cumbersome and time–consuming with large files. A better alternative is to use SPSS programming to take this existing file structure and modify it.

To do so we will need the XSAVE command. XSAVE produces an SPSS data file, just as the SAVE command does. The key difference is that XSAVE is not a procedure but a transformation; consequently, XSAVE can be placed within DO IF and LOOP structures. This often improves processing time by reducing the number of data passes, and sometimes allows you to do things that are basically impossible with only the SAVE command.

The program that restructures the file is named Repeat2.sps.

> Click on **File..Open..Syntax** (move to c:\Train\ProgSynMac)
> Double-click on **Repeat2**

Figure 5.10 Repeat2.sps Syntax File


```
VECTOR VREPEAT=V1 TO V6 .

LOOP VARPOSN = 1 TO 6.

+  COMPUTE VARVALUE=VREPEAT(VARPOSN).
DO IF NOT SYSMIS(VREPEAT(VARPOSN)).
+  XSAVE OUTFILE 'c:\Train\ProgSynMac\STACKDAT.SAV' / KEEP ID GROUP VARPOSN VARVALUE.
END IF.
END LOOP.
EXECUTE.

GET FILE 'c:\Train\ProgSynMac\STACKDAT.SAV'.
LIST.
```

Not surprisingly, the program relies upon a vector and looping to accomplish its job. It works in this fashion:

1) Create a vector to store the repeating data
2) Loop as many times as there are repeating data groups
3) Create the variable to store the repeating data group, and if it is not missing, write a case to the file STACKDAT.SAV, retaining only the necessary variables.

Specifically, the commands do the following:

VECTOR: The new vector VREPEAT is created from the existing repeating data variables, here V1 to V6.

LOOP: We loop as many times as there are repeating data groups. However, requesting more loops than there are groups will not cause an error or the program to function improperly.

COMPUTE VARVALUE: On the first loop, the value (117) of the first element of VREPEAT (V1) for the first record is placed into VARVALUE.

DO IF: This is executed if the current element of VREPEAT is *not missing*. In other words, the statements in the DO IF structure are executed if there is a valid repeating data value. This check means that no cases will be created when there is missing data, even when a record has fewer than 6 repeating data values.

XSAVE: The XSAVE command has the same format as SAVE. We name the output file and tell SPSS which variables to retain. Users often picture a SAVE command as being applied to the whole file at once, but if that were true, XSAVE couldn't write out the data appropriately here. Instead, both SAVE and XSAVE write out one case at a time, until all the data from the current working SPSS file have been read. This means that XSAVE writes out the first case to STACKDAT.SAV, whose values are these:

ID	GROUP	VARPOSN	VARVALUE
1	1	1.00	117.00

and waits for input from the next iteration of the loop.

In this manner, SPSS creates 21 cases from the original 6, one for each valid repeating data group. The idea of looping through a record and writing out the cases to an SPSS data file with XSAVE is the central idea of this, and similar, SPSS programs.

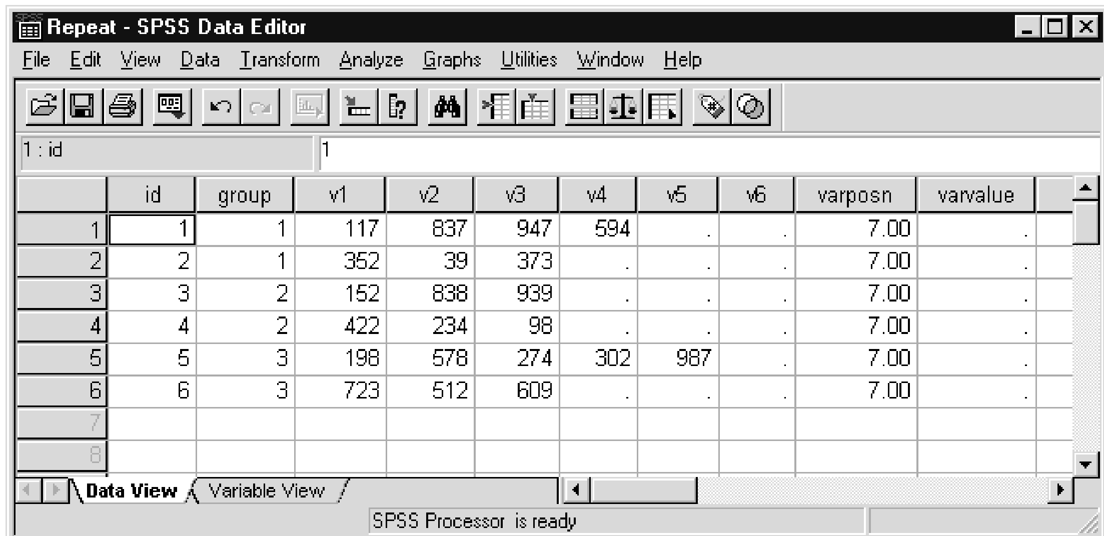
The program closes with END IF and END LOOP. We then need to get the file STACKDAT.SAV to see the new structure.

Highlight all the lines from **VECTOR** to **EXECUTE** (do not run the GET FILE command yet)

Click on the **Run** tool button

Switch to the **Data Editor** (click the Goto Data tool)

Figure 5.11 Data Editor Showing REPEAT.SAV With New Variables



Repeat - SPSS Data Editor

File Edit View Data Transform Analyze Graphs Utilities Window Help

1 : id 1

	id	group	v1	v2	v3	v4	v5	v6	varposn	varvalue
1	1	1	117	837	947	594	.	.	7.00	.
2	2	1	352	39	373	.	.	.	7.00	.
3	3	2	152	838	939	.	.	.	7.00	.
4	4	2	422	234	98	.	.	.	7.00	.
5	5	3	198	578	274	302	987	.	7.00	.
6	6	3	723	512	609	.	.	.	7.00	.
7										
8										

Data View / Variable View

SPSS Processor is ready

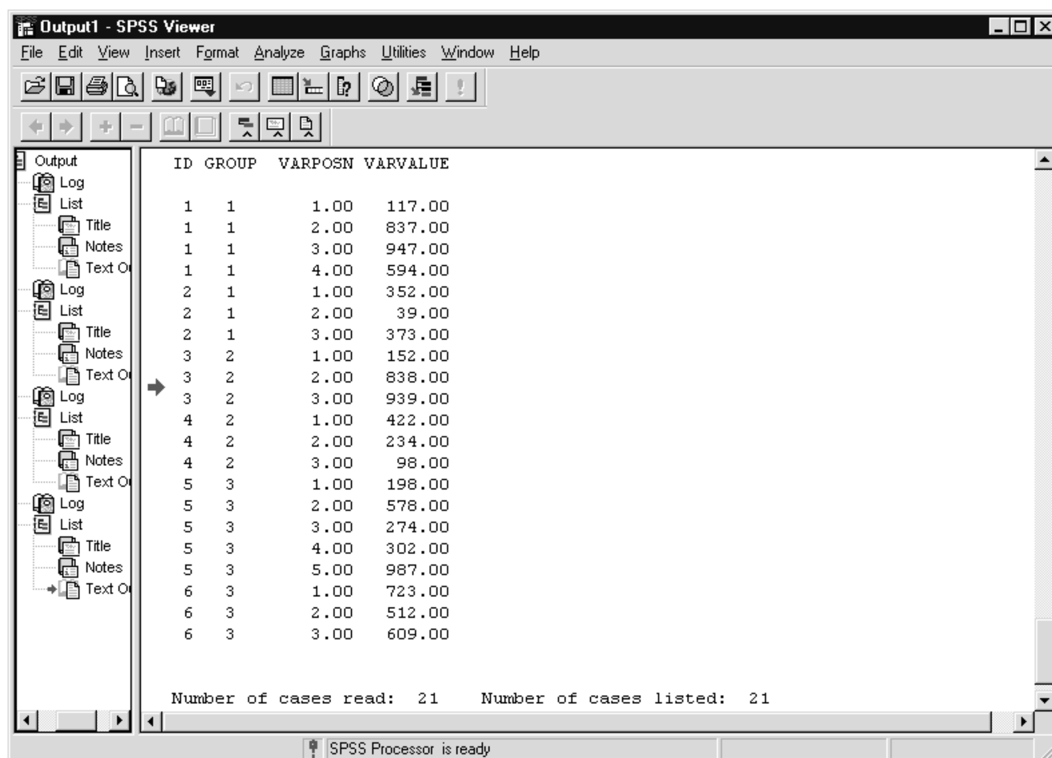
We are first looking at the existing file, not STACKDAT.DAV, so you can understand better how this program functions. Two new variables are created, VARPOSN and VARVALUE. The latter is system–missing for every case because no input record had a sixth valid field of repeating data. What may be surprising is that VARPOSN is equal to 7, not 6, even though the LOOP command is designed to iterate six times. What SPSS does is execute the loop six times, then increment the value of VARPOSN to 7, and test for the loop ending condition (which is VARPOSN=6). Since 7 is greater than 6, SPSS doesn't execute another loop, but it has set the value of the index variable to one higher than the end of the loop sequence. This is how loops always operate.

Now let's look at the new file structure.

> Click **Window..Repeat2 - SPSS Syntax Editor**
> **Highlight** the two lines **GET FILE** and **LIST**
> Click on the **Run** button

The output from LIST shows that we have created the desired file structure. There are 21 cases in total, and we have only the variables we need, unlike the program with the REPEATING DATA command, in which variables were dropped to create the final file.

Figure 5.12 List Output from STACKDAT.DAV



ID	GROUP	VARPOSN	VARVALUE
1	1	1.00	117.00
1	1	2.00	837.00
1	1	3.00	947.00
1	1	4.00	594.00
2	1	1.00	352.00
2	1	2.00	39.00
2	1	3.00	373.00
3	2	1.00	152.00
3	2	2.00	838.00
3	2	3.00	939.00
4	2	1.00	422.00
4	2	2.00	234.00
4	2	3.00	98.00
5	3	1.00	198.00
5	3	2.00	578.00
5	3	3.00	274.00
5	3	4.00	302.00
5	3	5.00	987.00
6	3	1.00	723.00
6	3	2.00	512.00
6	3	3.00	609.00

Number of cases read: 21   Number of cases listed: 21

These examples have illustrated how there is often more than one way to accomplish the same result in SPSS. If a file is small, it may make little difference which method is used, but for larger files, efficiency is often important (see Chapter 8). This example also demonstrates how an

existing file can be restructured, just as we did in Chapter 4 when we changed columns into cases.

PRINT COMMAND FOR DIAGNOSTICS

In the examples run thus far we frequently made use of the List command to display values for specified variables in the active data file. One disadvantage of the List command is that it is a procedure. As a result, List cannot be placed within a loop or Do If structure, and would be of limited value in determining what is happening within such structures. Also, as a procedure the List command forces a data pass. Thus if we insert three List commands in a program, three data passes must be performed, which is inefficient.

To answer these issues, SPSS has a transformation command called Print. It provides the same general features as List, but is a transformation. Print can be placed within Do If and loop structures, and can provide detailed feedback about what occurs. However, be aware that if a Print command is placed within a loop, then it will print a line of output during each iteration of the loop for every case in the file. Such output is extensive for large data files, so care must be taken. Below we briefly demonstrate the use of the Print command by inserting one into the program run earlier that read a comma-delimited file.

> Click **Window..Comma - SPSS Syntax Editor**
> Insert a **blank line** before the **COMPUTE VAR**=… line
> Type **PRINT / '#Comma= ' #comma (F2) ' Var= ' var (a50).** (all on one line)

Figure 5.13 Comma.sps Program with Print Command Added

Comma - SPSS Syntax Editor

File Edit View Analyze Graphs Utilities Run Window Help

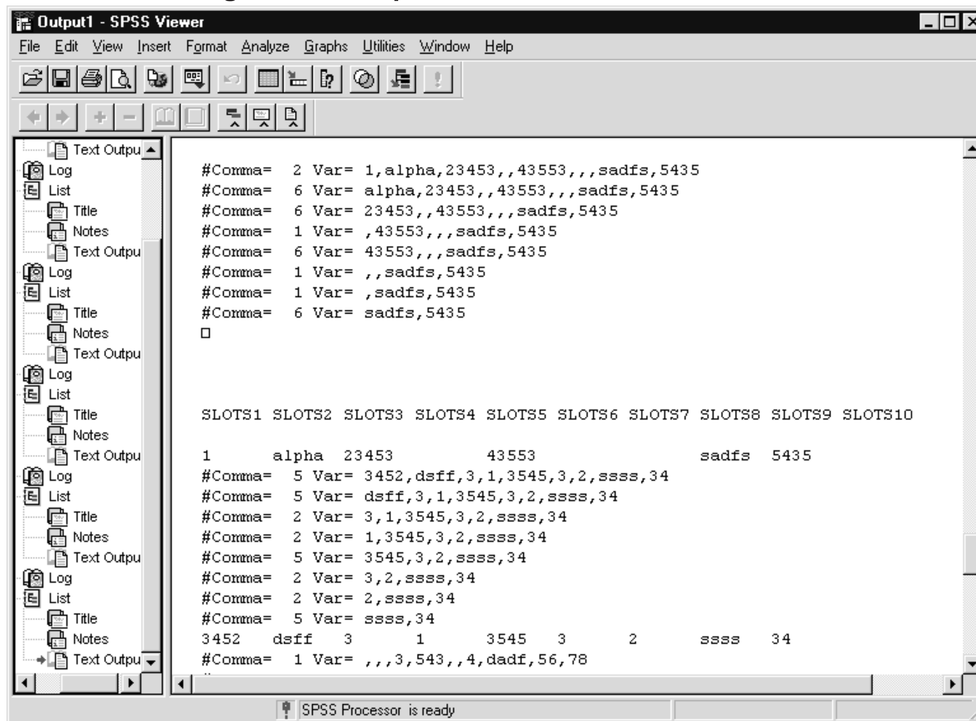
```
DATA LIST / VAR 1-80 (A).
BEGIN DATA
1,alpha,23453,,43553,,,sadfs,5435
3452,dsff,3,1,3545,3,2,ssss,34
,,,3,543,,4,dadf,56,78
END DATA .

VECTOR SLOTS (10,A5).
COMPUTE #I=0.
LOOP.
.  COMPUTE #comma=INDEX(VAR,',').
.  COMPUTE #I=#I+1.
DO IF #comma > 0 .
.  COMPUTE SLOTS(#I)=(SUBSTR(VAR,1,#comma-1)).

PRINT / '#Comma= ' #comma (F2) ' Var= ' var (a50).

.  COMPUTE VAR=SUBSTR(VAR,#comma+1).
END IF.
END LOOP IF #comma=0.
COMPUTE SLOTS(#I)=VAR.
```

SPSS Processor is ready

By default the Print command will print to the Viewer window, but it can be directed to a file. Text can be added to a Print command by enclosing it in quote marks. We use this feature to identify the variables whose values will be printed. Values for two variables (#comma and VAR) will appear. Note that unlike the List command, Print can print values for scratch variables. Formats can be given for the variables printed. Here we print only the first 50 characters of the string variable VAR in order to prevent the line from wrapping.

Click **Run..All**
Scroll up in the Viewer window to the lines beginning with **#comma=**

Figure 5.14 Output from Print Command



The output shows the values for #comma (column position for the next comma) and VAR (the string containing the data). In each iteration of the loop we see how the first element (preceding the comma) of VAR is stripped off, leaving the tail end (beyond the comma) to be processed in the next step. Now we have access to just what is occurring within the loop. Recall that the List command, although useful, can display only the final result. Thus the Print command provides a glimpse into the workings of loops, Do Ifs, and other transformations and input program structures. Print is extremely useful when debugging programs containing such structures or complex transformations.

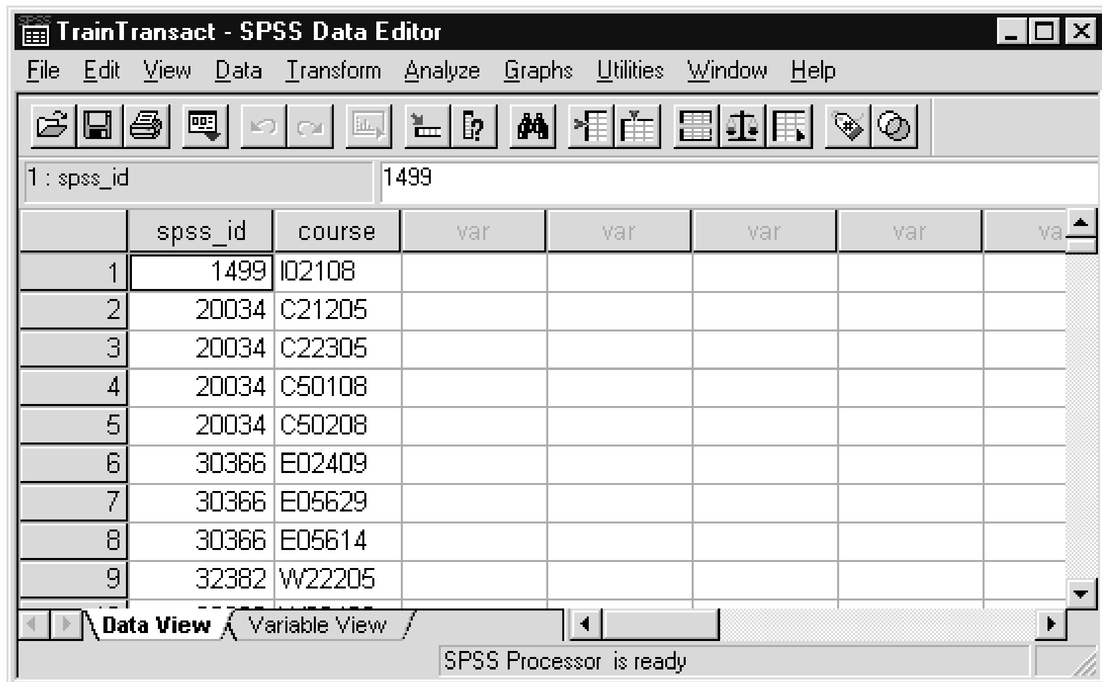
PRACTICAL EXAMPLE: CONSOLIDATING TRANSACTIONS

To demonstrate how SPSS data manipulation commands can be combined to answer a practical question, we present a question posed within the SPSS Training Department. There was interest in exploring what combinations of training classes customers take in the course of a year. However, attendance at an SPSS training class is recorded as a single sales transaction within the sales database, so a customer who took five training classes would appear on five sales transaction records. For the analysis, we needed a single record (case) per training customer, on which all training classes attended during the year would be recorded. In this section, the steps taken to reformat the data are detailed. We should note that additional data cleaning operations were performed (selecting only training class transactions within the specified time period; checking that only valid training course codes were present, etc.), but are not shown here in order to allow us to focus on the data reorganization aspect of the problem.

We begin with a sales transaction file containing sales for training classes.

> Click **File..Open..Data** (move to c:\Train\ProgSynMac directory)
> Double click on **TrainTransact**
> Click **No** if asked to save the contents of the Data Editor

Figure 5.15 Training-Class Sales Transactions

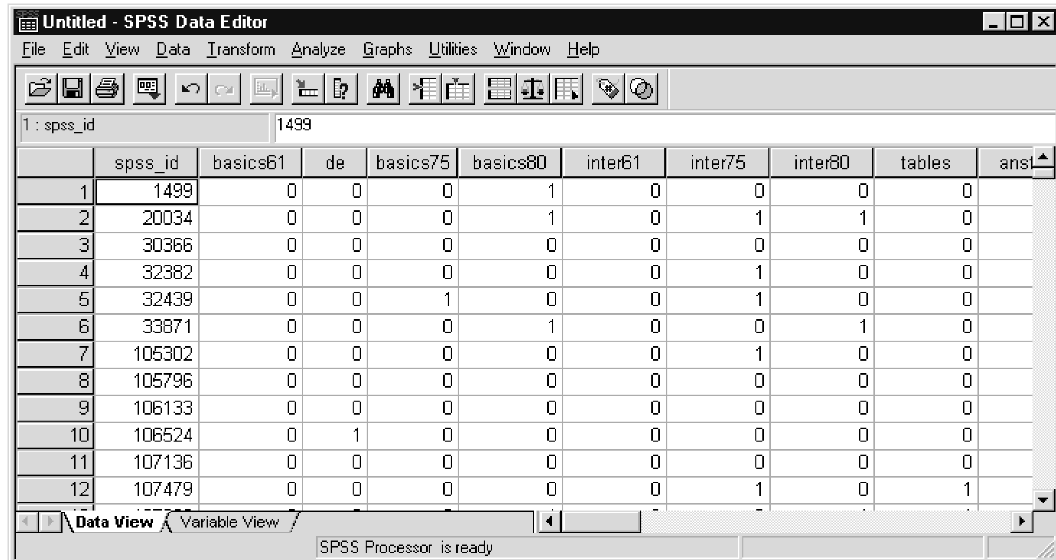


Each record, representing a single training class transaction, contains a course code and the customer SPSS ID number. Other fields (date, price, etc.) have been previously removed from the file. The COURSE variable is a 7-character string. The first character is a city location code. The second and third characters record the sequence number of the

course within the city during that year. Finally, the last three characters (positions 4-6) represent a 3-digit training course topic code number (for example 108 is “The Basics: SPSS for Windows 8.0”).

The goal is to reorganize the data so each customer appears in a single record containing a variable for every course, coded 1 if the customer took the course, and 0 otherwise. This target data structure is shown below.

Figure 5.16 Training Course Data Organized by Customer ID



	spss_id	basics61	de	basics75	basics80	inter61	inter75	inter80	tables
1	1499	0	0	0	1	0	0	0	0
2	20034	0	0	0	1	0	1	1	0
3	30366	0	0	0	0	0	0	0	0
4	32382	0	0	0	0	0	1	0	0
5	32439	0	0	1	0	0	1	0	0
6	33871	0	0	0	1	0	0	1	0
7	105302	0	0	0	0	0	1	0	0
8	105796	0	0	0	0	0	0	0	0
9	106133	0	0	0	0	0	0	0	0
10	106524	0	1	0	0	0	0	0	0
11	107136	0	0	0	0	0	0	0	0
12	107479	0	0	0	0	0	1	0	1

Now it is relatively easy to examine whether taking one course is related to another. It should be noted that if sequence or city information were deemed important, then this structure would not be adequate. This serves to reinforce the notion that no single data structure is convenient for all possible analyses, and the ability to restructure data, whether within a database or through a program like SPSS, is important.

So how do we go get to this data structure? The steps are summarized below.

1) Extract the 3-digit training course topic number from the COURSE variable and recode it to an integer value within a contiguous range (1-30).
2) Create a vector of course variables and for each training course transaction, set the appropriate course variable to 1.
3) Aggregate the course transaction file to the customer ID level.

Click **File..Open..Syntax** (move to c:\Train\ProgSynMac)
Double click on **TransactionAgg**
Scroll down to the **GET** command

Figure 5.17 TransactionAgg.sps Syntax File (Beginning)

TransactionAgg - SPSS Syntax Editor

File Edit View Analyze Graphs Utilities Run Window Help

```
Get File = 'c:\Train\ProgSynMac\TrainTransact.sav'.

* Extract the three-digit course code from the course field.

string scode(a3).
compute scode = substr(course,4,3).
compute code = number(scode,f3).
variable labels code 'Course Code'.
LIST   /CASES 30.

* Value labels are for reference purposes, they are not used in program.
* value labels code   102 'Basics 6.1'
   104 'Data Entry' 105 'Basics 7.5' 108 'Basics 8.0' 202 'Intermediate 6.1'
   205 'Intermediate 7.5' 208 'Intermediate 8.0' 222 'Tables' 227 'AnswerTree'
   302 'Adv. Topics 6.1' 305 'Adv. Topics 8.0' 306 'Scripting' 312 'Trends'
   320 'Categories' 409 'Stat. Analysis' 410 'TextSmart'
   510 'Adv. Stat Analysis' 518 'Anova' 519 'Regression'  605 'SigmaPlot'
   610 'Data Mining' 614 'Survey Research' 629 'Market Segmentation'
   701 'Qian' 703 'Teleform' 704 'Neural Connection' 901 'allClear'
```

SPSS Processor is ready | Ln 51 Col 12

The first task is to obtain the 3-digit training course topic code from the COURSE variable.

STRING: We create a new string variable (SCODE) three characters long (A3). Since SPSS assumes newly created variables are numeric, we must declare SCODE to be a string with the STRING command before using it in the COMPUTE command that follows. SCODE will store the training course topic code.

COMPUTE SCODE: The SUBSTR (substring) function will extract, from the COURSE variable, a 3-character string, beginning in position 4 and extending three characters. The result of this substring extraction is placed in the SCODE variable. If COURSE is “C22108 “, then SCODE will be “108”.

COMPUTE CODE: Since it will be easier to deal with numeric course topic codes, that is, it is easier to type 108 than ‘108’, we convert the string training course topic code stored in SCODE into a numeric form. This is accomplished with the NUMBER function and F3 (Fixed number 3 columns long) represents the format for the numeric variable CODE. The two COMPUTE commands could be consolidated into a single command, with the NUMBER function being directly applied to SUBSTR(COURSE,4,3), but the logic may be more clear when broken down into single steps.

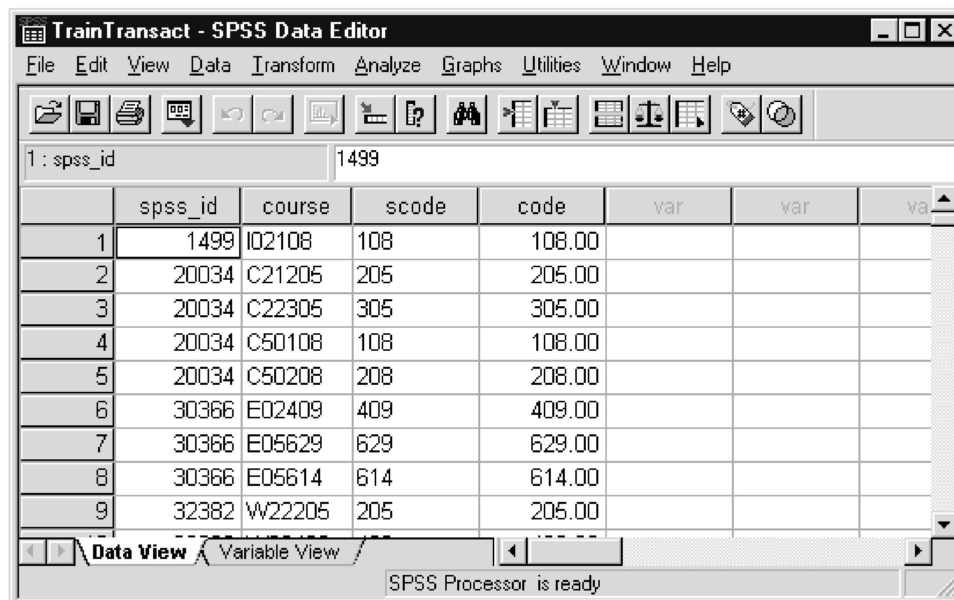
LIST: Since the transaction file contains thousands of records, we limit the number of cases displayed in the LIST command using the CASES subcommand. Only the first 30 cases will appear in the Viewer window.

Highlight the five lines from the **STRING** to the **LIST** command

Click the **Run** button

Switch to the **Data Editor** (click Goto Data tool)

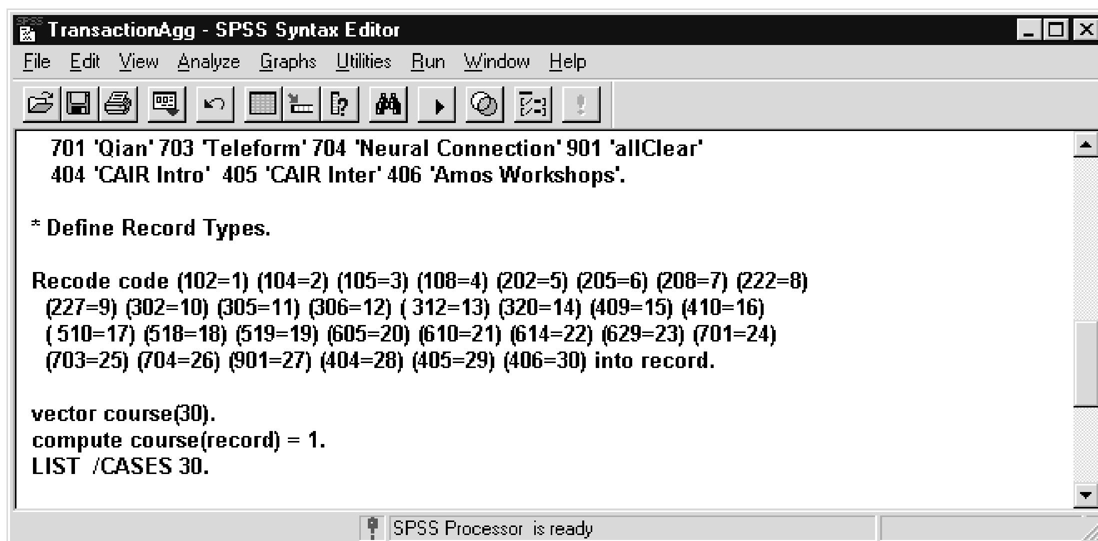
Figure 5.18 Training Course Topic Code as a Separate Field



	spss_id	course	scode	code	var	var	va
1	1499	I02108	108	108.00			
2	20034	C21205	205	205.00			
3	20034	C22305	305	305.00			
4	20034	C50108	108	108.00			
5	20034	C50208	208	208.00			
6	30366	E02409	409	409.00			
7	30366	E05629	629	629.00			
8	30366	E05614	614	614.00			
9	32382	W22205	205	205.00			

Switch to the **TransactionAgg – SPSS Syntax Editor** window
Scroll down to the ***Define Record Types.** command

Figure 5.19 Next Portion of TransactionAgg.sps Syntax File




```
   701 'Qian' 703 'Teleform' 704 'Neural Connection' 901 'allClear'
   404 'CAIR Intro'  405 'CAIR Inter' 406 'Amos Workshops'.

* Define Record Types.

Recode code (102=1) (104=2) (105=3) (108=4) (202=5) (205=6) (208=7) (222=8)
  (227=9) (302=10) (305=11) (306=12) ( 312=13) (320=14) (409=15) (410=16)
  ( 510=17) (518=18) (519=19) (605=20) (610=21) (614=22) (629=23) (701=24)
  (703=25) (704=26) (901=27) (404=28) (405=29) (406=30) into record.

vector course(30).
compute course(record) = 1.
LIST  /CASES 30.
```

* : Comments can be added to SPSS Syntax programs. Comments begin with an asterisk in column 1 and the remainder of that line, and any continuation lines (up to a command terminating period), are treated as a comment. Comments can also begin with /* and terminate with */, and such comments can be inserted within a command line.

RECODE: The RECODE statement converts the training course topic 3-digit codes into integer values from 1 to 30, which will be used as vector indices. These integer values are stored in a variable called RECORD (short for record type). Care must be taken to note which original training course topic code is mapped into which integer code. In this program they are recorded in a VALUE LABELS command just above the RECODE. Note that an AUTORECODE could be used in place of the RECODE here, but the output from the AUTORECODE, or the value labels from the new variable, would then need to be examined to determine the mapping of training course topic codes to the numbers 1 through 30.

VECTOR: A vector named COURSE is defined and its range set to 30, since there are 30 course topics used in the RECODE. Setting it too high would simply mean that there would be a few extra variables with all system-missing values; setting it too low would produce error messages since a vector subscript would be out of range.

COMPUTE: Since there is a single training class transaction per case, there is no need for a loop. Rather, the variable from the COURSE vector in the RECORD position is set equal to one. For example, the first case had a training course topic code (CODE) of 108. The RECODE command would assign it a value of 4 for the RECORD variable, so the fourth COURSE vector variable (COURSE(RECORD)) is assigned a value of 1.

Thus as each training transaction is processed, a single variable in the COURSE vector is set equal to one, while the others remain system missing. We see this below.

Highlight from the **RECODE** to the **LIST** command

Click the **Run** button

Switch to the **Data Editor** (click Goto Data tool)

Figure 5.20 Assigning Each Training Course to a Variable

TrainTransact - SPSS Data Editor

File Edit View Data Transform Analyze Graphs Utilities Window Help

1 : spss_id 1499

	spss_id	course	scode	code	record	course1	course2	course3	course4
1	1499	I02108	108	108.00	4.00	.	.	.	1.00
2	20034	C21205	205	205.00	6.00	.	.	.	.
3	20034	C22305	305	305.00	11.00	.	.	.	.
4	20034	C50108	108	108.00	4.00	.	.	.	1.00
5	20034	C50208	208	208.00	7.00	.	.	.	.
6	30366	E02409	409	409.00	15.00	.	.	.	.
7	30366	E05629	629	629.00	23.00	.	.	.	.
8	30366	E05614	614	614.00	22.00	.	.	.	.
9	32382	W22205	205	205.00	6.00	.	.	.	.
10	32382	W30409	409	409.00	15.00	.	.	.	.
11	32382	W33614	614	614.00	22.00	.	.	.	.
12	32439	C10105	105	105.00	3.00	.	.	1.00	.

Data View / Variable View

SPSS Processor is ready

Consistent with the RECODE command, we see that sales transactions of training course topic code 108 (The Basics: SPSS for Windows 8.0) now have a code of 1 in the COURSE4 variable (the fourth variable in the COURSE vector).

We face the now familiar problem of information from a single customer being spread across multiple records. As before, we will use the Aggregate procedure to consolidate all the training transactions for a customer into a single case.

Switch to the **TransactionAgg – SPSS Syntax Editor** window
Scroll down to the **AGGREGATE** command

Figure 5.21 Last Portion of TransactionAgg.sps Syntax File

TransactionAgg - SPSS Syntax Editor

File Edit View Analyze Graphs Utilities Run Window Help

```
aggregate outfile=*
   /break=spss_id
   /basics61 de basics75 basics80 inter61 inter75 inter80 tables
    anstree adv61 adv80 script trends categ stats textsm advstat
    anova regress sigplot datamine survey mkseg qian teleform
    neural allclear cairbas cairinte amos =
    first(course1 to course30).

format basics61 to amos (f1).
recode basics61 to amos (sysmis=0).
LIST /CASES 20.
```

SPSS Processor is ready

The AGGREGATE command does most of the work here.

AGGREGATE: The AGGREGATE command will produce a file containing one case for each customer (/BREAK = SPSS_ID) and it will replace the current file in the Data Editor window (/OUTFILE = *). The variable names in the aggregate file (i.e. Basics61, DE, …, Amos) are chosen to describe the relevant SPSS training products. These will be based on the COURSE1 to COURSE30 variables in the current file. Notice that the FIRST (first valid value) aggregate summary function is used; it will return the first valid value within the group formed by the BREAK variable's value. For a course, this value will be 1 if the customer took the course and system missing otherwise.

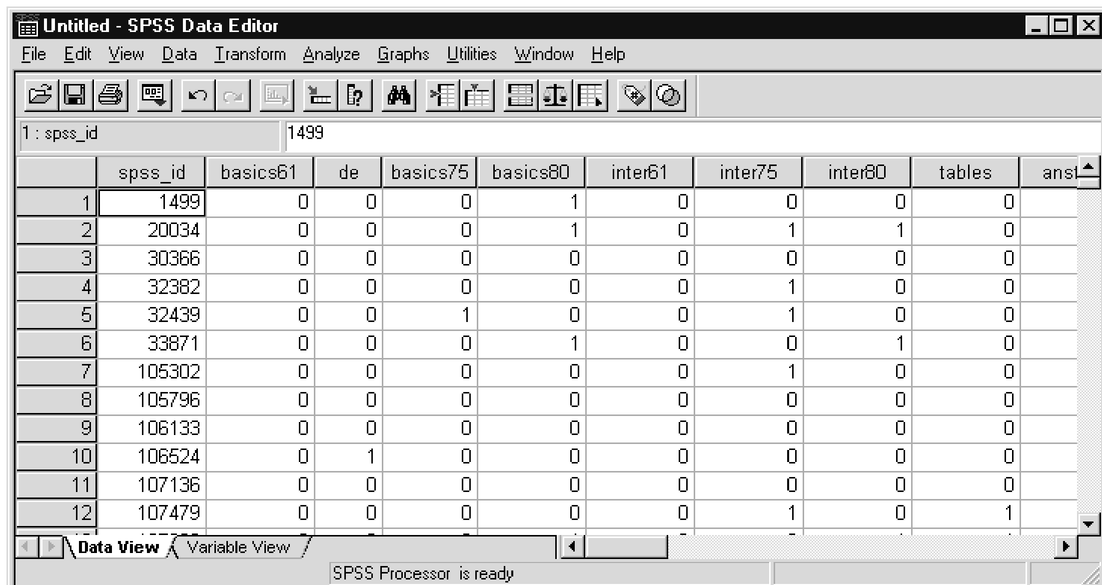
FORMAT: This changes the display and writing formats for the course variables to be as fixed numbers one column wide. Recall the default would be as a fixed number eight columns wide and with two decimals.

RECODE: If a training course were not taken by a customer we want the appropriate course variable's value to be 0, not system missing. This is because cases with system-missing values are excluded from many SPSS procedures (for example, Crosstabs). The RECODE command, which replaces system missing with 0 for the course variables, accomplishes this.

Highlight from the **AGGREGATE** to the **LIST** command
Click **Run..Selection**

Switch to the **Data Editor** (click the Goto Data tool)

Figure 5.22 Training Course Transactions Aggregated to Customer Level



	spss_id	basics61	de	basics75	basics80	inter61	inter75	inter80	tables
1	1499	0	0	0	1	0	0	0	0
2	20034	0	0	0	1	0	1	1	0
3	30366	0	0	0	0	0	0	0	0
4	32382	0	0	0	0	0	1	0	0
5	32439	0	0	1	0	0	1	0	0
6	33871	0	0	0	1	0	0	1	0
7	105302	0	0	0	0	0	1	0	0
8	105796	0	0	0	0	0	0	0	0
9	106133	0	0	0	0	0	0	0	0
10	106524	0	1	0	0	0	0	0	0
11	107136	0	0	0	0	0	0	0	0
12	107479	0	0	0	0	0	1	0	1

We now see, on an individual customer basis, which training courses were taken. Within SPSS we could now use Crosstabs, Tables, or the Categories procedure Homals, to explore training course patterns. Alternatively, Clementine contains dedicated procedures (rule association analysis) to explore such data.

SUMMARY

We discussed four programs that perform advanced data manipulation: reading comma–delimited files, showing two ways to read repeating data on one record, and converting transaction level data to customer level data (performing complex file manipulation to create a missing data report is in the Appendix). These few examples should provide some indication of the power of the SPSS transformation language.

APPENDIX: IDENTIFYING MISSING VARIABLES BY CASE

SPSS via the Frequencies procedure can easily provide output to note the categories and amount of missing information for each variable in a file. What if, instead, you wished to know which variables were missing for each case, and to automatically produce a report of this information? To see an example of this type of analysis, examine Figure 5.23.

In this small file of 7 cases and 10 variables, we can see that variables X7 and X9 are missing for case 1, X4 and X10 for case 2, and so forth. This type of information can be very handy in searching for missing data patterns and when doing multivariate analysis to understand what is causing some cases to be dropped from an analysis.

Figure 5.23 Missing Data Report by Variable

ID	NMISS	VNAMES1	VNAMES2	VNAMES3	VNAMES4	VNAMES5
1.00	2	X7	X9			
2.00	2	X4	X10			
3.00	2	X6	X7			
4.00	3	X2	X3	X6		
5.00	2	X2	X6			
6.00	5	X2	X3	X8	X9	X10
7.00	2	X4	X5			

Producing such a report from an SPSS data file, though, is a bit of a challenge. It will not require us to learn any new commands (except possibly AUTORECODE), but it will require us to visualize the file structure and the intermediate steps necessary to reach our goal. Programming this problem is an excellent example of the capability of SPSS in the hands of a knowledgeable user.

To create the report in Figure 5.23, we need to have a file that has one row for every case, which is already true in a standard SPSS data file, so that certainly doesn't sound difficult. But the tricky part is that the best way to create the output in Figure 5.23 is to have a series of variables—as many as in the original file—that are all strings, with the values of these strings the variable names with missing data from the original file.

To make this clear, we've displayed the target file in the Data Editor in Figure 5.24. The fact that the values in the columns beginning with the variable VNAMES1 are left justified indicates that these are string variables. This file is what we need to create.

Figure 5.24 SPSS Data File to Create Missing Variable Report

Untitled - SPSS Data Editor

File Edit View Data Transform Analyze Graphs Utilities Window Help

1 : id 1

	id	vnames1	vnames2	vnames3	vnames4	vnames5	vnames6	vnam
1	1.00	X7	X9					
2	2.00	X4	X10					
3	3.00	X6	X7					
4	4.00	X2	X3	X6				
5	5.00	X2	X6					
6	6.00	X2	X3	X8	X9	X10		
7	7.00	X4	X5					
8								
9								
10								
11								
12								
13								

Data View / Variable View

SPSS Processor is ready

So how do we get there? The answer involves several types of manipulation, but is summarized below.

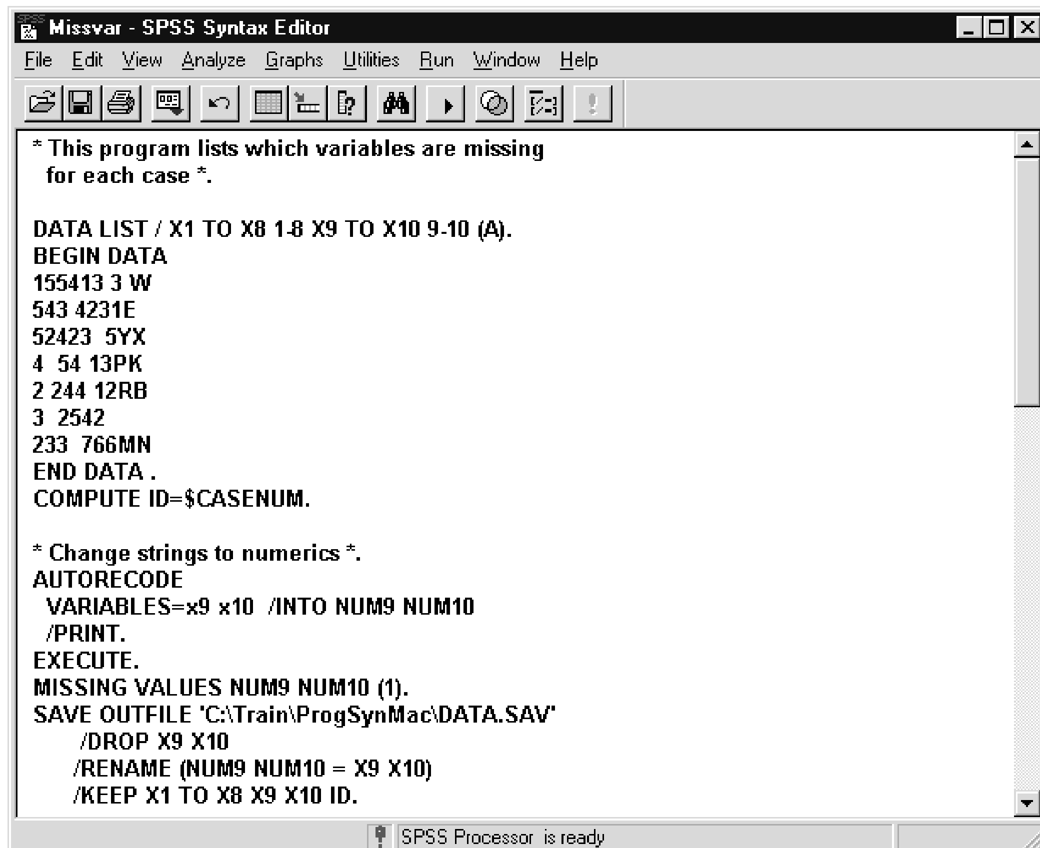
1) Save a copy of the original file, but make sure there is an ID variable.
2) Loop through this same file, creating a series of indices that identify which variables are missing for each case, and write this information out, plus the case ID, to a separate file.
3) Continue with the original file, select the first case, flip the file, and compute a new variable that contains each variable's position in the original file. Save this as a new file.
4) Get the file saved after looping and create a counter that increments by 1 for each additional variable missing for each case, sort the file by the variable position indicator, then match this file to the file saved in step 3 with a table match.
5) Take the resulting file and compute the new string variables that will hold the crucial information. Then aggregate this file, which has as many cases for each original case as there were missing variables, to a file with one case for each original case.

We don't expect that these actions will be intuitively obvious, unless you are a natural born programmer. We'll work through the program, block by block, to see how it functions.

Click on **File..Open..Syntax** (move to c:\Train\ProgSynMac)
Double-click on **Missvar.sps**

The first portion of the program reads in a small dataset with both numeric and string variables. In this file, blanks represent missing data in the lines between BEGIN DATA and END DATA. There are eight numeric variables and two string variables. An ID variable is computed by using $CASENUM, the SPSS system variable that records case number.

Figure 5.25 Missvar.sps Syntax File



The program is going to use the VECTOR command, but a vector must be composed entirely of numeric or string variables, not a mixture. Accordingly, the AUTORECODE command is used to change variables X9 and X10 into the numeric variables NUM9 and NUM10. AUTORECODE creates new values beginning with 1, where that value represents the first value of X9 or X10 in ascending alphabetic order. A blank goes first in the sort order, so it become a 1, and since the blank is missing data, we next include the MISSING VALUES command to code 1 as missing for NUM9 and NUM10.

This file is then saved by dropping the original X9 and X10 (the strings), renaming NUM9 and NUM10 back to X9 and X10 so we can retain the original variable names, and using the KEEP subcommand to place all the variables in consecutive order in the file before ID. This is done so that they can all be placed in one vector (because variables on a vector must be contiguous in the data file).

Run this syntax.

Highlight the lines from **DATA LIST** to **SAVE**

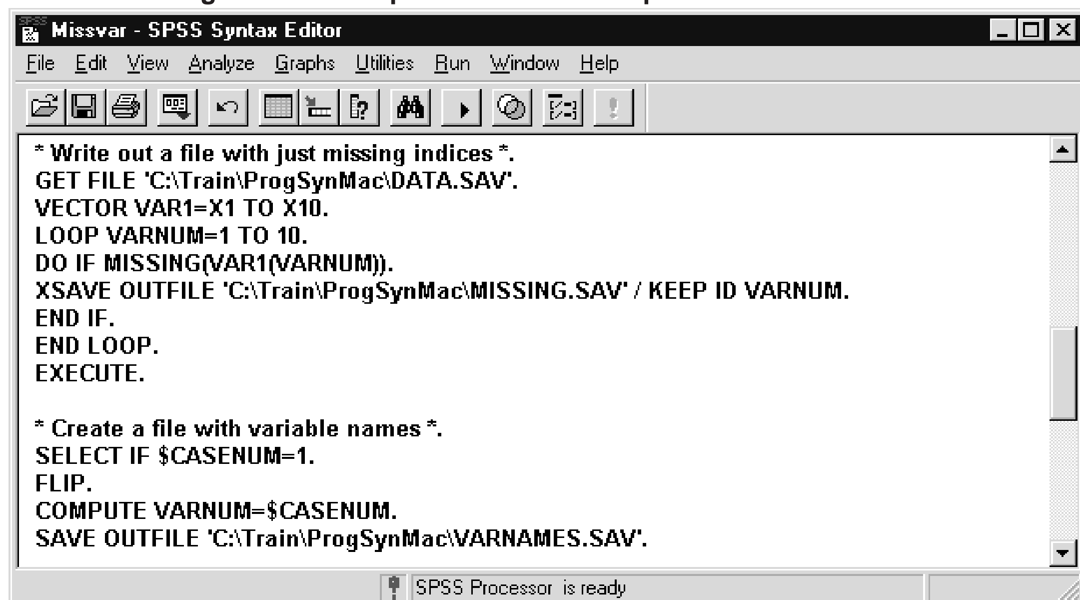
Click on the **Run** button

In the output from AUTORECODE (not shown), we can see that a blank has been recoded to a 1 for both NUM9 and NUM10.

With the original file saved in correct format, we now need to create a file that stores information on which variables are missing for which cases.

Switch to **Missvar.sps** (click Window..Missvar - SPSS Syntax Editor)
Scroll down to the next section of the program (as in Figure 5.26)

Figure 5.26 Next portion of Missvar.sps



The critical part of the whole program is next. First we must get the file DATA.SAV, which has the new file structure we need.

VECTOR: We create a vector with the 10 variables (in a file with more variables you would simply create a much larger vector).

LOOP: We loop 10 times because there are 10 variables. On each loop we test with the MISSING function on a DO IF to see if that vector element for that case is missing.

XSAVE: If the variable is missing, we write a case to the new SPSS data file (MISSING.SAV), keeping only ID and VARNUM. ID tells us the case and VARNUM tells us which variable is missing, so this file is an index of that information.

The EXECUTE statement forces execution of the transformations, including XSAVE.

Highlight the lines from **GET FILE** to **EXECUTE**

Click on the **Run** button

The Viewer echoes back the syntax from these commands. The important thing is the information in MISSING.SAV. Figure 5.27 shows this file. If you refer to Figure 5.24 you can see that there were two missing variables for the first case, variables X7 and X9. Therefore, MISSING.SAV has two rows for case 1, and the VARNUM for each represents which variables have missing values, 7 and 9. The second case has missing values for variables 4 and 10, and so forth.

Figure 5.27 MISSING.SAV Data File

Missing - SPSS Data Editor

File Edit View Data Transform Analyze Graphs Utilities Window Help

1 : id 1

	id	varnum	var	var	var	var	var
1	1.00	7.00					
2	1.00	9.00					
3	2.00	4.00					
4	2.00	10.00					
5	3.00	6.00					
6	3.00	7.00					
7	4.00	2.00					
8	4.00	3.00					
9	4.00	6.00					
10	5.00	2.00					
11	5.00	6.00					
12	6.00	2.00					
13	6.00	3.00					
14	6.00	8.00					

Data View / Variable View

SPSS Processor is ready

We've now created half of what we need. In some respects, in fact, this file is an answer to the original question, but it has as many cases as there are occurrences of missing data (18 in this instance), and this would be unwieldy for larger files. Also, MISSING.SAV is not in a format that makes it easy to create reports summarized by case.

We continue by creating a small file with the variable names.

Switch to the **Data Editor (**click Goto Data tool **)**

The working data file has the information we need, but only if we flip, or transpose, the file. Transpose creates a new data file in which the rows and columns in the original data file are transposed so that cases (rows) become variables and variables (columns) become cases. SPSS automatically creates new variable names and displays a list of the new variable names. A new string variable, CASE_LBL, containing the original variable names is also automatically created.

Figure 5.28 Current Working Data File

Data - SPSS Data Editor

File Edit View Data Transform Analyze Graphs Utilities Window Help

1 : x1 | 1

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	id	varnum	var
1	1	5	5	4	1	3	.	3	1	5	1.00	11.00	
2	5	4	3	.	4	2	3	1	2	1	2.00	11.00	
3	5	2	4	2	3	.	.	5	6	6	3.00	11.00	
4	4	.	.	5	4	.	1	3	4	3	4.00	11.00	
5	2	.	2	4	4	.	1	2	5	2	5.00	11.00	
6	3	.	.	2	5	4	2	.	1	1	6.00	11.00	
7	2	3	3	.	.	7	6	6	3	4	7.00	11.00	
8													
9													
10													

Data View / Variable View

SPSS Processor is ready

We don't need all the data in this file, just the labeling information, so we will select out the first case and transpose the file.

Switch to **Missvar.sps** (click Window..Missvar - SPSS Syntax Editor)

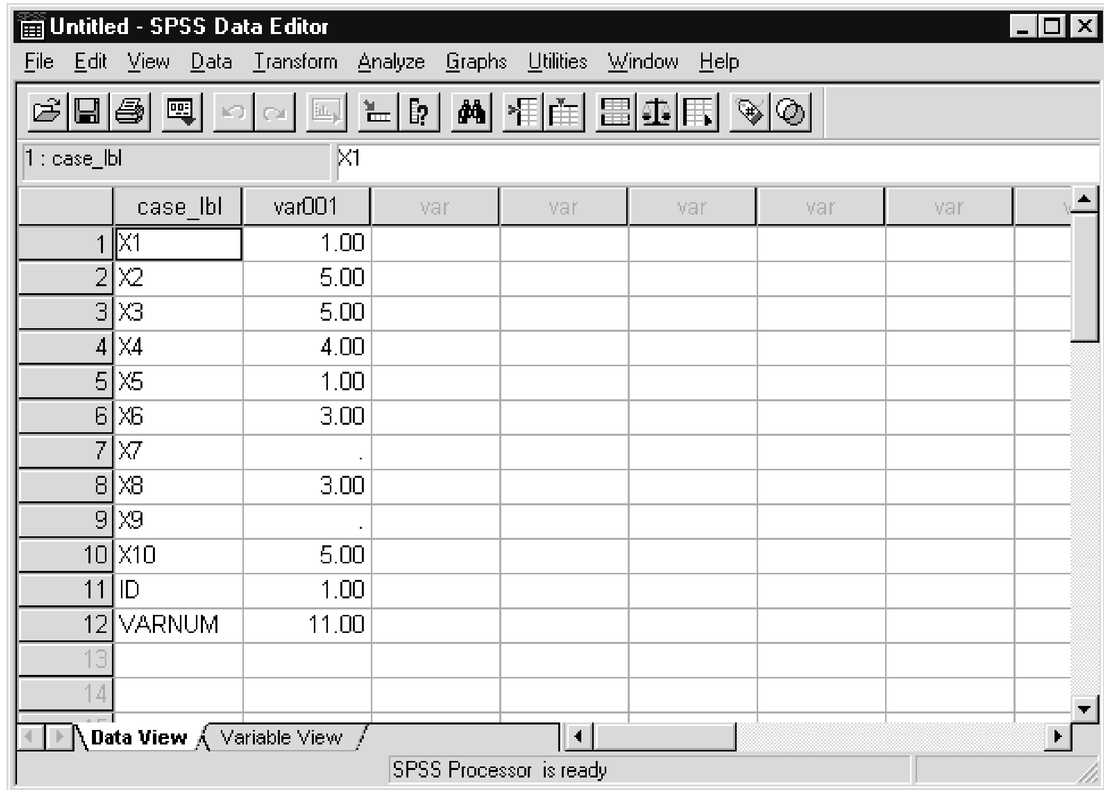
Highlight the two lines **SELECT IF** and **FLIP**

Click the **Run** button

Switch to the **Data Editor (**click Goto Data tool**)**

We now have the file you see in Figure 5.29. This file has the variable CASE_LBL that contains the original variable names. It also has a column labeled VAR001 that was the first row of data, which is unimportant. For purposes of matching this file back to the MISSING.SAV file, we need to create a case ID (actually a variable ID). Since the first ten values of CASE_LBL are X1 to X10, we can create such a variable by using the SPSS system variable $CASENUM as before.

Figure 5.29 Transposed Data File



Switch to **Missvar.sps** (click Window..Missvar - SPSS Syntax Editor)
Highlight **COMPUTE** and **SAVE**

Click on the **Run** button

Notice we called the ID variable VARNUM because it is really an ID variable for the position of a variable, and we gave it that name because the same name is used in the MISSING.SAV file.

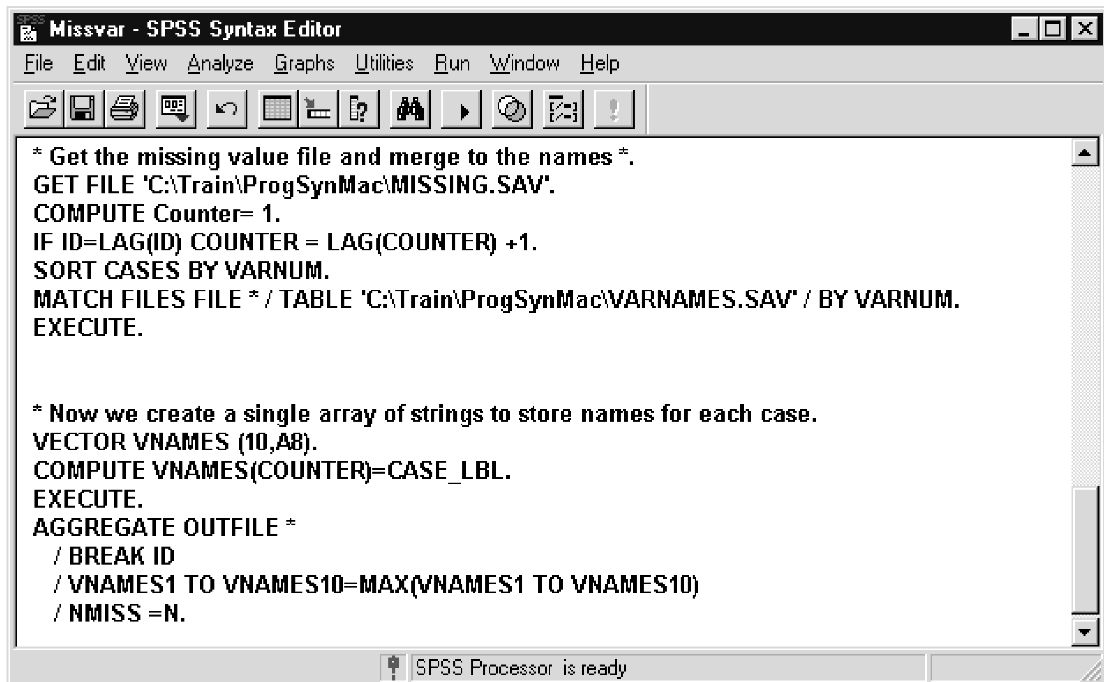
Switch to the **Data Editor (**click Goto Data tool)

Compare the working data file in the Data Editor (not shown) to Figure 5.27 that displays the MISSING.SAV file. We are going to match the CASE_LBL information from the file VARNAMES.SAV by VARNUM to each case in MISSING.SAV. In other words, for the first case in MISSING.SAV, with VARNUM=7, the value of "X7" will be added to the file. For the second case, the value of "X9" will be added, etc. This action will place the correct variable names on MISSING.SAV. We do this through a table match.

In addition, a counter will be created that increments by one for each case (each case represents a missing value) but resets to 1 when the ID variable in MISSING.SAV changes. We do this for purposes of reporting, not for the match itself.

Switch to **Missvar.sps** (click Window..Missvar - SPSS Syntax Editor)
Scroll down until you see the commands in Figure 5.30

Figure 5.30 Matching Portion of MISSVAR.SPS



We must also sort the MISSVAR.SAV file by VARNUM because it is not in that order now. The variable COUNTER is created in the standard manner.

Highlight the lines from **GET FILE** to **EXECUTE**

Click on the **Run** button

Switch to the **Data Editor (**click Goto Data tool **)**

The resulting file is, of course, ordered by VARNUM. The COUNTER is out of its original order, but that won't be a problem. We've added the Variable CASE_LBL to MISSING.SAV so that we can see, for example, that for cases 4, 5, and 6, each is missing on variable X2.

Figure 5.31 Data File After Table Match

Missing - SPSS Data Editor

File Edit View Data Transform Analyze Graphs Utilities Window Help

1 : id 4

	id	varnum	counter	case_lbl	var001	var	var	va
1	4.00	2.00	1.00	X2	5.00			
2	5.00	2.00	1.00	X2	5.00			
3	6.00	2.00	1.00	X2	5.00			
4	4.00	3.00	2.00	X3	5.00			
5	6.00	3.00	2.00	X3	5.00			
6	2.00	4.00	1.00	X4	4.00			
7	7.00	4.00	1.00	X4	4.00			
8	7.00	5.00	2.00	X5	1.00			
9	3.00	6.00	1.00	X6	3.00			
10	4.00	6.00	3.00	X6	3.00			
11	5.00	6.00	2.00	X6	3.00			
12	1.00	7.00	1.00	X7	.			
13	3.00	7.00	2.00	X7	.			

Data View / Variable View

SPSS Processor is ready

The hard part of our work is over. If you refer to Figure 5.24, you will see a file with only seven cases, one for each of the original cases, and a column for each of the ten original variables. To create such a file from the current working data file, we need to use the AGGREGATE command, breaking on ID, and spreading the CASE_LBL information into separate variables.

We accomplish the second task first by using a vector.

Switch to **Missvar.sps** (click Window..Missvar - SPSS Syntax Editor)

Referring back to Figure 5.30, the VECTOR command creates the vector VNAMES which has 10 string elements, each with format A8. We then place the CASE_LBL values into VNAMES, using COUNTER as the placeholder. To see how this works, look at Figure 5.31.

When COUNTER=1, this variable is the first missing variable for that case (in order from X1 to X10). This is true for cases 4, 5, and 6 for X2, so the value "X2" is placed in VNAMES(1) for those three cases.

Highlight the commands from **VECTOR** to **EXECUTE**

Click the **Run** button

Switch to the **Data Editor** (click Goto Data tool)

In the Data Editor (not shown), you can see how 10 new variables have been created from the vector VNAMES and how the values of CASE_LBL have been placed in the correct slots in VNAMES.

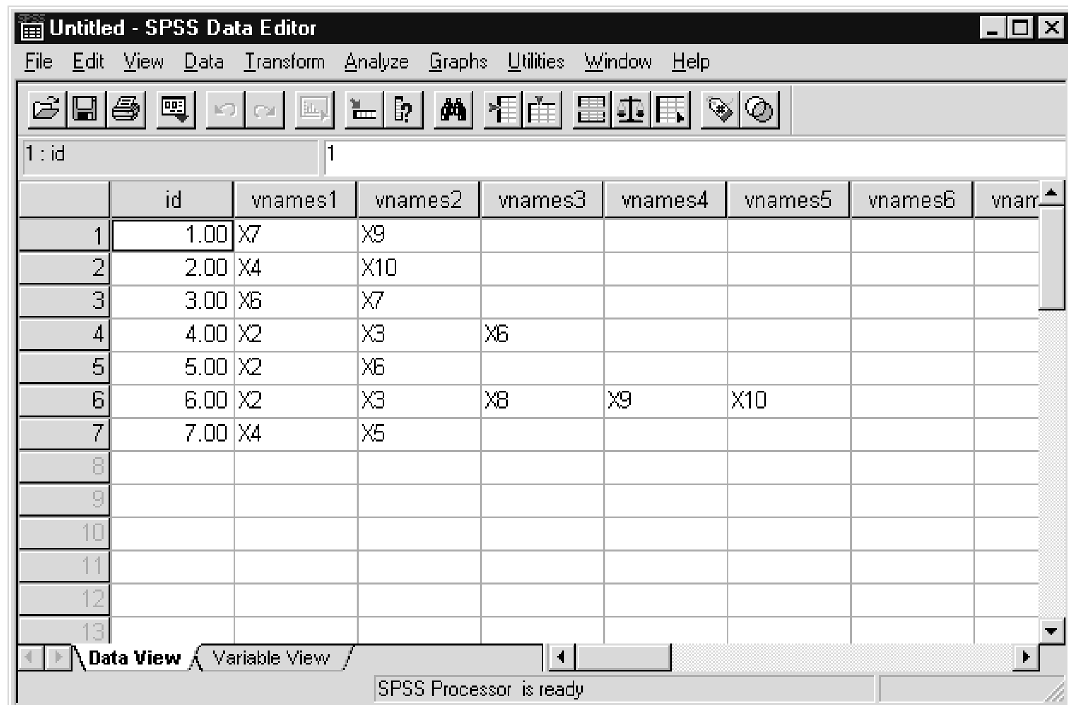
The final step is to use AGGREGATE. The MAX function is one way to take the values of VNAMES1 to VNAMES10 and place them into variables with the same names (we could also have used the MIN function). Although the VNAMES variables are strings, the MAX function can be applied. The N function puts the number of cases (here the number of missing variables) into the variable NMISS.

Switch to **Missvar.sps** (click Window..Missvar - SPSS Syntax Editor)
Click on the **AGGREGATE** command
Click on the **Run** button

Switch to the **Data Editor** (click Goto Data tool)

We have created the target file structure, with one case for each case in the original data file, and 10 string variables which store the names of the missing variables for each case. A report can now be produced like Figure 5.23.

Figure 5.32 Data File With Missing Variable Information



	id	vnames1	vnames2	vnames3	vnames4	vnames5	vnames6	vnam
1	1.00	X7	X9					
2	2.00	X4	X10					
3	3.00	X6	X7					
4	4.00	X2	X3	X6				
5	5.00	X2	X6					
6	6.00	X2	X3	X8	X9	X10		
7	7.00	X4	X5					
8								
9								
10								
11								
12								
13								

To make the syntax easier to write, we used variable names X1 to X10. You are likely to have less generic variable names, and if so, you will simply need to specify them at the appropriate places in the program. If you don't want to do the AUTORECODE for the string variables and add them to the numeric variables, you can do a separate report for each type

of variable. And if you have many variables, you could use more than one vector command.

Hopefully, this program provided you some hints and examples of how to create elaborate SPSS programs that do data manipulation.

Note: The SPSS Missing Values module produces a report similar to the output from this program. It also contains algorithms for imputing (substituting values) for missing data.

Chapter 6 Introduction to Macros

Topics

Introduction
Macro Basics
Macro Arguments
Macro Tokens
Viewing a Macro Expansion
Keyword Arguments
Using a Varying Number of Tokens
When Things Go Wrong

INTRODUCTION

Macros in most software programs are small routines of commands that automate one or more tasks to make your work more efficient and easier. Using a macro means not having to construct commands each time you need to do a particular analysis or data transformation task.

The SPSS Macro facility provides a means of constructing SPSS commands and controlling their execution. Thus, the output of a macro is a new set of tailored SPSS commands. Macros are most useful when one or more SPSS programs contain repeating blocks of identical or similar commands, but a macro can be useful in several different contexts. For example, a macro can be used to:

1) Issue a series of the same or similar commands repeatedly, using looping constructs.
2) Produce output from several procedures with a single command;
3) Change specifications on several procedures at once.

As noted in Chapter 2, SPSS macros are a bit different than those you may have encountered in other software. Although the SPSS macro facility has its own language, you don't invoke a macro editor to create macros. And SPSS macros don't directly automate actions you take with a mouse. Instead, SPSS macros are created like any other syntax file, then executed in the same manner as other syntax. SPSS macros always contain regular SPSS syntax in addition to the specialized macro commands.

MACRO BASICS

The structure of a macro is outlined below. Macros begin with the DEFINE command and end with the !ENDDEFINE command. Macro subcommands and keywords all begin with an exclamation point to distinguish them from regular SPSS commands. The macro name (which can also begin with an exclamation point) will be used in the macro call.

```
DEFINE   macro name    macro arguments and specifications
         are placed here

macro body

!ENDDEFINE
.
```

The macro body can include SPSS commands, parts of commands, or macro statements.

Once it has been constructed, using a macro is a straightforward process. First, the macro has to be processed like any other syntax file. This defines the macro to SPSS. Then you must do a *macro call* where you specify the name of the macro and any necessary arguments (such as variable names). The call, too, is equivalent to an SPSS command, so it is also executed like any other syntax.

The macro definition commands are placed into a standard SPSS syntax file, though they do not have to be in the same syntax file as the macro call. As a consequence, the syntax to use an existing macro requires just two commands, as shown next. The INCLUDE command places the commands in the named file into the SPSS command stream. In this instance, the INCLUDE command brings the macro definition commands in MACRO.SPS into SPSS. The macro call is next, using the macro name (MYMACRO) defined in MACRO.SPS. Any arguments on MYMACRO are placed after the name, and the macro call ends with a period, like all SPSS commands.

```
INCLUDE 'PathToMacro\MACRO.SPS'.
MYMACRO  arguments.
```

Highlighting these two commands and clicking on the Run button will execute the macro. Again, no special macro editor needs to be invoked to execute macros in SPSS. And a macro call does not have to immediately follow a macro definition.

Interestingly, the macro facility does not build and execute commands by itself; rather, it expands strings in a process called *macro expansion*. After the strings are expanded, the commands that contain the expanded strings are executed as part of the SPSS command sequence. This may seem rather esoteric knowledge, but knowing that the macro facility manipulates strings is important when creating complex macros and deciphering their errors.

MACRO ARGUMENTS

There are many subcommands and specifications available in the macro facility, but macro *arguments* are central to macro creation. Arguments are input to the macro that will be supplied by the user, and they can be of two types: keyword and positional.

Keyword arguments are assigned names in the macro definition, and when the macro is called, they are literally identified by name and thus can be given in any order.

Positional arguments are declared after the keyword !POSITIONAL; when the macro is called, they are identified by their relative position within the macro definition. The first positional argument is referred to by "!1" in the macro body, the second by "!2", and so forth.

MACRO TOKENS

Tokens are used in conjunction with arguments in macros. A *token* is a character or group of characters that has a predefined function in a specified context. That definition probably doesn't seem too illuminating because tokens are so varied in their operation. In essence, a token declaration tells SPSS how to know or recognize which elements on the macro call should be associated with this argument. The simplest token definition is one that assigns the next n values (or tokens) to the argument. It is better to see this in action where we can also discuss other issues of macro syntax.

In the macro below, we'll focus first on the argument and tokens. The arguments to a macro must be enclosed in parentheses. The use of the !POSITIONAL argument keyword tells SPSS that arguments, or input, to the macro will be positional in the macro call. The !TOKENS (4) specification declares that there will be four positional tokens.

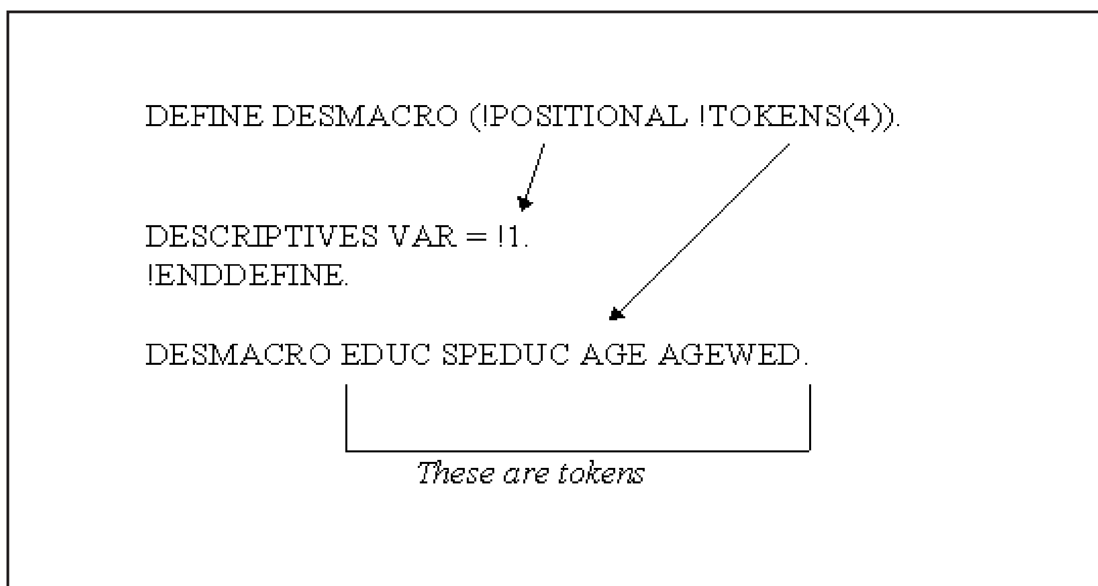
```
DEFINE DESMACRO  (!POSITIONAL !TOKENS (4)).
DESCRIPTIVES  VAR = !1.
!ENDDEFINE.

DESMACRO EDUC SPEDUC AGE AGEWED.
```

Next comes the body of the macro, which is a simple DESCRIPTIVES command, with one exception. We replace the usual list of variables with the keyword !1, which refers to the first (and only) positional argument in the macro. When SPSS expands this macro, it will place the four positional tokens in place of the !1. This means, in this instance, that they must be variables for the command to execute properly.

The macro closes with the !ENDDEFINE command. The macro DESMACRO is now defined (or at least will be once SPSS runs these statements). We call the macro by using its name and then supplying four positional arguments, which are variable names.

The diagram in Figure 6.1 may make the relationship between arguments and tokens more clear. The keyword !POSITIONAL, an argument, refers to where in the macro body the tokens will be placed. Hence !1, the first positional argument, is placed after the equals sign. The !TOKENS (4) specification on the argument is not referred to in the macro body; instead, it tells SPSS how many elements in the positional argument will be input by the user. In the macro call, the four variable names are not just variable names, but also tokens.

Figure 6.1 Outline of Macro Structure



We will use the 1994 General Social Survey file to demonstrate how this macro functions. We open the GSS file and then run this macro.

Click **File..Open..Data**
Switch to the **c:\Train\ProgSynMac** directory if necessary
Double-click on **GSS94**

Then

Click **File..Open..Syntax**
Double-click on **MACRO1** (not shown)

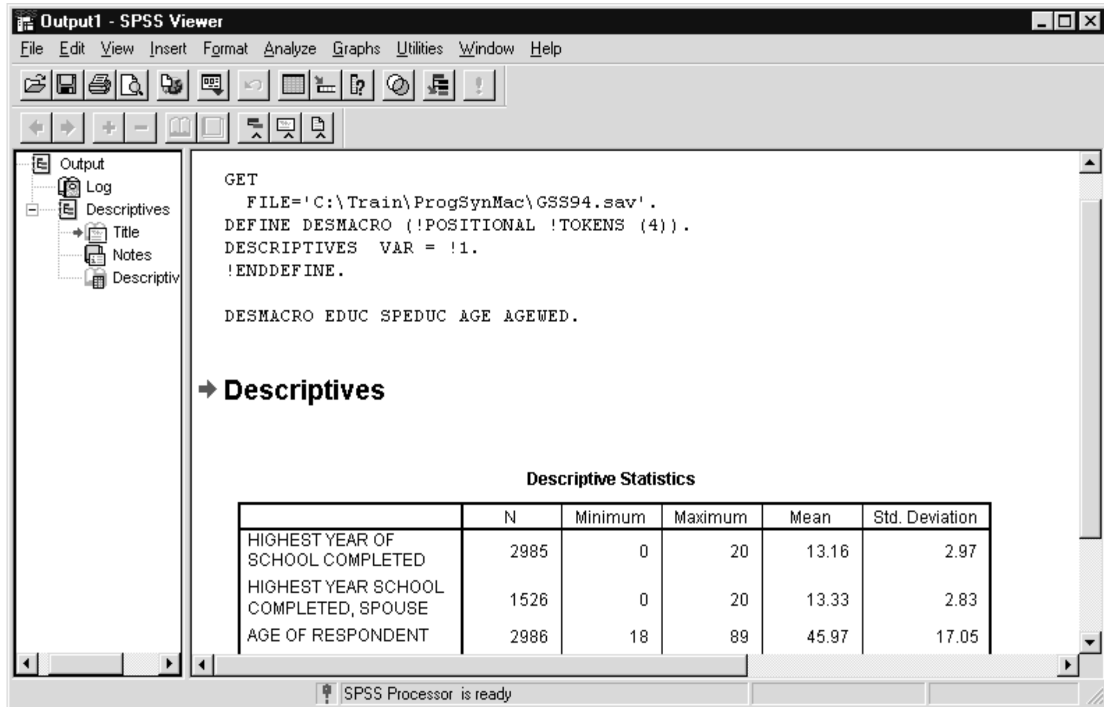
Running a macro involves defining it, then calling it with any necessary input. To do so,

Highlight the lines from **DEFINE to DESMACRO**

Click on the **Run** tool button

SPSS processes all the commands, echoing them to the Viewer. It then runs a DESCRIPTIVES command that has been created by the macro facility for the four variables we specified.

Figure 6.2 Descriptives Output from DESMACRO



	N	Minimum	Maximum	Mean	Std. Deviation
HIGHEST YEAR OF SCHOOL COMPLETED	2985	0	20	13.16	2.97
HIGHEST YEAR SCHOOL COMPLETED, SPOUSE	1526	0	20	13.33	2.83
AGE OF RESPONDENT	2986	18	89	45.97	17.05

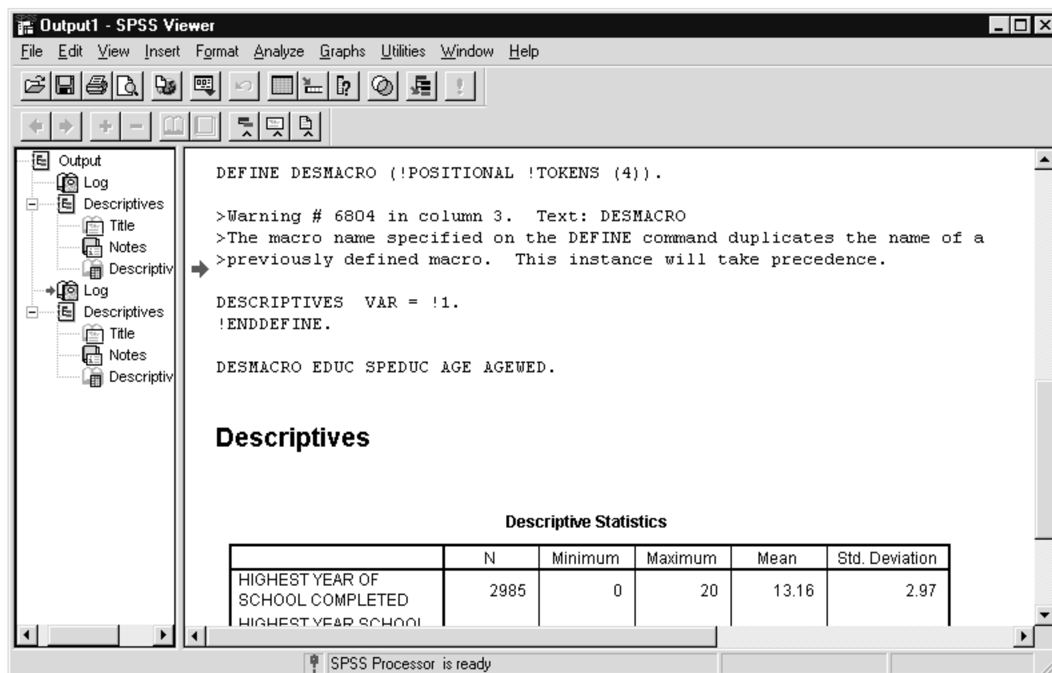
That's all there is to using a macro! Understandably, this was a lot of work for a DESCRIPTIVES command, but it's best to begin with simple macros to illustrate their features and operation.

What happens when you define the same macro twice? Let's find out by rerunning the commands.

Switch to the M**ACRO1** Syntax (click Window..Macro1 - SPSS Syntax Editor)
Highlight the lines from **DEFINE to DESMACRO** again
Click the **Run** tool button

First, note that we got another set of output from DESCRIPTIVES, so clearly running the program again didn't seem to cause any major problems. However, we received a warning message that politely tells us that DESMACRO has previously been defined, but that this new instance will take precedence.

Figure 6.3 Warning Message From Defining DESMACRO Twice



When you are writing macros you will undoubtedly see warning 6804 now and then. Warnings can be turned off with the SET command, but we don't recommend that approach, especially when debugging your macro definitions. Clearly, this warning means that once defined in an SPSS session, a macro need not be defined again.

VIEWING A MACRO EXPANSION

When constructing complex macros, it is usually necessary to see the exact commands created by the macro facility. This is particularly crucial when debugging macros. You can turn this option on with the SET command,

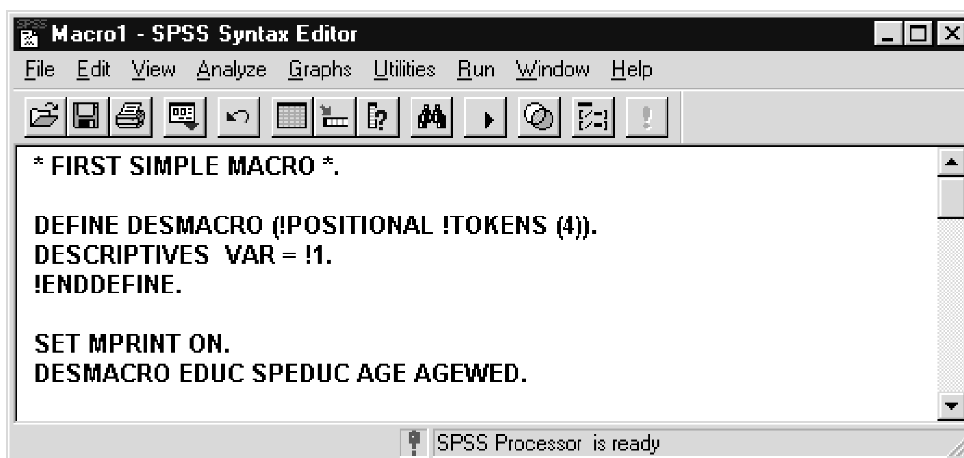
SET MPRINT ON.

We'll try this to see the result for DESMACRO.

Switch to the M**ACRO1** Syntax (click Window..Macro1 - SPSS Syntax Editor)
Add the line **SET MPRINT ON.** just above the DESMACRO call
Highlight only these **two lines**

Click on the **Run** tool button

Figure 6.4 SET MPRINT ON. Added



In the Viewer, the lines beginning with "M>" appear due to the SET command. The macro call appears just above these lines, and the DESCRIPTIVES command constructed by the macro appears below. The four positional tokens (EDUC, SPEDUC, AGE, and AGEWED) are placed, in that order, on the DESCRIPTIVES command just after the equals sign (which is where the !1 positional argument was in the macro definition).

Figure 6.5 Macro Expansion of DESMACRO

```
SET MPRINT ON.
DESMACRO EDUC SPEDUC AGE AGEWED.
  24 M>
  25 M>  .
  26 M>  DESCRIPTIVES VAR = EDUC SPEDUC AGE AGEWED
  27 M>   .
```

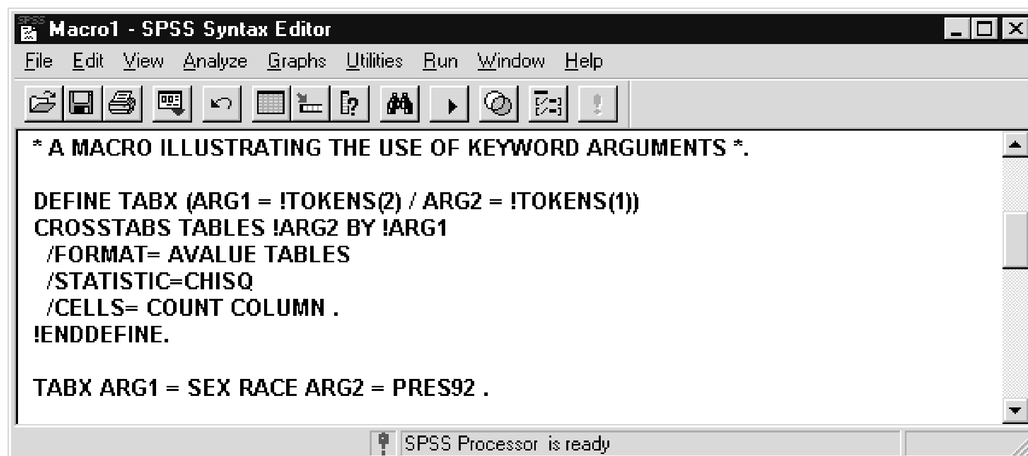
KEYWORD ARGUMENTS

Since arguments are central to macros, we next review the syntax for a keyword argument. This type of argument is given a user-defined keyword in the macro definition. In the macro body the argument name is preceded by an exclamation point, so keyword arguments should be seven characters or less in length. On the macro call, though, the keyword is specified without the exclamation point (when and when not to use an exclamation point is often a point of confusion for those new to macros).

Switch to the M**ACRO1** Syntax (click Window..Macro1 - SPSS Syntax Editor)
Scroll down until you see the program in **Figure 6.6**

This macro is called TABX and allows the user to create a crosstabulation of two column variables and one row variable. There are two keyword arguments defined, ARG1 and ARG2. The first is defined with two tokens and the second with one token. The arguments are separated by a slash, and the whole set is enclosed in parentheses.

Figure 6.6 TABX Macro Definition



The body of the macro consists of a standard CROSSTABS command, but as with the previous example, the variables are replaced with arguments. The keyword !ARG2 is placed where the row variables go, and !ARG1 is put in the column variable position. They are both preceded with exclamation points because they are in the macro body. The CROSSTABS command also requests column percentages and the chi–square statistic.

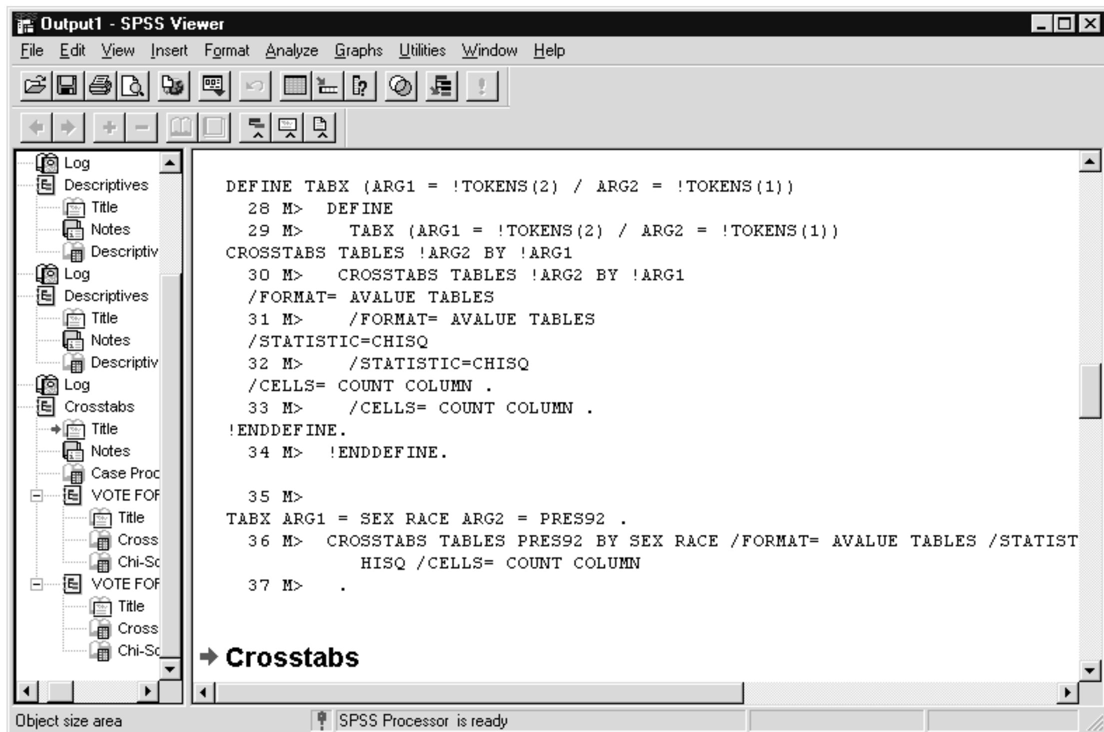
Notice that !ARG1 appears after !ARG2 in the macro body. Keyword arguments can appear in any order in the macro. When the macro is called, the three variables are input as arguments, two for ARG1 and one for ARG2. Two crosstabulations will be produced, PRES92 by SEX and PRES92 by RACE.

Highlight the lines from **DEFINE TABX** to **TABX**

Click on the **Run** tool button

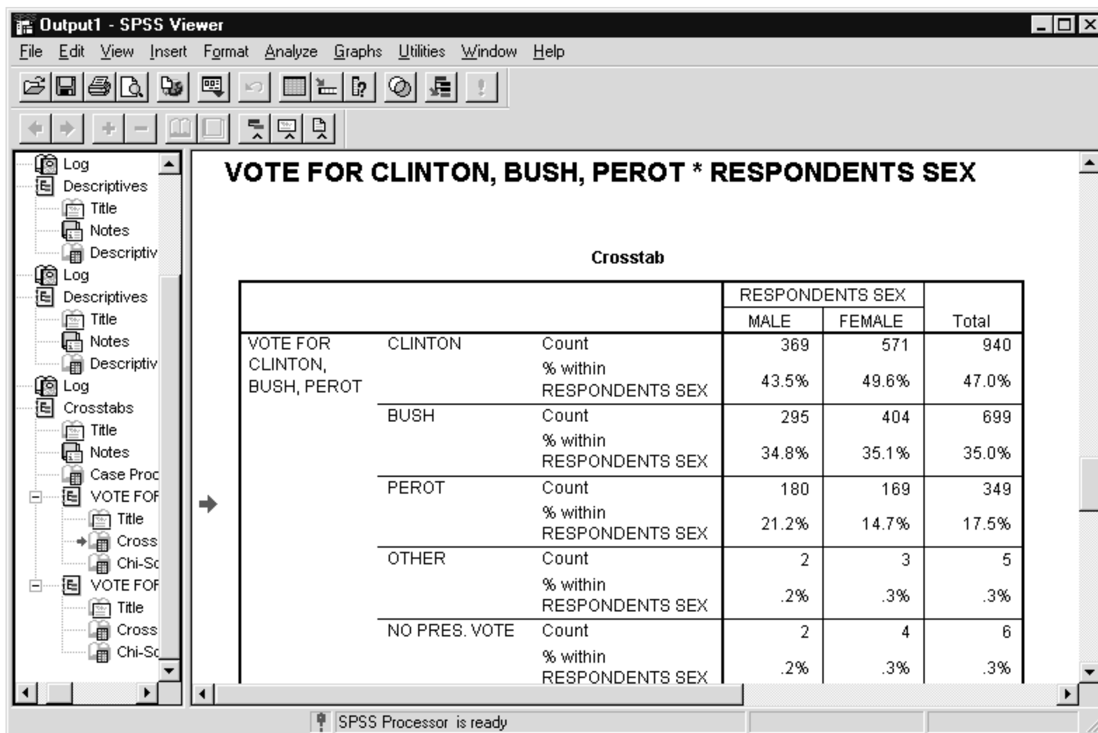
When the output is produced, the first thing to observe is that the macro expansion is displayed not just for the macro call, but also for the macro definition itself. This is not a problem, but it can be annoying. You can turn this off by placing a SET MPRINT OFF command before DEFINE, and then a SET MPRINT ON before the macro call.

Figure 6.7 Macro Expansion for Macro Definition and Call



If you scroll down a bit, you will see the table in Figure 6.8. The macro facility created the CROSSTAB command shown in Figure 6.7, which when executed by SPSS created two tables. In the first table, we see that females were more likely to vote for Clinton in 1992, and males more likely to vote for Perot.

Figure 6.8 Crosstabs of 1992 Presidential Vote and Sex



			RESPONDENTS SEX		
			MALE	FEMALE	Total
VOTE FOR CLINTON, BUSH, PEROT	CLINTON	Count	369	571	940
		% within RESPONDENTS SEX	43.5%	49.6%	47.0%
	BUSH	Count	295	404	699
		% within RESPONDENTS SEX	34.8%	35.1%	35.0%
	PEROT	Count	180	169	349
		% within RESPONDENTS SEX	21.2%	14.7%	17.5%
	OTHER	Count	2	3	5
		% within RESPONDENTS SEX	.2%	.3%	.3%
	NO PRES. VOTE	Count	2	4	6
		% within RESPONDENTS SEX	.2%	.3%	.3%

USING A VARYING NUMBER OF TOKENS

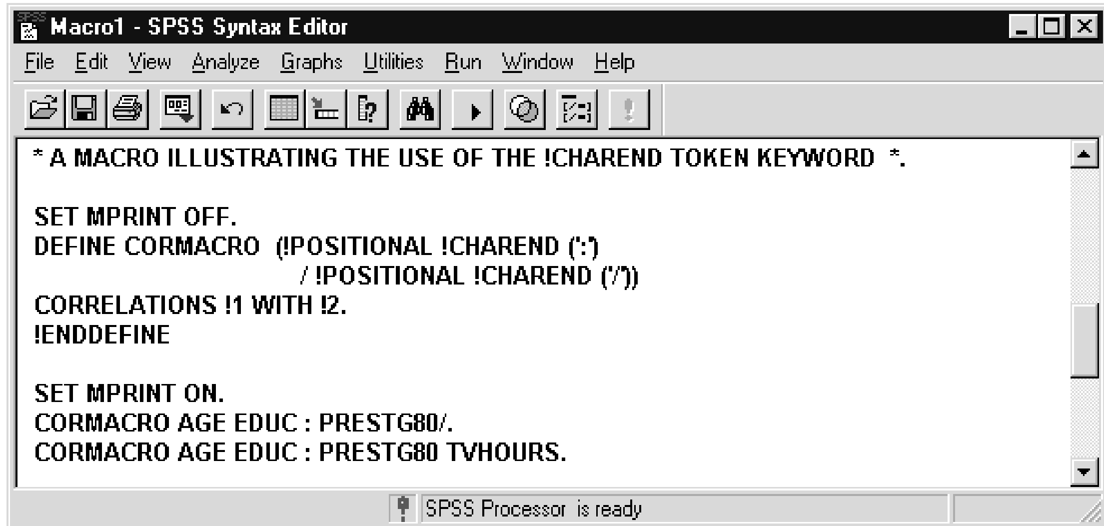
What if you wish to use a different number of inputs, or tokens, to the macro arguments? You might want to create crosstab tables with four row variables and three column variables the next time you call TABX. This is easily accomplished with the !CHAREND or the !ENCLOSE keywords, both of which we will briefly review.

The next series of commands in MACRO1.SPS define a macro named CORMACRO that calculates correlation coefficients for a series of variables.

> Switch to the M**ACRO1** Syntax (click Window..Macro1 - SPSS Syntax Editor)
> **Scroll** down until you see the **CORMACRO definition**

The macro definition contains two positional arguments. The first uses the !CHAREND keyword and in quotes specifies a colon (:) as the ending character. !CHAREND assigns all tokens up to the specified character to the first positional argument, so no exact number of tokens is specified. The second positional argument also uses !CHAREND, assigning all tokens up to the slash (/) to the second positional argument. The SPSS CORRELATIONS command is the macro body, with the two positional arguments on either side of the WITH keyword. We have also first turned macro printing off, then back on, to avoid extraneous output.

Figure 6.9 CORMACRO Macro Definition



The first call of the macro assigns the variables AGE and EDUC to the first positional argument. This is because the two variables are followed by a colon, which tells the macro facility that input to the first positional argument has ended. The second positional argument has only PRESTG80, ended by a slash.

Before running this we should verify that the Correlation Autoscript (Autoscripts are discussed in the *Programming Scripts Using SPSS* course) is not enabled. To do so:

Click **Edit..Options**, then click the **Scripts** tab
Click **Enable Autoscripting** checkbox, if necessary, to **uncheck** it
Click **OK**

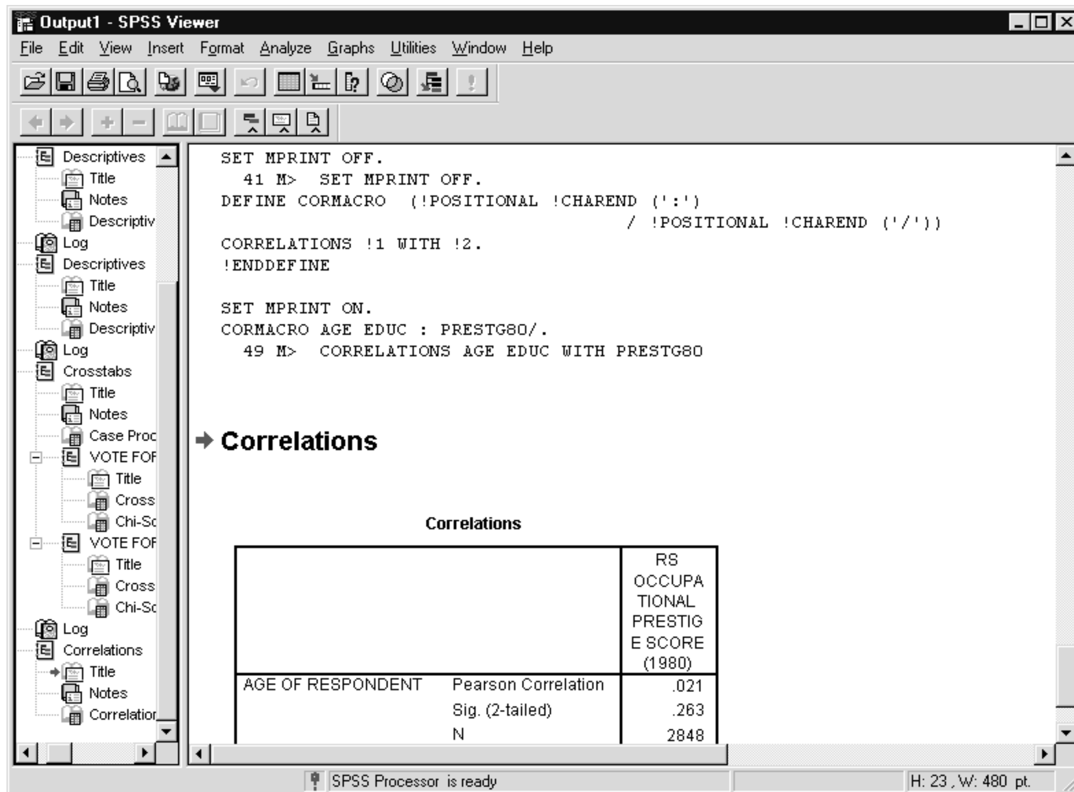
We'll run the commands through the first macro call.

Highlight the commands from **SET MPRINT OFF** to the first call of **CORMACRO**

Click on the **Run** tool button

You can see the macro expansion in the Viewer and the CORRELATIONS command created by the expansion. Below this is the correlation output, where education is reassuringly highly correlated with occupational prestige, but not age. The key point is that SPSS knew where to end each set of tokens by using ending characters. This means that with this feature any number of variables can be specified.

Figure 6.10 Output From Call of CORMACRO

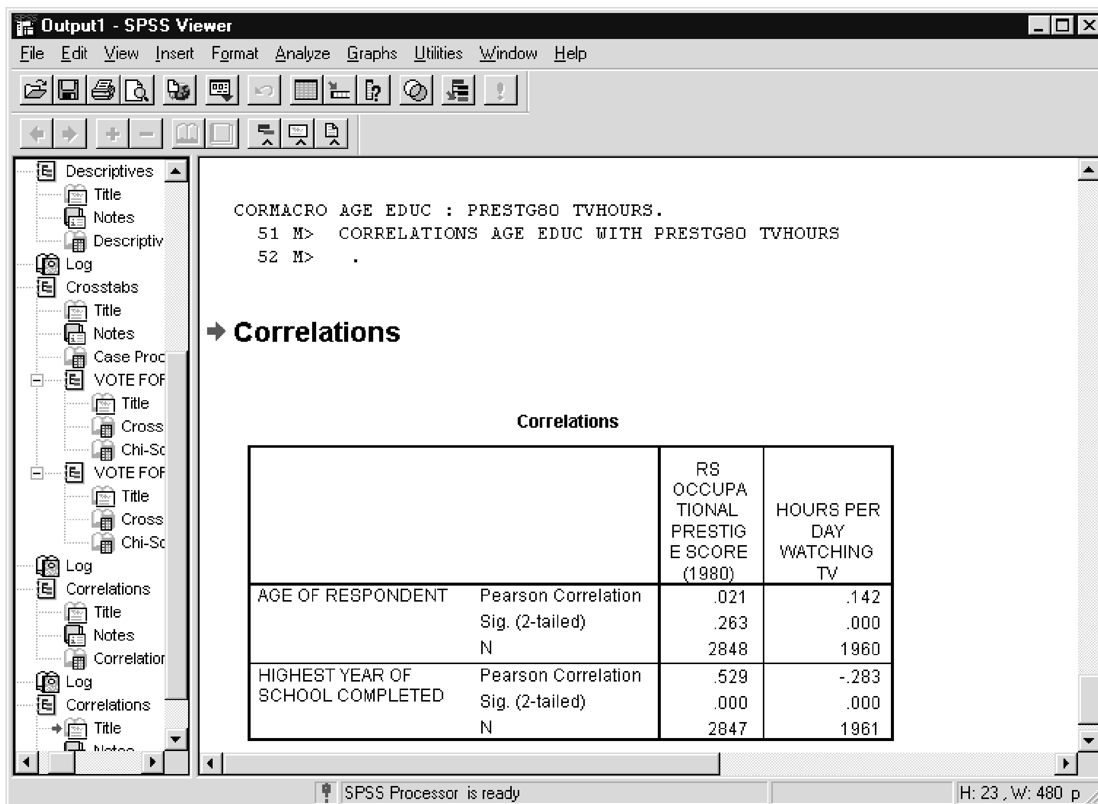


To illustrate another aspect of macro operation, run the second call of CORMACRO

Switch to the **MACRO1** Syntax (click Window..Macro1 - SPSS Syntax Editor)
Put your cursor on the **second CORMACRO** call
Click on the **Run** tool button

The macro call does not end with a slash after TVHOURS, the second variable for the second positional argument. However, the macro worked perfectly, creating the two by two correlation matrix you see in Figure 6.11.

Figure 6.11 Output From Second Call of CORMACRO



```
CORMACRO AGE EDUC : PRESTG80 TVHOURS.
  51 M>  CORRELATIONS AGE EDUC WITH PRESTG80 TVHOURS
  52 M>   .
```

Correlations

Correlations

		RS OCCUPATIONAL PRESTIGE SCORE (1980)	HOURS PER DAY WATCHING TV
AGE OF RESPONDENT	Pearson Correlation	.021	.142
	Sig. (2-tailed)	.263	.000
	N	2848	1960
HIGHEST YEAR OF SCHOOL COMPLETED	Pearson Correlation	.529	-.283
	Sig. (2-tailed)	.000	.000
	N	2847	1961

What happened is that the macro expansion placed PRESTG80 and TVHOURS after the WITH keyword on the CORRELATIONS command, but then kept reading in the macro call because no slash had been encountered. However, before a slash was encountered, SPSS read a period, which ends any command. Macro expansion stopped, and the command was passed to SPSS to be executed. Although you shouldn't always rely on a period to correctly end a macro expansion, it can work in certain instances.

The !ENCLOSE token keyword can be an alternative to !CHAREND in this instance. The next macro definition in MACRO1.SPS illustrates its operation.

Switch to the M**ACRO1** Syntax (click Window..Macro1 - SPSS Syntax Editor)
Scroll down until you see the **CORMAC2** definition

Figure 6.12 CORMAC2 Macro Definition

```
* A MACRO ILLUSTRATING THE USE OF THE !ENCLOSE TOKEN KEYWORD *.

SET MPRINT OFF.
DEFINE CORMAC2 (!POSITIONAL !ENCLOSE ( '[', ']')
                   /!POSITIONAL !ENCLOSE( '[' , ']') )
CORRELATIONS !1 WITH !2.
!ENDDEFINE.

SET MPRINT ON.
CORMAC2 [AGE EDUC TVHOURS] [PRESTG80 SPEDUC].
```

The !ENCLOSE keyword assigns all tokens between two specified characters to an argument. The starting and ending characters can be any one character string. Thus, the first positional argument, !1, will include all the tokens between the specified characters, which are the left and right brackets. The syntax may look rather daunting, but the characters have to be placed in apostrophes, separated by a comma, and the whole specification enclosed in parentheses. The second positional argument, !2, also will have an indeterminate number of tokens, also to be defined as those between left and right brackets.

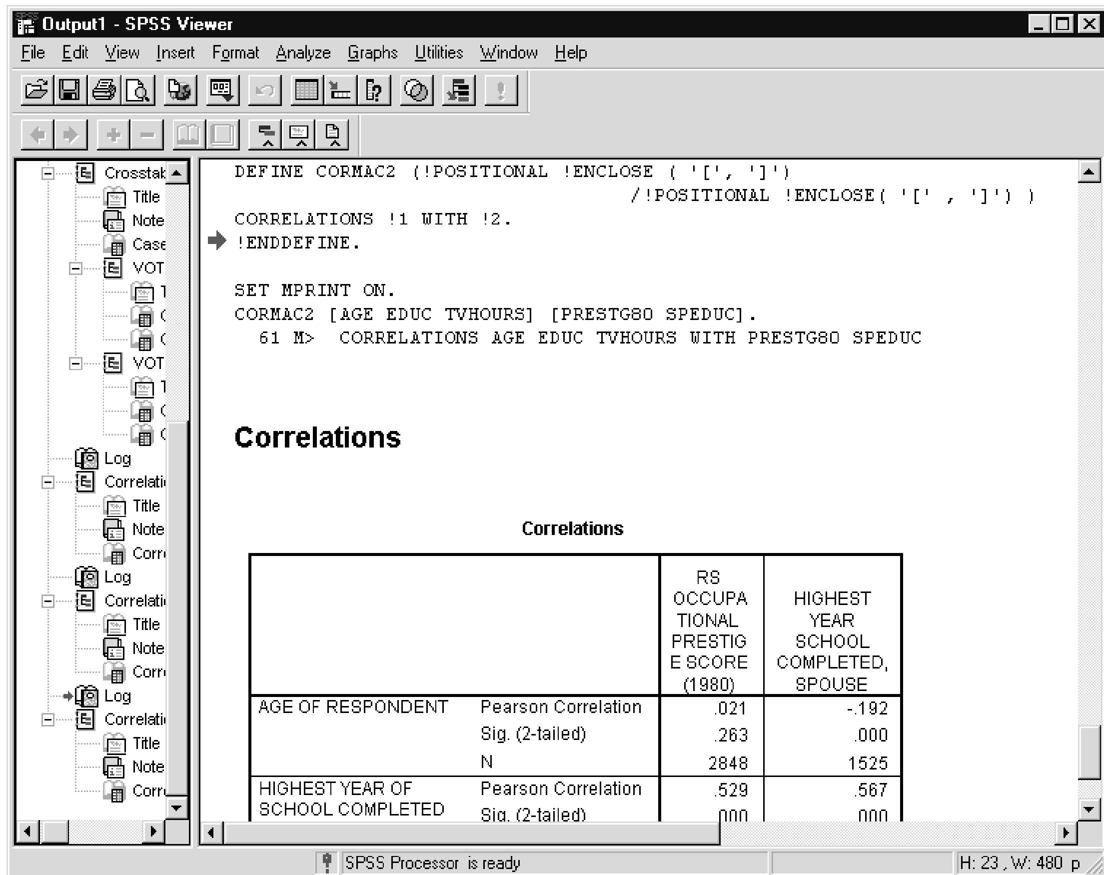
The macro body is identical to the previous macro. The macro call of CORMAC2 puts three variables between the first set of brackets. They will be placed on the left of the WITH keyword in the CORRELATIONS command. The second list of variables between brackets will be placed at the second positional argument, to the right of the WITH keyword.

Highlight the lines from **SET MPRINT OFF** to **CORMAC2**

Click on the **Run** tool button

The macro is expanded and creates the intended CORRELATIONS command. SPSS executes the command and a portion of the table is displayed in Figure 6.13.

Figure 6.13 Output From Call of CORMAC2



Correlations

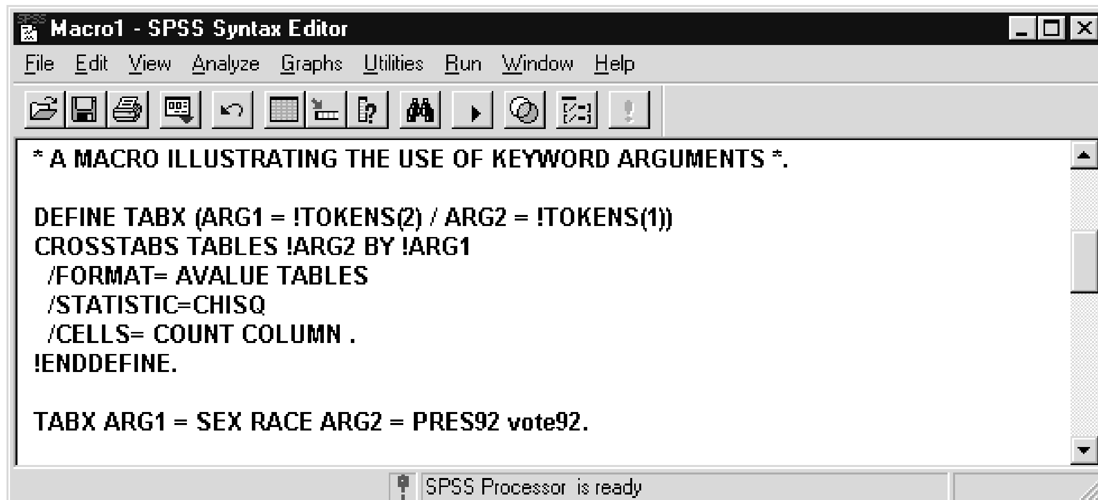
		RS OCCUPATIONAL PRESTIGE SCORE (1980)	HIGHEST YEAR SCHOOL COMPLETED, SPOUSE
AGE OF RESPONDENT	Pearson Correlation	.021	-.192
	Sig. (2-tailed)	.263	.000
	N	2848	1525
HIGHEST YEAR OF SCHOOL COMPLETED	Pearson Correlation	.529	.567
	Sig. (2-tailed)	.000	.000

WHEN THINGS GO WRONG

The key to understanding errors in macros is to understand the difference between errors in macro definition and macro calls. The macro facility simply creates strings that (you hope) are valid SPSS commands, so incorrect SPSS syntax will not prevent macro expansion. Instead, the syntax error will show up when SPSS executes the command.

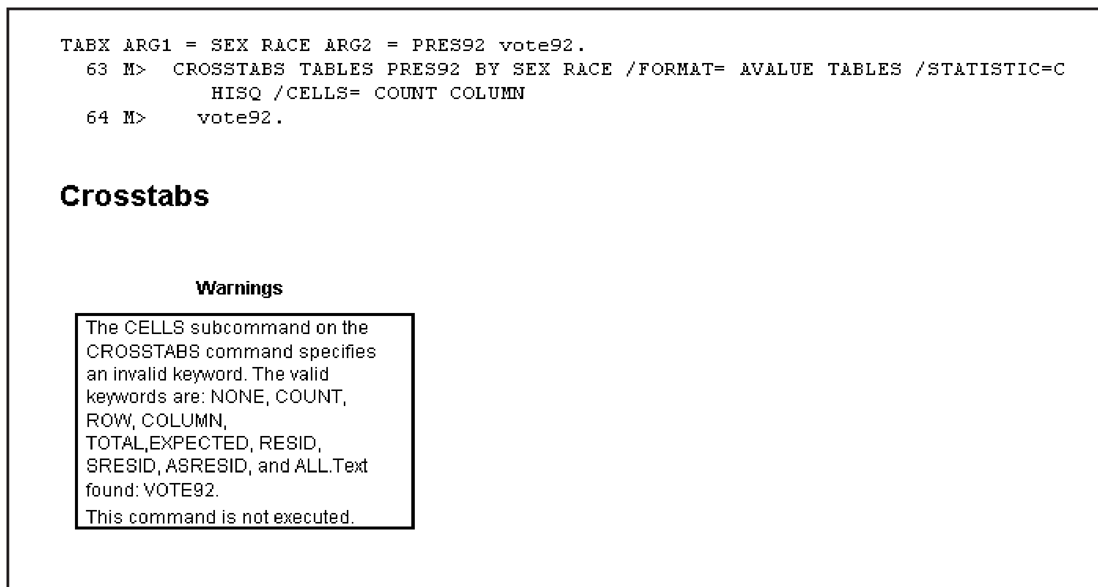
So what does happen when the user calls a macro with the wrong type of input? One possible problem is to add extra tokens to a list. In Figure 6.14 you see the TABX macro with an error on its call. Instead of one token on ARG2 we have two, PRES92 and VOTE92. If we execute this call, what will SPSS do?

Figure 6.14 Misspecified Macro Call: Extra Token



Not surprisingly, the command doesn't work, but perhaps not in the way we might have expected. SPSS expands the macro and doesn't complain upon expansion. As is evident in Figure 6.14, PRES92 is placed in the correct spot before the BY keyword, and VOTE92 is ignored while the rest of the command is built. Then, rather than throwing away this extra specification, SPSS simply adds it to the end of the command and then puts in a period. This causes a problem because it now appears that VOTE92 is a part of the CELLS subcommand, which it certainly is not. SPSS then provides a warning message when it tries to run the constructed command.

Figure 6.15 Error in Macro Call: Extra Token



And if instead we invoke this same macro with no tokens for ARG2, the result is another problem, as in Figure 6.16. The CROSSTABS

command has no variables before the BY keyword, but the macro expansion didn't complain about the missing argument. SPSS, of course, did complain when it processed the command, with an accurate warning message about the problem.

Figure 6.16 Error in Macro Call: Missing Tokens

```
TABX ARG1 = SEX RACE .
  82 M>  CROSSTABS TABLES BY SEX RACE /FORMAT= AVALUE TABLES /STATISTIC=CHISQ /C
            ELLS= COUNT COLUMN
  83 M>   .
```

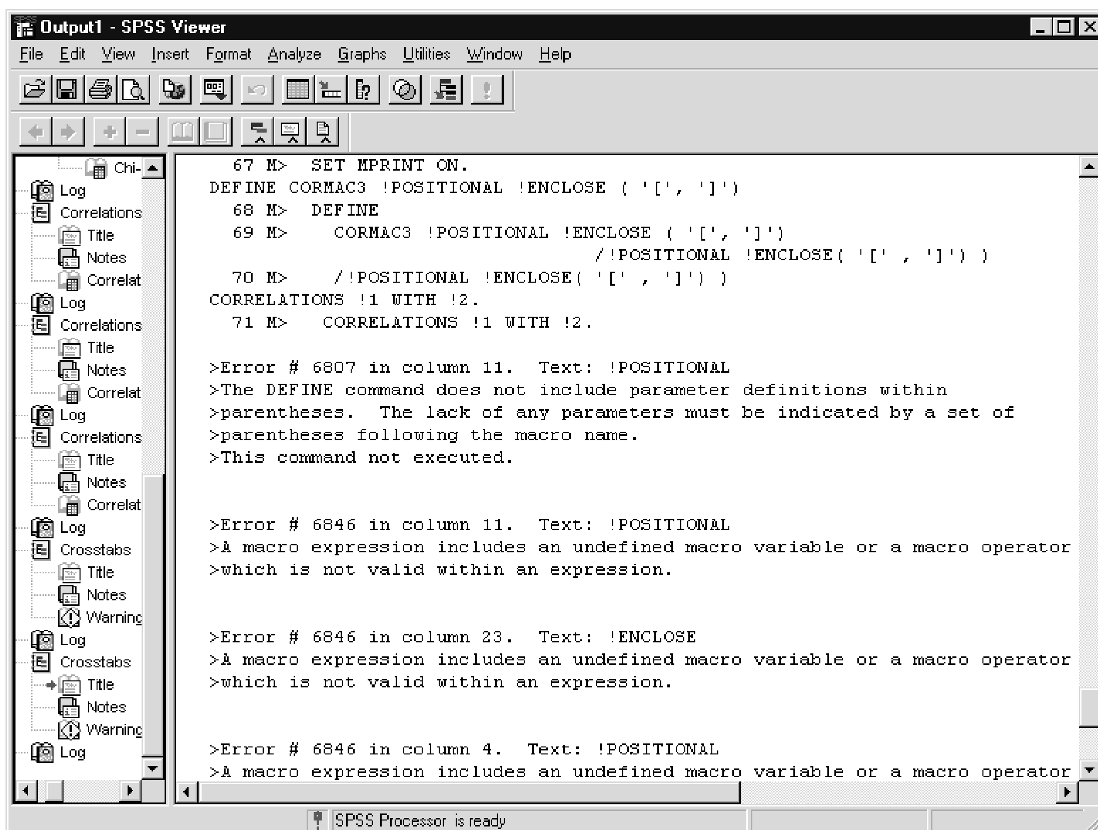
Crosstabs

Warnings

Text: BY No variables were specified where a list of variables was required. This command not executed.

What happens when a macro is defined with incorrect macro syntax? The answer is that you will receive error messages, and possibly quite a few, as the macro facility becomes confused. As you might imagine, one of the most common errors is to leave out a parenthesis. We've redefined the last correlations macro (naming it CORMAC3), but left out the first parenthesis before the first !POSITIONAL keyword. The result is the mess you see in Figure 6.17.

Figure 6.17 Error in Macro Definition



We received error 6807, which clearly states that the DEFINE command is missing definitions within a set of parentheses. However, this error causes a cascade of six other similar errors, all concerning an undefined macro variable or operator. As is usual when reading error messages, begin with the first, not the last, when looking for the problem. The CORRELATIONS command couldn't be constructed, given these errors.

SUMMARY

This chapter has reviewed the basics of macro definition and calls, focusing on arguments and tokens. The macros in this chapter were simple so that we could focus on understanding macro construction. The next chapter will introduce more complex macro features and more typical uses of macros.

Chapter 7 Advanced Macros

Topics

Introduction
Looping in Macros
Producing Several Clustered Bar Charts
Double Loops in Macros
String Manipulation Functions
Direct Assignment of Macro Variables
Conditional Processing
Creating Concatenated Stub and Banner Tables
Additional Recommendations

INTRODUCTION

The macro facility is very powerful and offers several additional features that were not discussed in the previous chapter. For example, since macros essentially manipulate and create strings, the macro facility offers several functions that manipulate strings, including some that are equivalent to the string functions available in SPSS. Macros can also do looping to accomplish repetitive tasks and conditional processing (IF like statements). Also, macros permit assignment of values to macro variables either as constants or through the evaluation of an expression.

We will review two programs that illustrate some of these features.

LOOPING IN MACROS

As we have discussed, SPSS syntax does not permit procedures to be placed inside loops. However, macros can be used to mimic this function. Since macros produce strings which are SPSS commands, the use of a loop in a macro can yield a similar, repeating set of SPSS commands. This technique can be used to advantage in many circumstances.

There are two types of loop constructs in macros: the index loop and the list processing loop. These constructs allow the user to iterate over just about anything, including variables, numbers, file names, or procedures.

The index loop is similar to the LOOP command in SPSS, looping a number of times defined by numeric arguments on the index. The syntax is:

!DO !VAR = start !TO finish [!BY step].
specifications
!DOEND.

where !VAR could be !I or a similar macro variable, and start and finish are numeric values supplied by arguments.

The list processing loop also begins with !DO and ends with !DOEND, but has this syntax

!DO !VAR !IN (list).
specifications
!DOEND.

Here, the loop iterates for as many elements as there are in the list, which is usually a macro argument (specifically, the tokens for that argument). On each iteration of the loop, the value of !VAR is set to one of the values of the list, in order.

PRODUCING SEVERAL CLUSTERED BAR CHARTS

The first macro illustrates the list processing looping construct.

Click **File..Open..Data**
Switch directories to **c:\Train\ProgSynMac** if necessary
Double click on **GSS94**

Now that we have the data file, we'll open the syntax file.

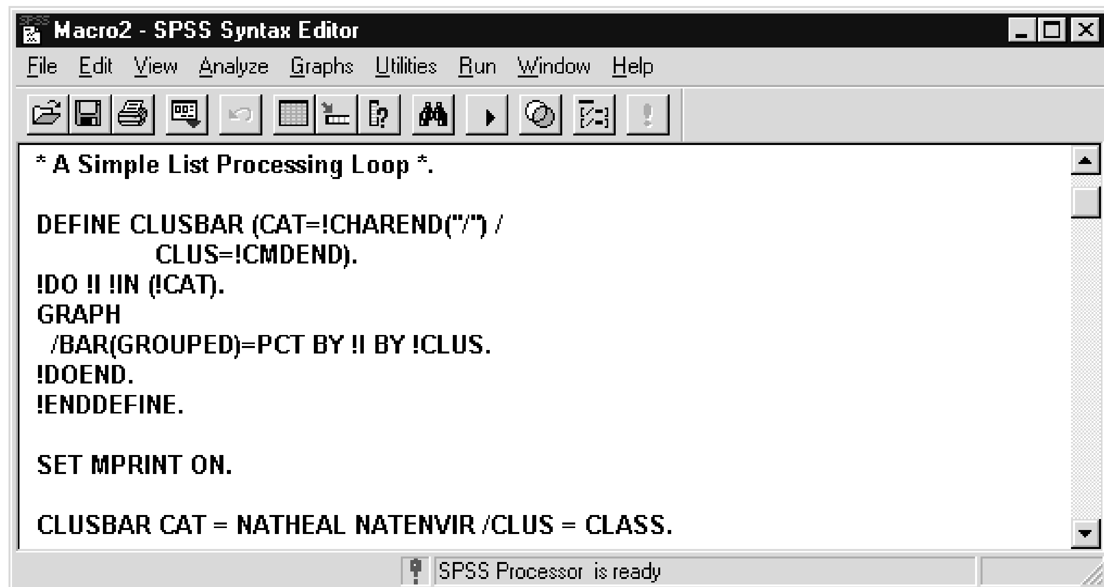
Click **File..Open..Syntax**
Double click on **MACRO2**

When creating a clustered bar chart from the menus, neither standard nor Interactive graphs permit multiple graph requests by placing several variables in the Category Axis (Horizontal or X-axis in IGraphs) or Define Clusters By (by default, Color in IGraphs) boxes. That

can be frustrating when a user wants to produce several bar charts at once. The first macro in MACRO2.SPS, displayed in Figure 7.1, solves this problem by allowing several clustered bar charts to be specified at once.

Note We illustrate using standard graphs because the basic Graph command is simpler than an IGraph command. For those using Interactive Graphs, equivalent macros using Interactive Graphs are included at the end of the Macro2.sps Syntax file.

Figure 7.1 Macro to Produce Clustered Bar Charts



The DEFINE command names the macro CLUSBAR. Two arguments are declared, CAT and CLUS, which represent the Category Axis and Define Clusters By variables. We use the !CHAREND keyword to assign the first set of tokens (variable names) up to the slash (/) to CAT, and assign the remaining tokens to CLUS with the !CMDEND keyword.

The !DO--!DOEND list processing syntax tells the macro facility to loop over the tokens in !CAT, one by one. The value of the loop on each iteration—a token from !CAT—will be placed in the macro variable !I. There will be as many iterations as there are tokens in !CAT. It's no more complicated than this.

A standard GRAPH command follows !DO to produce a clustered bar chart. However, rather than specify variable names after the two BY keywords, the value of the macro variable !I is substituted in the position of the Category Axis variable (after the first BY keyword), and !CLUS is placed in the Define Clusters By spot (after the second BY).

Only one token is named for CLUS on the macro call, so there will be one clustering variable. But on each iteration of the loop, a token from CAT will be placed in the position of !I. The macro call names two tokens (here variable names) for CAT, so two GRAPH commands will be constructed, one for each token.

The macro body ends with !DOEND first to close the loop, then !ENDDEFINE to close the definition.

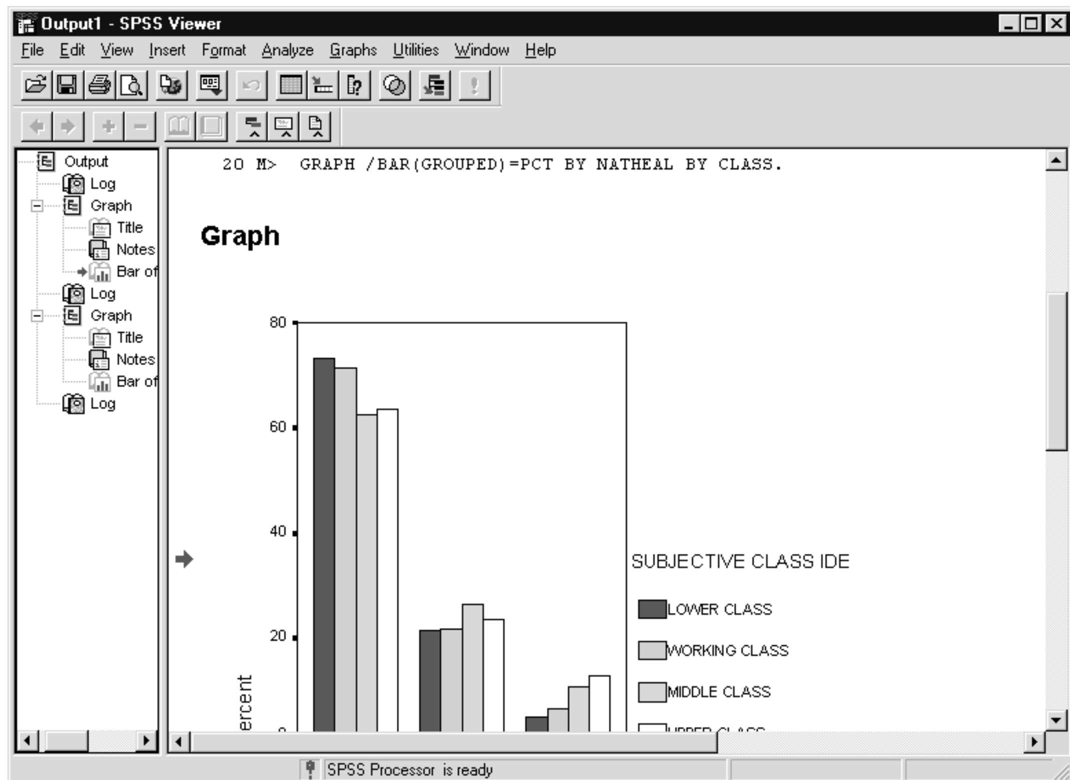
On the macro call, the variables NATHEAL and NATENVIR (asking about the level of federal spending on these issues) are read as values (tokens) of CAT, and the variable CLASS as the value of CLUS.

Highlight the lines from **DEFINE to CLUSBAR**

Click the **Run** tool button

SPSS produced two clustered bar charts, creating the command echoed in the Viewer for the first bar chart of NATHEAL and CLASS in Figure 7.2. It appears that class is related to attitude toward spending on the nation's health.

Figure 7.2 Bar Chart of NATHEAL and CLASS



If you scroll to the second bar chart (not shown) you can see that the macro call did indeed produce two graphs.

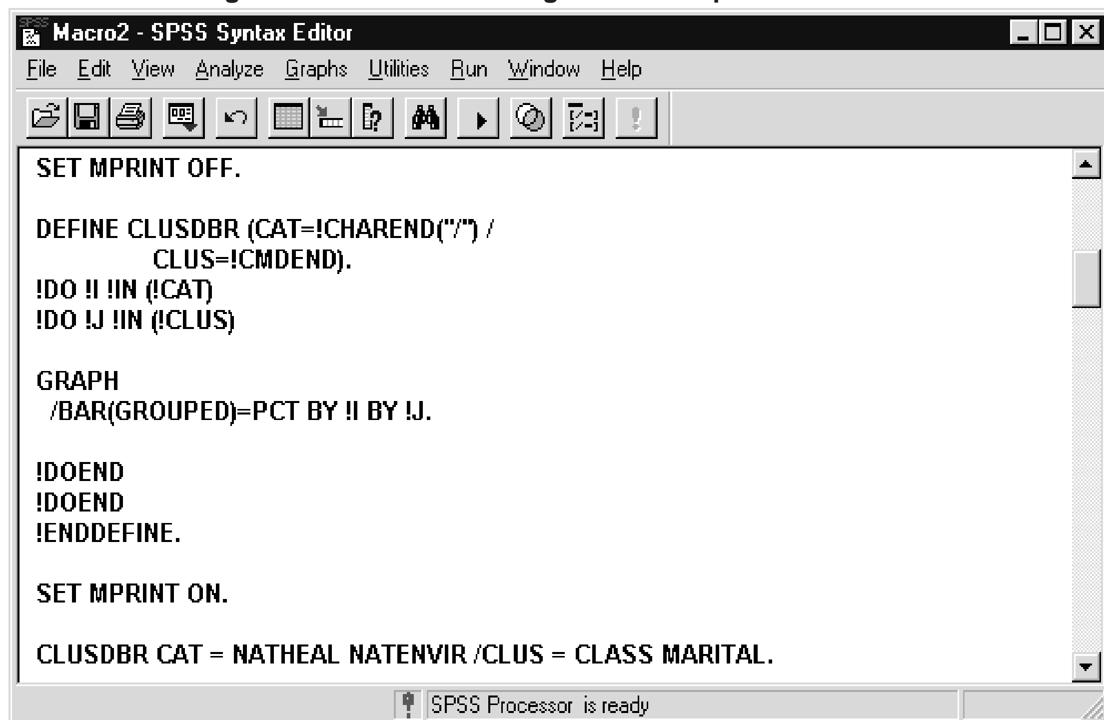
The ability to loop over variable names is very powerful and allows you to create a huge amount of output very quickly. Below we rewrite the macro to allow additional variables for the CLUS argument.

DOUBLE LOOPS IN MACROS

Our next example directly extends the macro we just examined to allow multiple variables for both the Category Axis (Horizontal or X-Axis variable for Interactive Graphs) and Define Clusters By (by default, Color variable for Interactive Graphs) variables. Again we present the example using a simple Graph command, and the equivalent macro using an IGraph command appears at the end of the Macro2.sps file.

Return to the **Macro2** Syntax window (click Window..Macro2 – SPSS Syntax Editor)
Scroll down to the **CLUSDBR** macro (the command Define CLUSDBR)

Figure 7.3 Macro Containing Double Loop



As before, the DEFINE command names the macro CLUSDBR and the CAT and CLUS arguments are declared.

Within the DO !I…!DOEND loop examined earlier, there is a second loop nested: DO !J…!DOEND. This latter syntax tells the macro facility to loop over tokens in !CLUS, one by one. The value on each iteration of this inner loop– a token from !CLUS– will be placed in the macro variable !J. There will be as many iterations of this inner loop as there are tokens in !CLUS. Also, since this loop (involving !J) is nested within an outer loop (involving !I), the inner loop will be repeated in its entirety for each token in !CAT, which drives the outer loop.

Both of the macro loop variables, !I and !J, appear in the GRAPH command. As before !I is substituted in the position of the Category Axis variable (after the first BY keyword), but now !J appears in the position of the Define Clusters By spot (after the second BY). This position was formerly occupied by the !CLUS argument itself.

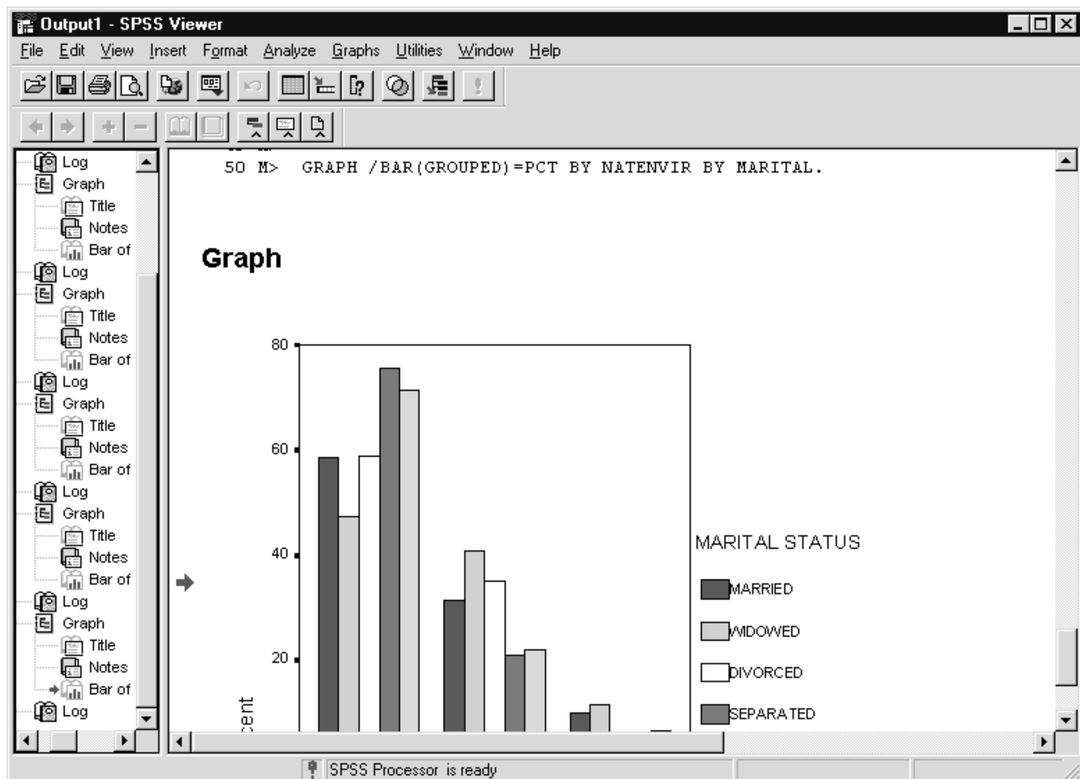
Since there are two !DO structures, there must be two !DOENDs; the inner !DOEND ends the inner (!J) loop and the outer !DOEND ends the outer (!I) loop. If we reversed the order of the two loops, that is, if !DO !J preceded !DO !I, the same set of Graph commands (and graphs) would be produced, but in a different order. This is because the inner loop iterates completely through its range or set of values for each successive value of the outer loop. The macro definition is closed with an !ENDDEFINE.

The macro call names two tokens for CAT (NATHEAL and NATENVIR) and two for CLUS (CLASS and MARITAL), so four GRAPH commands will be constructed, one for each combination of a CAT and CLUS token.

Highlight the lines from **SET MPRINT OFF** to **CLUSDBR**

Click the **Run** tool button

Figure 7.4 Bar Chart Produced from Macro with Double Loop



The Outline pane indicates, or you can discover by scrolling in the Content pane, that the macro call produced four bar charts.

Next, we turn our attention to a more complex macro, though we will first review some additional macro features.

STRING MANIPULATION FUNCTIONS

Since macro expansion creates strings, it is only natural that the macro facility supplies about a dozen string manipulation functions to create exactly the command needed. Some of the most commonly–used are these.

!CONCAT (string1, string2, ...): Concatenates several strings.
!QUOTE(string): Puts apostrophes around the string. Used to construct SPSS syntax commands that require apostrophes, such as labels and titles.
!HEAD(string): Returns the first token within a string.
!TAIL(string): Returns the end of the string after removing the first token.
!BLANKS(n): Generates a string containing the specified number of blanks. However, the blanks must be quoted because the macro facility otherwise compresses (deletes) blanks in macro expansion.

DIRECT ASSIGNMENT OF MACRO VARIABLES

In SPSS, a COMPUTE command can assign a value to a variable, like this:

```
COMPUTE  X = 5.
```

The macro facility has essentially the same capability by use of the !LET command. Its syntax is:

!LET !var = expression

where the expression may be a single token, a constant, or a macro function. So it is possible to have statements like these:

```
!LET  !X = 5
!LET  !Y =  !CONCAT(ABC,!1)
```

CONDITIONAL PROCESSING

Conditional processing in a macro is accomplished with an !IF–!IFEND construct. Unlike SPSS, macros use a !THEN operator to make an assignment. Thus the syntax is:

```
!IF (expression)  !THEN  statements
!ELSE  other statements
!IFEND
```

When the expression is true, the statements following !THEN are executed. When the expression is false and when !ELSE is specified, the statements following that command are executed.

CREATING CONCATENATED STUB AND BANNER TABLES

Using multiple categorical variables in both the rows and columns of a table is very common. The simplest possible TABLES syntax to create such a table, for a set of generic variables A through F with column percents, is this:

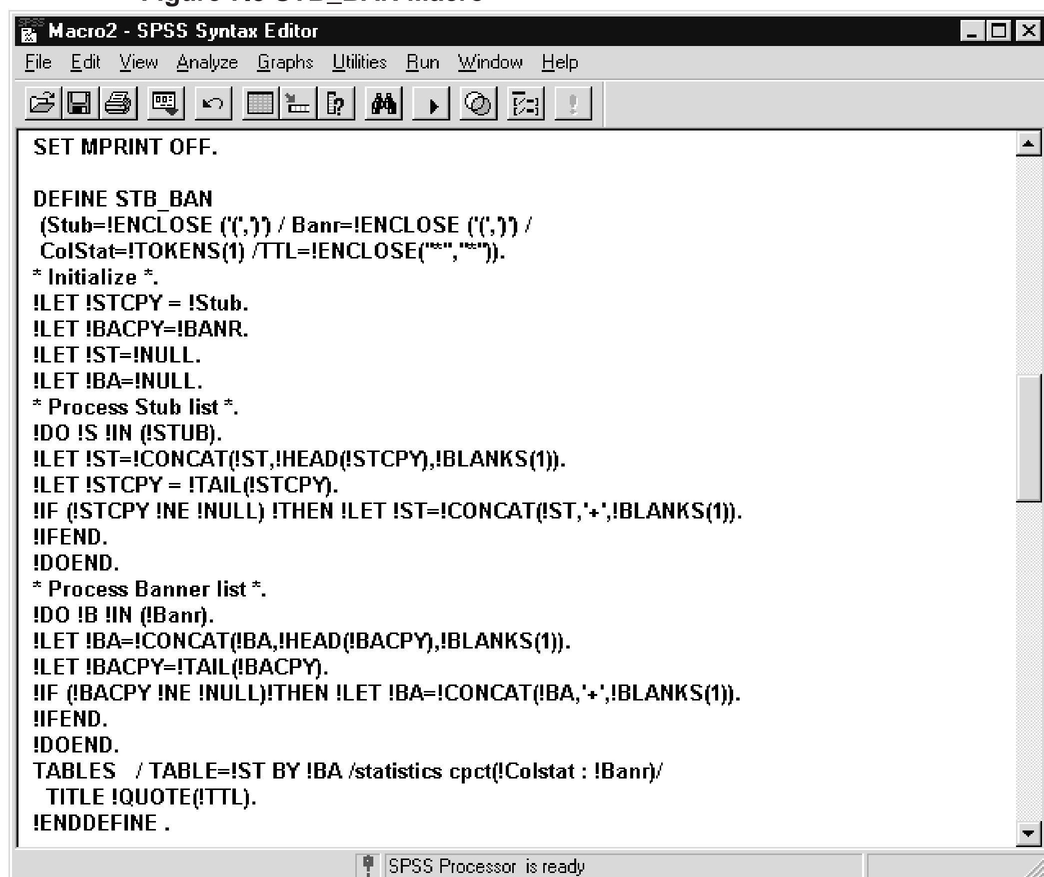
```
TABLES
  /TABLE= A + B + C  BY D + E + F
  /STATISTICS  cpct(  :D E F ).
```

If creating such tables were a recurring task, it would be handy to automate their production rather than have to use the TABLES dialog boxes for each table. The next macro carries out this job by using several of the macro features introduced above.

Return to the **Macro2** Syntax window (click Window..Macro2 – SPSS Syntax Editor)
Scroll down until you see the commands in Figure 7.5

The macro, called STB_BAN, will create a table with any number of variables on the stub and banner. It also allows the user to specify a title for the column percent statistic and an overall table title.

Figure 7.5 STB_BAN Macro



```
SET MPRINT OFF.

DEFINE STB_BAN
 (Stub=!ENCLOSE ('(',')') / Banr=!ENCLOSE ('(',')') /
 ColStat=!TOKENS(1) /TTL=!ENCLOSE("*","*")).
* Initialize *.
!LET !STCPY = !Stub.
!LET !BACPY=!BANR.
!LET !ST=!NULL.
!LET !BA=!NULL.
* Process Stub list *.
!DO !S !IN (!STUB).
!LET !ST=!CONCAT(!ST,!HEAD(!STCPY),!BLANKS(1)).
!LET !STCPY = !TAIL(!STCPY).
!IF (!STCPY !NE !NULL) !THEN !LET !ST=!CONCAT(!ST,'+',!BLANKS(1)).
!IFEND.
!DOEND.
* Process Banner list *.
!DO !B !IN (!Banr).
!LET !BA=!CONCAT(!BA,!HEAD(!BACPY),!BLANKS(1)).
!LET !BACPY=!TAIL(!BACPY).
!IF (!BACPY !NE !NULL)!THEN !LET !BA=!CONCAT(!BA,'+',!BLANKS(1)).
!IFEND.
!DOEND.
TABLES   / TABLE=!ST BY !BA /statistics cpct(!Colstat : !Banr)/
  TITLE !QUOTE(!TTL).
!ENDDEFINE .
```

The syntax of a TABLES command doesn't appear very tricky to construct, but we have to put a plus sign (+) between the variables on the stub and banner. And if the macro is to be written to allow an unlimited

number of variables, there is no way to specify ahead of time how many plus signs are needed. That is the chief problem the macro solves.

We work through this macro, line by line.

DEFINE: An argument called STUB is defined that has tokens assigned to it that are enclosed in parentheses. A second argument called BANR is defined that also has tokens assigned that are in parentheses. These two arguments will contain the variables used in the table. If you scroll down in MACRO2.SPS, you can see that the macro call (listed below) places the variables DRINK, SMOKE, and VOTE92 in STUB and SEX, RACE, and CLASS in BANR.

```
STB_BAN Stub (DRINK SMOKE VOTE92 )
     Banr ( SEX RACE CLASS) Colstat "Percent"
     TTL *Drinking, Smoking, and Voting*.
```

The DEFINE command continues by defining the argument COLSTAT and assigning it one token on the macro call. It concludes by defining the argument TTL that will have, as a token, all the text between two asterisks on the macro call. Looking at the macro call in MACRO2.SPS, we can see that TTL is used to specify the table title "Drinking, Smoking, and Voting," and COLSTAT specifies a statistics label for the column percent.

All of this is straightforward and similar to what we have seen before. The interesting part of the macro begins next. First, though, look at the TABLES command itself in the macro. It is reproduced here for reference:

```
TABLES   / TABLE=!ST BY !BA /statistics cpct(!Colstat : !Banr)/
  TITLE !QUOTE(!TTL).
```

Two arguments, !ST and !BA, which we have not defined yet, are placed in the position of the stub and banner variables. This seems odd since we have defined STUB and BANR as arguments to contain these same variables. In fact, !BANR is used after the colon in parentheses on the STATISTICS subcommand to tell SPSS to calculate column percentages. We can use !BANR in the latter position because the variables appear together with no intervening commas or plus signs, but that is not true in the TABLE subcommand.

Four !LET commands follow the DEFINE command. They set the macro variable !STCPY equal to STUB and !BACPY equal to BANR. In other words, they make copies of the variables that the user inputs. The third and fourth !LET commands set the new macro variables !ST and !BA (the ones that will actually be used in the TABLES command) to !NULL, which is an empty string of length 0.

The macro then has two similar sections, one for the stub variables and one for the banner variables.

!DO: This is a list processing loop, with the variable !S taking on successive values of the tokens in !STUB. To follow the macro through, we will use the first token in !STUB, which is DRINK. That is the first value of !S.

!LET !ST: !ST, which begins as a null string, is set equal to the concatenation of itself, plus the first token in !STCPY, which is DRINK, plus one blank. So !ST = “DRINK ”. So far, so good.

!LET !STCPY: The !TAIL function returns all tokens except the first, so !STCPY now consists of SMOKE and VOTE92.

!IF: This IF construct checks to see whether !STCPY is not equal to !NULL (the null string). In this first pass through the loop, that is true, since it contains SMOKE and VOTE92. Then the !LET command sets !ST equal to the current value of !ST, which is “DRINK ”, plus a plus sign and another blank. So now !ST will be equal to “DRINK + ”. As you can see, we are beginning to create the TABLES subcommand syntax.

Now we understand why copies were made of the original arguments; we are modifying the list of tokens in the macro on the fly.

On the second iteration, !S is set to SMOKE. !ST is set to “DRINK + SMOKE ”. !STCPY is set to the last token, VOTE92. Since !STCPY is not yet null, !ST is set to “DRINK + SMOKE + ”.

On the third and last iteration, !S is set to VOTE92. !ST is set to “DRINK + SMOKE + VOTE92”. !STCPY is set to the null string because there is only one token in !STCPY when the !TAIL function is evaluated. The !IF command returns a false result, so the !LET is no longer executed, and the !IF ends along with the loop. The correct stub syntax has been created.

Exactly the same process is used to create the syntax for the banner. Then in the TABLES command within the macro body, quotes are placed around the title, which is input as the token for TTL, using the !QUOTE function.

Interestingly, !COLSTAT is placed in the position for a statistics label but seemingly has no quotes placed around it in the macro body. However, in the macro call, note that the value of COLSTAT (“Percent”) is specified with quotes. When you specify a token on a macro call, it is not like specifying a label in a regular SPSS command. Since the token specification for COLSTAT did not use the !ENCLOSE keyword (as we did for TTL), all the text after COLSTAT on the call is read as one token, including the quote marks. This turns out to be an elegant solution to put quotes around labels in macros.

Let’s define the macro.

Highlight the lines from **SET MPRINT OFF.** to **!ENDDEFINE.**

Click the **Run tool** button

Switch to the **Viewer** window

If everything worked correctly, there will be no errors in the output.

Then let's call the macro.

Return to the Macro2 Syntax window (click **Window..Macro2 – SPSS Syntax Editor**)
Highlight the next three commands: **SET MPRINT ON.**, **STB_BAN**, and **SET MPRINT OFF.**

Click the **Run tool** button

The macro worked perfectly, placing three variables each on the stub and banner and creating the command you see in Figure 7.6. The token "Percent" was substituted for !COLSTAT, and the TITLE subcommand was created with the title "Drinking, Smoking, and Voting" placed in quotes. The !BANR argument after the colon placed the three banner variables in that spot.

Figure 7.6 Command Created From STB_BAN Call

```
TABLES / TABLE= DRINK + SMOKE + VOTE92 BY SEX + RACE + CLASS /STATISTIC
   S CPCT( "Percent" : SEX RACE CLASS )/ TITLE 'Drinking, Smoking, and
   Voting'
 .
```

Scrolling down a bit will display the table created from this command. There are interesting relationships between the three demographic variables and the three behavioral variables in the stub.

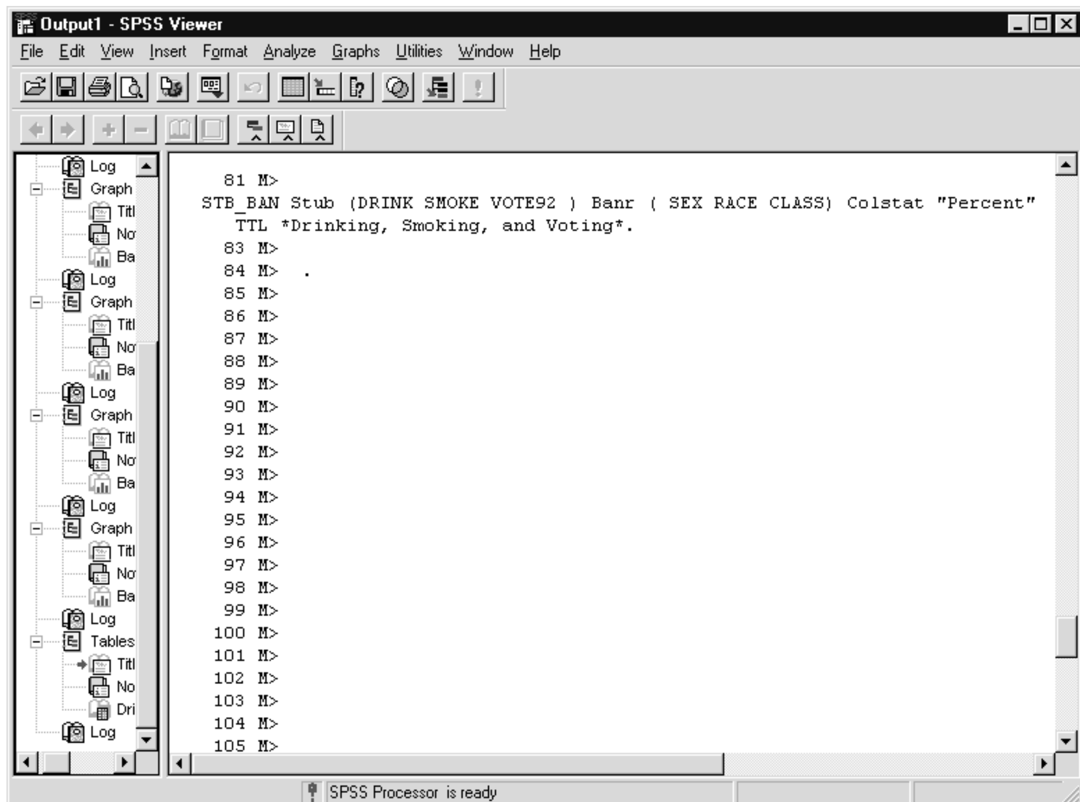
Figure 7.7 Table Created From STB_BAN Call

Drinking, Smoking, and Voting

		RESPONDENTS SEX		RACE OF RESPONDENT			SUBJECTIVE CLASS IDENTIFICATION			
		MALE	FEMALE	WHITE	BLACK	OTHER	LOWER CLASS	WORKING CLASS	MIDDLE CLASS	UPPER CLASS
		Percent	Percent	Percent	Percent	Percent	Percent	Percent	Percent	Percent
EVER DRINK ALCOHOLIC BEVERAGES?	YES	75.0%	64.3%	69.8%	63.5%	71.4%	57.1%	69.1%	70.4%	79.2%
	NO	25.0%	35.7%	30.2%	36.5%	28.6%	42.9%	30.9%	29.6%	20.8%
DOES R SMOKE	YES	27.2%	28.3%	27.6%	30.2%	23.8%	42.9%	35.9%	16.8%	16.7%
	NO	72.8%	71.7%	72.4%	69.8%	76.2%	57.1%	64.1%	83.2%	83.3%
DID R VOTE IN 1992 ELECTION	VOTED	68.5%	69.8%	70.7%	67.4%	44.9%	59.5%	63.0%	75.7%	84.4%
	DID NOT VOTE	27.4%	27.7%	26.7%	29.4%	39.8%	37.2%	34.0%	21.1%	13.5%
	NOT ELIGIBLE	3.3%	2.3%	2.2%	2.3%	14.4%	2.7%	2.5%	2.9%	2.1%
	REFUSED	.8%	.2%	.4%	.8%	.8%	.7%	.5%	.4%	

One oddity you may have noticed is the large number of lines between the macro definition in the Viewer, and the macro call and table. A portion of these lines is shown in Figure 7.8. This occurs because of all the periods ending the commands in the macro body, such as on the eight !LET commands.

Figure 7.8 Blank Lines Created From STB_BAN Call



SPSS commands normally end with a period, but some macro commands don't have that restriction, including !LET, !IF, and !DO. Thus the following syntax is perfectly correct in a macro:

```
!LET !STCPY = !Stub  !LET !BACPY=!BANR  !LET !ST=!NULL
    !LET !BA=!NULL
```

Not only are several commands placed on the same line, but also the line doesn't end with a period. You can use this feature if you wish, and it will reduce the amount of output, but usually writing macros in this fashion makes them harder to read and debug, so it may not be the best practice except for the experienced user.

ADDITIONAL RECOMMENDATIONS

To ease production of macros we suggest the following steps for your first few macros.

1) Test the SPSS commands that the macro is designed to produce and make certain they work.

2) Add a single argument at a time to the macro and thus isolate potential problems.

3) Be careful about the order of arguments and how you instruct SPSS to detect when a particular set of tokens for an argument is finished.

4) Use default values for macros when debugging.

The last bit of advice refers to the ability in macros to establish a default setting for an argument. Here is an example.

```
DEFINE MACDEF (VARLIST = !DEFAULT (SEX) !TOKENS(3) /
                    STATS= !TOKENS (2)).
FREQUENCIES VARIABLES !VARLIST /STAT = !STATS.
!ENDDEFINE.

MACDEF STATS = MEAN STDDEV.
```

In this simple macro for a FREQUENCIES command, we have defined the argument VARLIST to consist of three tokens, or variables. However, if VARLIST is not specified in the macro call, it will have a default value of "SEX" and create frequencies output only for that variable. And this is true even though three tokens would normally be input for VARLIST. Using the !DEFAULT keyword is a way to reduce the number of things being substituted and to isolate potential problems.

SUMMARY

We've discussed more advanced macro commands and functions, and we've reviewed three programs that use many of these features to accomplish common tasks.

Chapter 8 Macro Tricks

Topics

Combining Input Programs and Macros
Ordering Tables and Charts
The Case of the Disappearing Command

INTRODUCTION

In previous chapters we have introduced the basic syntax of the SPSS macro language and have demonstrated macro use with a series of examples. In this chapter, we extend the discussion by presenting several macro features that we feel may not be obvious, but have proven to be very useful in practice. These examples are based on programs written by the SPSS Consulting group for customer applications. The macros presented here have been simplified so we can easily focus on the salient points.

One of the macros uses an input program to generate a data file for testing purposes. A second pairs together crosstabulation tables and clustered bar charts based on the same variables. The final example demonstrates how macro logic can be used to include or exclude SPSS commands from a program. These serve to illustrate how macros are used in practice.

COMBINING INPUT PROGRAMS AND MACROS

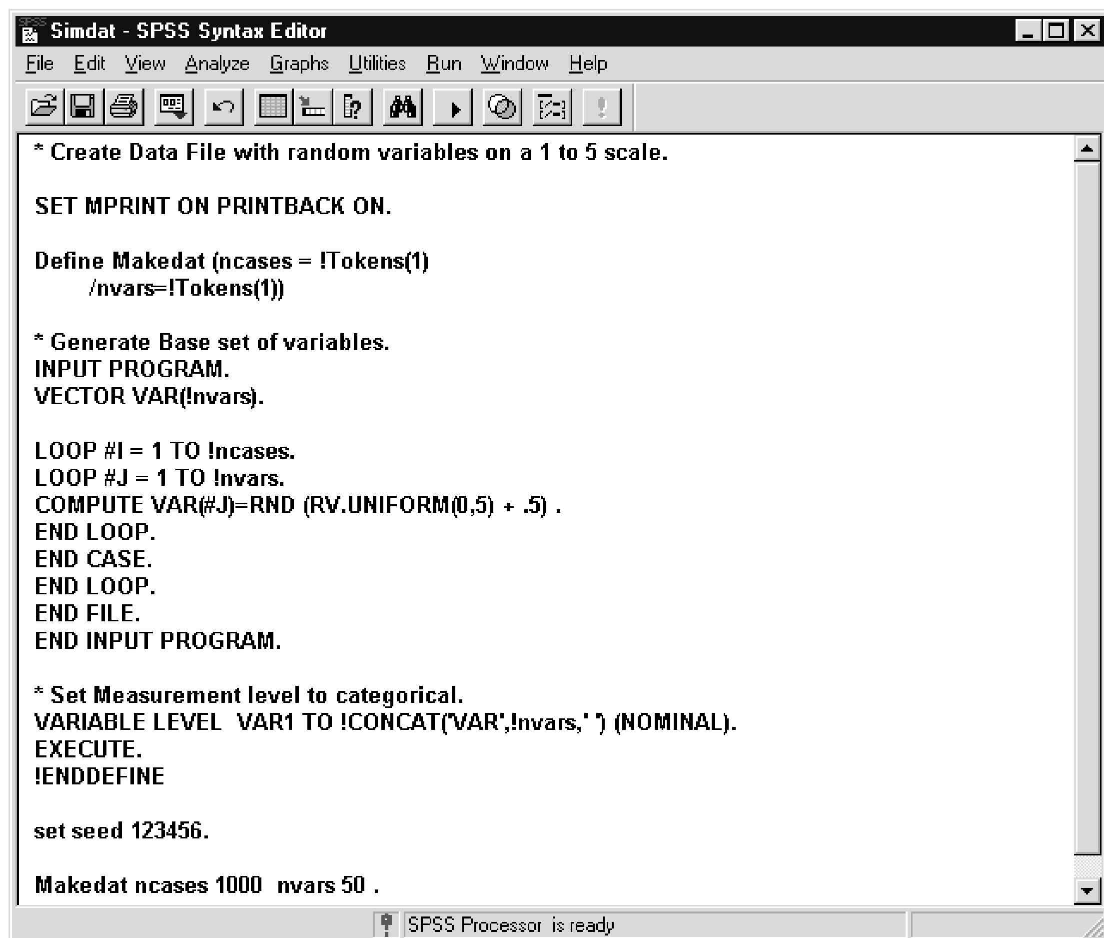
When developing custom applications, it is sometimes necessary to generate test data against which the application is run. This is done because a small data sample provided by a customer may not show the full range of variation that may be faced in practice. Also, in statistical training courses it is useful to create data files that simulate various relationships among variables.

To have SPSS create cases with no input requires, as we mentioned earlier, an input program. Placing the input program within a macro permits you produce data files with different characteristics by simply changing the macro arguments.

Since the macro will create a data file, no data need be in the Data Editor window in order to start.

Click **File..Open..Syntax** (move to c:\Train\ProgSynMac)
Double-click on **Simdat**

Figure 8.1 Macro to Produce Random Data with Values 1 to 5



The MAKEDAT macro contains two arguments, one for the number of cases (NCASES) and one for the number of variables (NVARS). For each argument a single token (TOKENS(1)) is expected, which makes sense, since only a single value need be given for the number of cases or the number of variables.

INPUT PROGRAM: If we focus first on the input program (between the INPUT PROGRAM and END INPUT PROGRAM commands), we see it is composed of two loops, one to create cases and the other to create variables within a case.

VECTOR: First a vector VAR is declared. It will contain the variables built by the macro. Since the number of variables is contained in the macro argument !NVARS, this argument is used in the VECTOR command. The size of the vector will be controlled through the !NVARS argument.

OUTER LOOP: The outer loop, indexed by #I, iterates from 1 to the number of cases specified (!NCASES). Since an END CASE command is within this loop, each iteration of the outer loop will add a case to the file. Variables are actually assigned values within the inner loop, while the outer loop controls case generation.

INNER LOOP: A secondary loop, indexed by #J, assigns random values to the variables for the current case. The loop contains a COMPUTE command that sets the #Jth variable in the VAR vector equal to a value produced by a pseudo-random number function (RV.UNIFORM). The RV.UNIFORM(0,5) function will generate a continuous value, uniformly distributed, in the range 0 through 5. By adding .5 to it, then rounding, we produce a uniformly distributed set of integer values 1 to 5. In this way, if #J (via !NVARS) were set to 20, twenty variables, VAR1 to VAR20, would each be assigned a value of 1 through 5 for the current case.

END FILE: The END FILE command is critical here. This is because no data are being input, so there is no end of file event to trigger the creation of the SPSS data file. Without the END FILE command, the program would remain paused in an input program state.

VARIABLE LEVEL: The VARIABLE LEVEL command is used to declare measurement level for SPSS variables. Since the variables are assigned integer values 1 through 5, we declare them to be nominal type. We know in advance the name of the first variable created VAR1. However, the name of the last variable is constructed by concatenating (!CONCAT) the string 'VAR' and the number of variables argument (!NVARS) and a blank space ' '. Thus if !NVARS were set to 20, the last variable would be VAR20 .

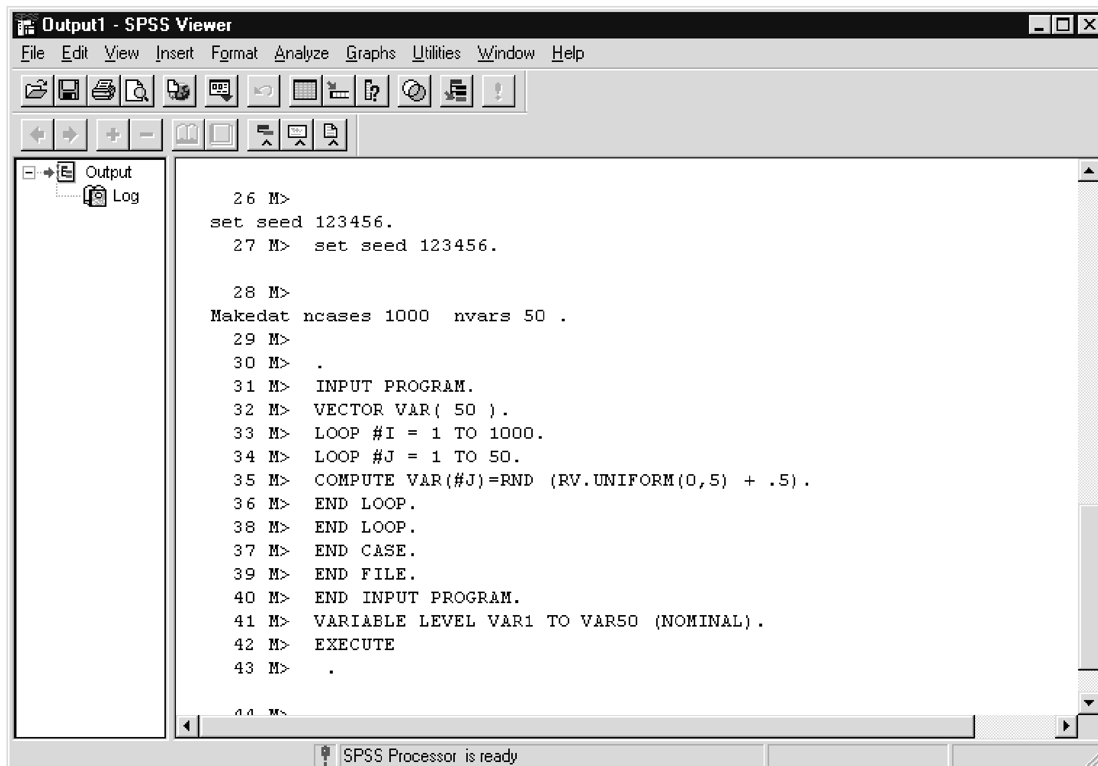
EXECUTE: Since this program contains no procedures, the EXECUTE command forces execution.

SET SEED: This is not required, but you can specify the starting point of the pseudo-random number generator by providing a large integer value for the SEED. This is mainly used to permit you to reproduce the same data values later, if desired.

When the MakeDat macro is called, it will create a data file containing 50 variables (!NVARS) and 1,000 cases (!NCASES).

> Click **Run..All**
> Switch to the **Viewer** window

Figure 8.2 Macro Generated Commands in Log



The argument value for !NCASES (1000) appears in the first LOOP (outer loop) command. Also, the argument value for !NVARS (50), has been substituted into the VECTOR, second LOOP (inner loop) and VARIABLE LEVEL commands.

> Switch to the **Data Editor** (click Goto Data tool)

Figure 8.3 Random Data Produced by Macro

Untitled - SPSS Data Editor

File Edit View Data Transform Analyze Graphs Utilities Window Help

1 : var1 4

	var1	var2	var3	var4	var5	var6	var7	va
1	4.00	2.00	4.00	2.00	2.00	1.00	1.00	
2	2.00	4.00	2.00	2.00	4.00	1.00	1.00	
3	4.00	2.00	5.00	5.00	3.00	5.00	3.00	
4	2.00	1.00	3.00	4.00	5.00	1.00	4.00	
5	4.00	3.00	2.00	5.00	5.00	3.00	1.00	
6	3.00	1.00	3.00	1.00	5.00	3.00	4.00	
7	4.00	5.00	1.00	2.00	4.00	3.00	3.00	
8	4.00	3.00	2.00	3.00	4.00	1.00	3.00	
9	4.00	2.00	2.00	5.00	5.00	3.00	5.00	
10	1.00	1.00	4.00	5.00	2.00	5.00	4.00	
11	4.00	4.00	4.00	2.00	2.00	1.00	2.00	
12	5.00	3.00	2.00	4.00	5.00	3.00	5.00	
13	2.00	4.00	4.00	5.00	5.00	4.00	3.00	

Data View Variable View

SPSS Processor is ready

We see values (1 through 5) for the first few variables (of 50) and cases (of 1,000).

Extensions

The example presented was simple in that no relationships were imposed on the created variables. You might want to create a data set that demonstrates specific relationships, for example, strong or weak correlations. Those interested in examining a more complex variation, in which the variables are first forced to be uncorrelated and then a relationship imposed, should examine the Syntax file AtSimdat.sps. It generates a data file containing a user-specified number of categorical and continuous predictor variables that are related to an outcome variable. In addition, the number of categories present in the categorical variables is passed as an argument.

ORDERING TABLES AND CHARTS

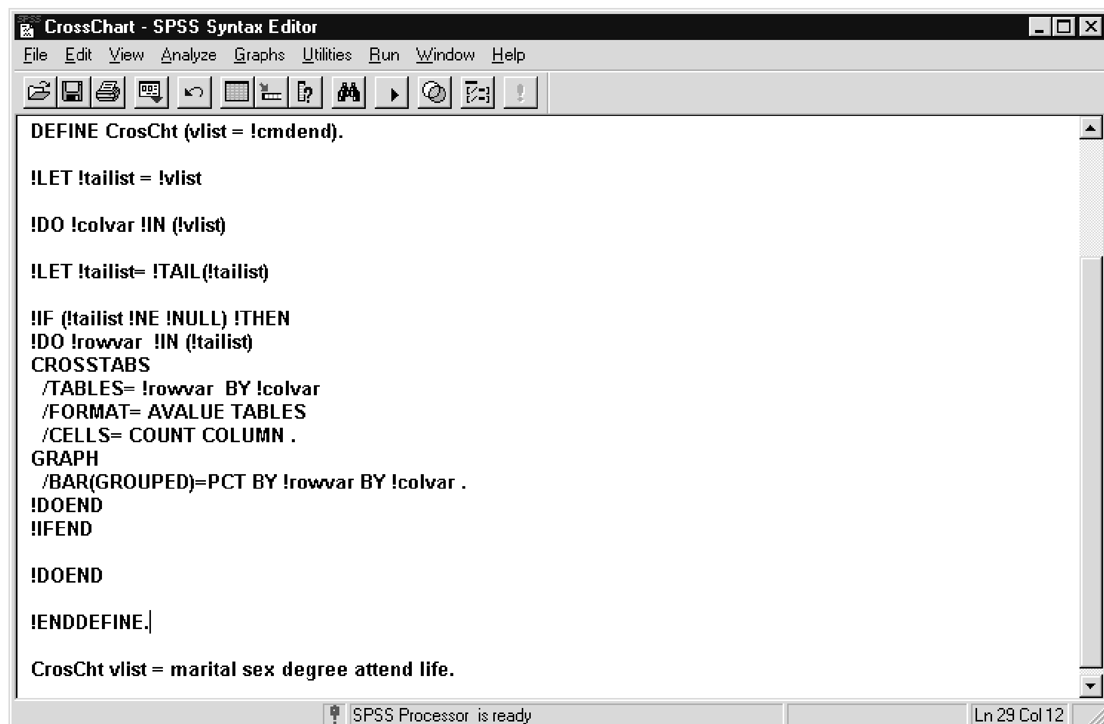
In Chapter 7 we saw a macro example in which a series of clustered bar charts were generated by placing the CHART command within macro loops. We will use a variation of this to produce output that contains, for each pair of variables on a list, a crosstabulation table and a clustered bar chart displaying the same percentages.

The challenge here is to produce just the desired tables and charts in the proper order. Specifically, if four variables were given, A, B, C and D, then we want a crosstab followed by a bar chart displaying A by B, A by C, A by D, then B by C, B by D, and finally C by D. Thus each unique pairing of the variables on the list should produce a table and chart, which should follow each other in the Viewer window.

As you, no doubt, suspect at this point, loops will be involved. Although we have seen loops within macros earlier, here the same list of variables must drive two separate loops in a coordinated fashion. In order to accomplish this, we will make use of the !TAIL macro function reviewed in Chapter 7.

> Click **File..Open..Data** (move to c:\Train\ProgSynMac)
> Double-click on **GSS94**
> Click **No** when asked to save contents of Data Editor
>
> Click **File..Open..Syntax** (move to c:\Train\ProgSynMac)
> Double-click on **CrossChart**

Figure 8.4 CrosCht Macro




```
DEFINE CrosCht (vlist = !cmdend).

!LET !tailist = !vlist

!DO !colvar !IN (!vlist)

!LET !tailist= !TAIL(!tailist)

!IF (!tailist !NE !NULL) !THEN
!DO !rowvar  !IN (!tailist)
CROSSTABS
  /TABLES= !rowvar  BY !colvar
  /FORMAT= AVALUE TABLES
  /CELLS= COUNT COLUMN .
GRAPH
  /BAR(GROUPED)=PCT BY !rowvar BY !colvar .
!DOEND
!IFEND

!DOEND

!ENDDEFINE.

CrosCht vlist = marital sex degree attend life.
```

The CrosCht macro takes a single argument (VLIST), a list of variables on which the tables and charts will be based. Below we discuss the elements of the macro.

!LET !TAILIST: The macro variable !TAILIST is set equal to the list of variables passed as an argument to the macro. It will be used later to store the tail (all tokens except the first) of the list of variables.

!DO !COLVAR !IN (!VLIST): This is the outer loop within the macro. !COLVAR is a macro variable that stores the variable name used as column variable in the crosstab table and the cluster variable in the barchart. At each iteration of the loop the next variable from !VLIST is assigned to !COLVAR.

!LET !TAILIST = !TAIL (!TAILIST): Here !TAILIST is set equal to the tail of itself. Since !TAILIST was initialized to !VLIST, in the first iteration of the loop it contains all variables passed to the macro except the first. We later pair each variable name in !TAILIST with the variable name stored in !COLVAR to produce tables and charts. For example, if the !VLIST argument contained variables A, B, C and D, during the first outer loop iteration, !COLVAR = 'A' and !TAILIST ='B C D". During the second iteration, !COLVAR = 'B" and !TAILIST = 'C D'. This allows us to match the head variable on the list with the remaining variables.

!IF..!IFEND: At the last iteration of the outer loop, !TAILIST will be empty and no tables or charts should be attempted. The !IF statement will only generate the CROSSTAB and GRAPH commands if !TAILIST is not null.

!DO !ROWVAR !IN (!TAILIST): The inner loop will assign, during each iteration, a value from !TAILIST to !ROWVAR. The variable name stored in !ROWVAR will be paired with the variable name in !COLVAR to create the table and chart.

CROSSTABS and GRAPH: The macro variables !ROWVAR and !COLVAR are substituted as the row (category) and column (cluster) variables in the CROSSTAB (GRAPH) command. Since the CROSSTABS and GRAPH commands are within the loop, they will be run in sequence for each combination of !ROWVAR and !COLVAR variable names. This insures that the table and graph for a pair of variables will appear contiguously in the Viewer window.

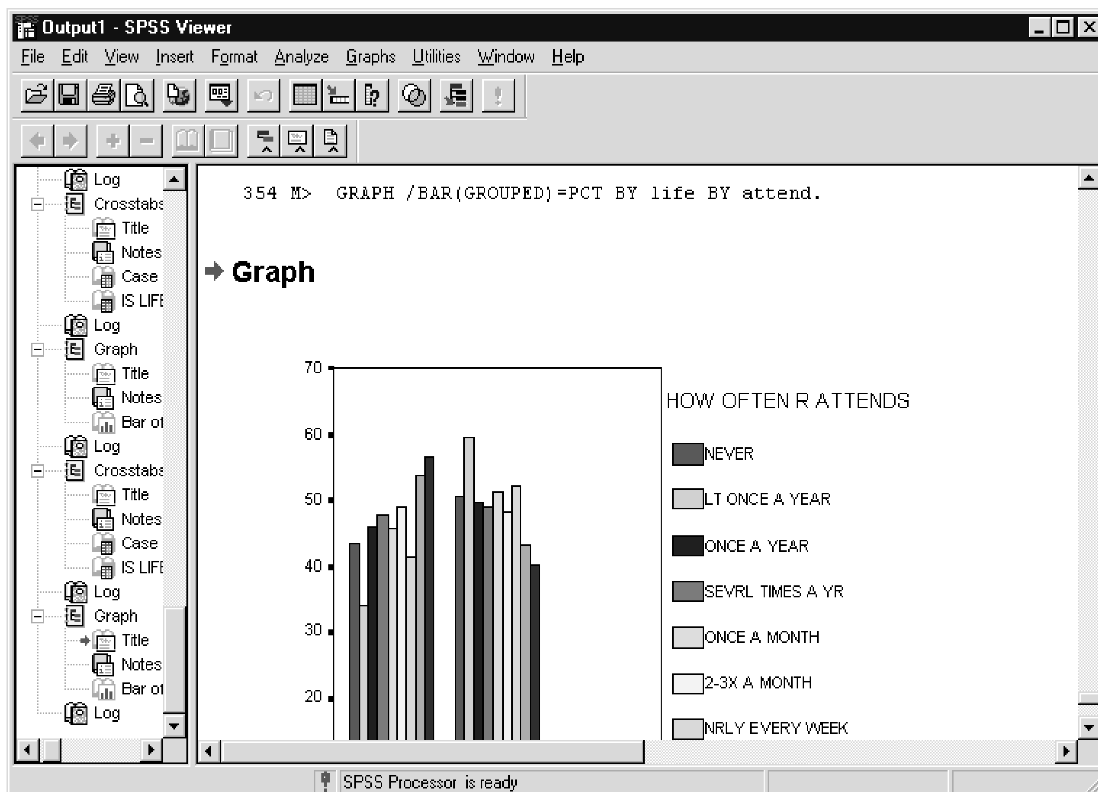
If the variables A, B, C, D were passed to the macro, !COLVAR and !TAILIST would take the following values at each iteration of the outer loop.

Iteration	*!COLVAR*	*!TAILIST*
1	A	B C D
2	B	C D
3	C	D
4	D	null

In this way, at each outer loop iteration the first variable name on the list of remaining variables (!TAILIST) is dropped from this list and becomes the !COLVAR.

Click **Run..All**

Figure 8.5 Crosstab Tables and Graphs Produced by CrosCht Macro



The Outline pane displays the sequence of a crosstab followed by a graph for the last few variable combinations. The creative use of macro string functions, such as !HEAD and !TAIL, permit macros to process argument lists in flexible ways.

THE CASE OF THE DISAPPEARING COMMAND

So far we have used macros to build commands. However, macro logic can be used to build or not build certain commands based on arguments passed to the macro. This can be useful if you wish data selection or modification done only when requested. We illustrate this in a macro that runs a simple human resources report for employee data. If an age argument is passed to the macro, then a data selection command (SELECT IF) will be run based on the age value. If no age argument is passed to the macro, then no data selection command is built. In addition, if age selection is performed, then a note is added to the title of the summary report (CASE SUMMARIES procedure).

> Click **File..Open..Data** (move to c:\Train\ProgSynMac)
> Double-click on **BankNew**

Figure 8.6 Employee Data

BankNew - SPSS Data Editor

File Edit View Data Transform Analyze Graphs Utilities Window Help

1 : id 1

	id	g	bdate	educ	jobcat	salary	salbegin	jobtime	prevexp	m
13	13	m	07/17/60	15	1	$27,750	$14,250	98	34	
14	14	f	02/26/49	15	1	$35,100	$16,800	98	137	
15	15	m	08/29/62	12	1	$27,300	$13,500	97	66	
16	16	m	11/17/64	12	1	$40,800	$15,000	97	24	
17	17	m	07/18/62	15	1	$46,000	$14,250	97	48	
18	18	m	03/20/56	16	3	$103,750	$27,510	97	70	
19	19	m	08/19/62	12	1	$42,300	$14,250	97	103	
20	20	f	01/23/40	12	1	$26,250	$11,550	97	48	
21	21	f	02/19/63	16	1	$38,850	$15,000	97	17	
22	22	m	09/24/40	12	1	$21,750	$12,750	97	315	
23	23	f	03/15/65	15	1	$24,000	$11,100	97	75	
24	24	f	03/27/33	12	1	$16,950	$9,000	97	124	
25	25	f	07/01/42	15	1	$21,150	$9,000	97	171	

Data View / Variable View

SPSS Processor is ready

The data file contains employee information. The macro will generate a report containing summaries of education, current salary and time in current job position for subgroups. Age is included in the file, but is not visible in Figure 8.6.

> Click **File..Open..Syntax** (move to c:\Train\ProgSynMac)
> Double-click on **MacroSelect**

Figure 8.7 MacroSelect Syntax File

MacroSelect - SPSS Syntax Editor

File Edit View Analyze Graphs Utilities Run Window Help

```
set mprint off printback on.
DEFINE REPORT1 (AGE=!DEFAULT(ALL) !TOKENS(1)
                /GROUP=!DEFAULT(JOBCAT) !TOKENS(1))


* Build a Select If command if Age value specified.
!IF (!AGE !NE 'ALL')  !THEN
TEMPORARY.
!CONCAT('SELECT IF (AGE > ',!AGE,').')
!LET !TITLE =!CONCAT('Human Resources Report:   Age > ', !AGE)
!ELSE
!LET !TITLE='Human Resources Report'
!IFEND


SUMMARIZE
  /TABLES=educ salary jobtime  BY !GROUP
  /FORMAT=NOLIST TOTAL
  /TITLE=!QUOTE(!TITLE)
  /MISSING=VARIABLE
  /CELLS=MEAN MEDIAN COUNT .

!ENDDEFINE.

set mprint on printback on.

REPORT1 AGE 50.
REPORT1 GROUP GENDER.
REPORT1 AGE 35 GROUP educ.
```

SPSS Processor is ready

We comment on several aspects of the macro, but focus on the instructions that build the data selection command.

REPORT1: The REPORT1 macro takes two arguments: AGE and GROUP. The AGE argument has a default value of 'ALL', but if an age value is supplied it will be used in a SELECT IF command. The GROUP argument has a default value of JOBCAT (the job category variable name), and the GROUP value (variable name) will be used to create subgroups within the report. Since only one value can be supplied for age or group (this could be generalized to allow for an age range, or a list of group variables), each is declared as a single token argument.

!IF..!IFEND: The !IF statement tests the value of the !AGE argument. If !AGE is not equal to 'ALL", then an age value has been passed in the macro call. In this case, a TEMPORARY command is added (so the SELECT IF command will only be in effect for the report). Also, a SELECT IF command is created and the title string includes

age information. If !AGE equals 'ALL', then no TEMPORARY and SELECT IF commands are created, and a standard title string is used.

TEMPORARY: The TEMPORARY command will cause any transformation statements between it and the next procedure to be temporary changes. That is, any data selection or modifications will apply to the next procedure, but then disappear. It is used because we do not want the age selection to be permanent, but rather apply only to the report.

!CONCAT: The !CONCAT statement builds a SELECT IF command by concatenating three strings. The first part is the beginning of a SELECT IF command based on age ('SELECT IF (AGE >'). Next comes the age value passed as a macro argument (!AGE), followed by the end of the command (').'). If AGE 30 were specified on the macro call, the !CONCAT statement would build the following command: 'SELECT IF (AGE > 30).'. This is a simple command and more complex variations, based on lists of values or names, can be built using the type of logic we reviewed in the Table example found in Macro2.sps (Chapter 7).

!LET !TITLE: A macro variable storing the title to be used in the report is set equal to a string that includes age cutoff information.

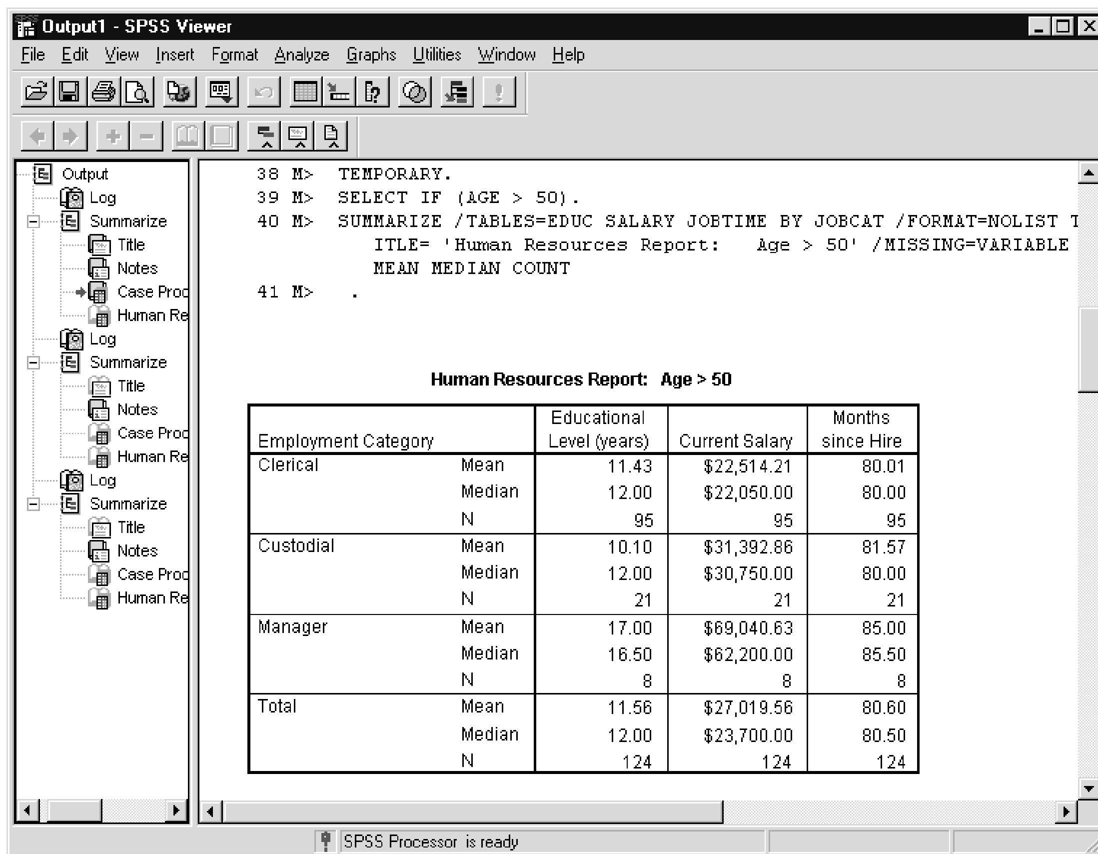
!ELSE..!IFEND: If no age argument was passed, then no TEMPORARY and SELECT IF commands are created, and the !TITLE string makes no mention of an age cutoff.

SUMMARIZE: The Case Summaries procedure (SUMMARIZE) will report on education (EDUC), current salary (SALARY) and time in current job position (JOBTIME) broken into groups based on the !GROUP value. Recall that, by default, !GROUP will be equal to JOBCAT (job category) and, if a !GROUP argument is used in the macro call, then !GROUP will contain the variable name supplied. Also, the !TITLE created earlier appears in the TITLE subcommand. Note that the !QUOTE function is applied to !TITLE. Titles must be enclosed in quotes, but if explicit quote marks ('!TITLE') were used , the report title would be "!TITLE" and not "Human Resources Report..".

To demonstrate, the macro is called three times. The first instance, "REPORT1 AGE 50.", supplies an age cutoff, but no group variable. The second, "REPORT1 GROUP GENDER." has no age cutoff, but requests that GENDER be used as the grouping variable in the report. Both arguments are used in the last example "REPORT1 AGE 35 GROUP EDUC.". Let's examine the results.

Click **Run..All**
Scroll to the **first Summarize pivot table**

Figure 8.8 Report with Age Selection



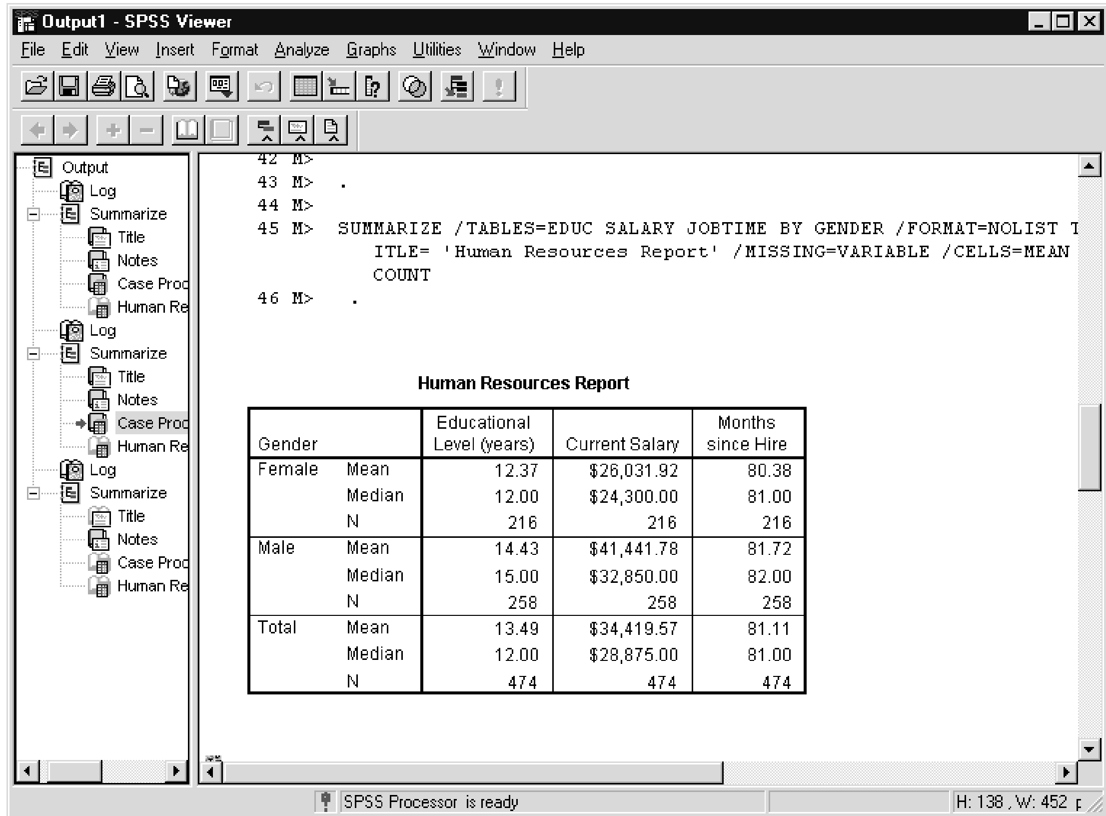
Employment Category		Educational Level (years)	Current Salary	Months since Hire
Clerical	Mean	11.43	$22,514.21	80.01
	Median	12.00	$22,050.00	80.00
	N	95	95	95
Custodial	Mean	10.10	$31,392.86	81.57
	Median	12.00	$30,750.00	80.00
	N	21	21	21
Manager	Mean	17.00	$69,040.63	85.00
	Median	16.50	$62,200.00	85.50
	N	8	8	8
Total	Mean	11.56	$27,019.56	80.60
	Median	12.00	$23,700.00	80.50
	N	124	124	124

Note that the Case Summaries pivot table and Title items are hidden (double-click on item in Outline pane to hide) in these screen shots.

The TEMPORARY and SELECT IF commands appear in the Log and we see that the report title reflects the age selection. The Case Summaries report is grouped by job category (recall JOBCAT is the default value for !GROUP). Notice the total sample size (N summary for Total) value of 124 employees.

Scroll to the **second Summarize pivot table**

Figure 8.9 Report with Gender Groups and No Age Selection



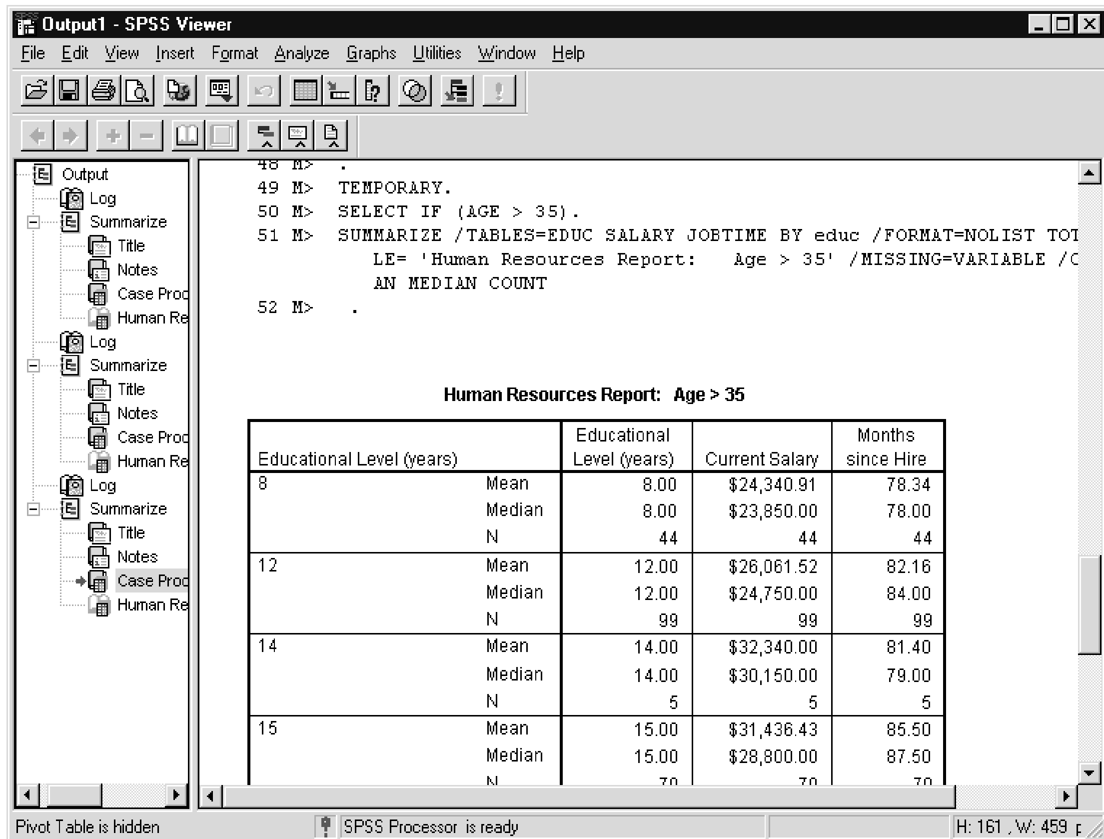
Human Resources Report

Gender		Educational Level (years)	Current Salary	Months since Hire
Female	Mean	12.37	$26,031.92	80.38
	Median	12.00	$24,300.00	81.00
	N	216	216	216
Male	Mean	14.43	$41,441.78	81.72
	Median	15.00	$32,850.00	82.00
	N	258	258	258
Total	Mean	13.49	$34,419.57	81.11
	Median	12.00	$28,875.00	81.00
	N	474	474	474

Since AGE was not used as an argument, neither a TEMPORARY nor a SELECT IF command appears in the Log. The pivot table title makes no reference to age. Also, note that the total sample size is 474, which is considerably larger than in the previous report, because no data selection is being done.

Scroll to the **third Summarize pivot table**

Figure 8.10 Report with Age Selection and Education Groups



Human Resources Report: Age > 35

Educational Level (years)		Educational Level (years)	Current Salary	Months since Hire
8	Mean	8.00	$24,340.91	78.34
	Median	8.00	$23,850.00	78.00
	N	44	44	44
12	Mean	12.00	$26,061.52	82.16
	Median	12.00	$24,750.00	84.00
	N	99	99	99
14	Mean	14.00	$32,340.00	81.40
	Median	14.00	$30,150.00	79.00
	N	5	5	5
15	Mean	15.00	$31,436.43	85.50
	Median	15.00	$28,800.00	87.50
	N	70	70	70

An age selection is performed and is reflected in the title of the report. The subgroup summaries are based on years of education (EDUC) since EDUC was passed as the GROUP argument.

Variations on this example can provide a flexible reporting system, especially when multiple pivot tables must be produced within an overall report.

SUMMARY

In this chapter we reviewed several examples (input programs within macros, single argument lists driving multiple loops, optionally generating data selection commands) that combine macro features to produce more flexible and powerful macros.

Exercises

All exercise files for this class are located in the c:\Train\ProgSynMac folder on your training machine. If you are not working in an SPSS Training center, the training files can be copied from the floppy disk that accompanies this course guide. If you are running SPSS Server (click File..Switch Server to check), then you should copy these files to the server or a machine that can be accessed (mapped from) the computer running SPSS Server.

Solutions to the major exercises (not every variation is included) can be found in SelectedSolutions.sps.

Chapter 3

Complex File Types

Use the ASCII text file ProdNest.dat for this exercise.

As part of a household study, household members were asked to rate a new product on three measures. Since most households have multiple members, the data was structured in the following manner. First there is a household record that contains household ID, number of members and income category. Next there are as many records as members of the household, each contains household ID, age, sex and three rating variables. The goal is to read this as a nested file, so household information (income) can be related to the individual ratings.

The record layout appears below:

Household record: Type 'H'

Field	***Columns***
Record Type	1
Household ID	2-5
Income group	8
# Members	11-13

Member record: Type 'M'

Field	***Columns***
Record Type	1
Household ID	2-5
Age	6-8
Sex	13
Rating 1	18
Rating 2	23
Rating 3	28

a) Read the data in SPSS as a nested file type.

b) Obtain means for age and the rating variables broken down by income group.

c) Examine the correlations between age and the rating variables.

Chapter 4

Input Programs

Use the ASCII text file ProdRatings.dat for this exercise.

Customers were asked to name the three products (from a list of 10 within a product category) they most often used, and then to rate each of the three products on five rating scale questions. Thus the data file contains (in addition to ID and SEX), eighteen variables (three sets of product plus five ratings). The task is to build a table of means with products in the rows and the rating scale questions across the columns. Since each customer picked three of the ten products in any order, a product could be in any of the three sets of variables. Thus the required table cannot be produced from the original data structure (one record per customer).

However, the table can be run if each product a customer rates is on a separate case. Your task is to accomplish this using the REPEATING DATA command.

The file (ProgRatings.dat) has the following format:

Variable	***Position***
ID	1-5
SEX	6-10
Product1	11-15
P1Rating1	16-20
P1Rating2	21-25
P1Rating3	26-30
P1Rating4	31-35
P1Rating5	36-40
Product2	41-45
P2Rating1	46-50
P2Rating2	51-55
P2Rating3	56-60
P2Rating4	61-65
P2Rating5	66-70
Product3	71-75
P3Rating1	76-80
P3Rating2	81-85
P3Rating3	86-90
P3Rating4	91-95
P3Rating5	96-100

a) Read the data so that there is one case per product rated by a customer, with variables ID, SEX, PRODUCT, and RATE1 to RATE5.

b) Use the Means procedure (Analyze..Compare Means..Means) to create a table of means of the rating scale variables broken down by product.

Chapter 5 Advanced Data Manipulation

SPSS data files, TranTransact.sav and ProdRatings.save are used in this exercise.

a) Open the TrainTransact.sav SPSS data file. Open the Syntax file Rating.sps and execute the commands, which will compute a RATING variable with values 1 to 100. Open the TransactionAgg.sps Syntax file. Modify the program so that if a customer takes a course, the rating is stored under the course name variable in the aggregated file. If a course was not taken by the customer, course variable's value in the aggregated file should be system missing.

This exercise involves the same data structure as the exercise in Chapter 4, except we begin with an SPSS data file. See the data description in the Chapter 4 exercises.

a) The SPSS data file, ProdRatings.sav, contains the following variables ID, SEX, P1, P1R1, P1R2, P1R3, P1R4, P1R5, P2, P2R1…P2R5, P3, P3R1…P3R5. The task is to create a new file with one case per product rated by a customer.

b) Use the Means procedure (Analyze..Compare Means..Means) to produce a table of means for the rating scale variables broken down by product. Use the Crosstabs procedure to examine the two-way table of product by sex.

Chapter 6 Introduction to Macros

Use the GSS94.sav SPSS data file for this exercise.

a) Use the SPSS menus to run a Descriptives analysis (Analyze..Descriptive Statistics..Descriptives) for a single variable. Return to the dialog and use the Paste pushbutton to obtain the Syntax for the Descriptives command.

b) Build a macro named TestDes that accepts a single argument (one variable name) and runs the DESCRIPTIVES command. Build several variations of the macro: one using a positional argument and one using a keyword argument. Experiment with additional argument variations: first have the argument accept a single token, then have it accept an argument enclosed within colons (i.e. : age :), then have it accept an argument ending in a colon (:), finally have it accept the end of command as a terminator. Test each variation with a single variable name and a list of several variables (note that the single token variation should fail when a list of variables is used in the macro call).

c) Modify the final version of the macro created in the previous step, so it accepts a second argument, named STATS, which will be a list of the statistics to be displayed in the Descriptives output.

d) *For those with extra time*: Test the TestDes macro with the arguments in different orders. If the macro fails due to argument order, modify the macro so that argument order does not matter.

Chapter 7 Advanced Macros

Use the GSS94.sav SPSS data file for this exercise.

a) Write a macro that takes one argument: a variable list. It should run a Frequencies command, using all the variables appearing on the variable list.

b) Modify the macro to accept a second argument: a chart option. If the Chart option is set to YES, then the Frequencies command should also produce bar charts (using the /BARCHART subcommand). Set the default value of the chart option to NO.

c) Test to see if your macro will work with upper and lower case letters (YES, Yes, yes). If not, use the !UPCASE function so it works in regardless of case.

d) Modify the macro so that instead of running a single Frequencies command for all variables passed through the argument, it runs a Frequencies command for each variable, followed by an standard pie chart. Save the Syntax file as FreqMac.sps.

Chapter 8 Macro Tricks

Use the BankNew.sav SPSS data file for this exercise.

a) Open the MacroSelect.sps Syntax file used in the chapter. Modify the macro so that in place of the age argument it takes a YEAR (4-digit) argument. If this argument is specified, only those born after the specified year will be included in the report (hint: use the XDATE.YEAR SPSS function to obtain the year from the bdate variable).

b) Modify this macro so the YEAR argument will take two values, a beginning year and an ending year. If this argument is used, only those born within the range of years given (including the years specified) will be included in the report. Save the macro as ModMacroSelect.sps.

c) For those with extra time: Modify the macro so that GROUP will accept a list of variables, and will use them to create additional subdivisions in the report. (Hint: the BY keyword must separate each grouping variable from the next in the SUMMARIZE command).