

GNU PSPP

A System for Statistical Analysis
Edition 0.4.0, for PSPP version 0.4.0

Ben Pfaff

This manual is for GNU PSPP version 0.4.0, software for statistical analysis.

Copyright © 1997, 1998, 2004, 2005 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with the Front-Cover Texts being “A GNU Manual,” and with the Back-Cover Texts as in (a) below. A copy of the license is included in the section entitled “GNU Free Documentation License.”

(a) The FSF’s Back-Cover Text is: “You have freedom to copy and modify this GNU Manual, like GNU software. Copies published by the Free Software Foundation raise funds for GNU development.”

Table of Contents

1	Introduction	1
2	Your rights and obligations	2
3	Invoking PSPP	3
3.1	Non-option Arguments	3
3.2	Configuration Options	3
3.3	Input and output options	4
3.4	Language control options	5
3.5	Informational options	6
4	The PSPP language	8
4.1	Tokens	8
4.2	Forming commands of tokens	9
4.3	Types of Commands	10
4.4	Order of Commands	10
4.5	Handling missing observations	12
4.6	Variables	12
4.6.1	Attributes of Variables	12
4.6.2	Variables Automatically Defined by PSPP	13
4.6.3	Lists of variable names	14
4.6.4	Input and Output Formats	14
4.6.5	Scratch Variables	19
4.7	Files Used by PSPP	19
4.8	Backus-Naur Form	19
5	Mathematical Expressions	21
5.1	Boolean Values	21
5.2	Missing Values in Expressions	21
5.3	Grouping Operators	21
5.4	Arithmetic Operators	21
5.5	Logical Operators	22
5.6	Relational Operators	22
5.7	Functions	23
5.7.1	Mathematical Functions	23
5.7.2	Miscellaneous Mathematical Functions	23
5.7.3	Trigonometric Functions	24
5.7.4	Missing-Value Functions	24
5.7.5	Set-Membership Functions	25
5.7.6	Statistical Functions	25
5.7.7	String Functions	26
5.7.8	Time & Date Functions	28

5.7.8.1	How times & dates are defined and represented	28
5.7.8.2	Functions that Produce Times	28
5.7.8.3	Functions that Examine Times	29
5.7.8.4	Functions that Produce Dates	29
5.7.8.5	Functions that Examine Dates	30
5.7.9	Miscellaneous Functions	31
5.7.10	Statistical Distribution Functions	32
5.7.10.1	Continuous Distributions	32
5.7.10.2	Discrete Distributions	37
5.8	Operator Precedence	37
6	Data Input and Output	39
6.1	BEGIN DATA	39
6.2	CLEAR TRANSFORMATIONS	39
6.3	DATA LIST	39
6.3.1	DATA LIST FIXED	39
	Examples	41
6.3.2	DATA LIST FREE	42
6.3.3	DATA LIST LIST	43
6.4	END CASE	43
6.5	END FILE	43
6.6	FILE HANDLE	43
6.7	INPUT PROGRAM	44
6.8	LIST	46
6.9	MATRIX DATA	47
6.10	NEW FILE	49
6.11	PRINT	49
6.12	PRINT EJECT	50
6.13	PRINT SPACE	50
6.14	REREAD	50
6.15	REPEATING DATA	51
6.16	WRITE	52
7	System Files and Portable Files	53
7.1	APPLY DICTIONARY	53
7.2	EXPORT	53
7.3	GET	54
7.4	IMPORT	54
7.5	MATCH FILES	55
7.6	SAVE	56
7.7	SYSFILE INFO	56
7.8	XSAVE	57

8	Manipulating variables	58
8.1	ADD VALUE LABELS	58
8.2	DISPLAY	58
8.3	DISPLAY VECTORS	58
8.4	FORMATS	59
8.5	LEAVE	59
8.6	MISSING VALUES	60
8.7	MODIFY VARS	60
8.8	NUMERIC	61
8.9	PRINT FORMATS	61
8.10	RENAME VARIABLES	61
8.11	VALUE LABELS	61
8.12	STRING	62
8.13	VARIABLE LABELS	62
8.14	VARIABLE ALIGNMENT	62
8.15	VARIABLE WIDTH	62
8.16	VARIABLE LEVEL	63
8.17	VECTOR	63
8.18	WRITE FORMATS	63
9	Data transformations	64
9.1	AGGREGATE	64
9.2	AUTORECODE	67
9.3	COMPUTE	67
9.4	COUNT	67
9.5	FLIP	69
9.6	IF	69
9.7	RECODE	70
9.8	SORT CASES	71
10	Selecting data for analysis	72
10.1	FILTER	72
10.2	N OF CASES	72
10.3	PROCESS IF	73
10.4	SAMPLE	73
10.5	SELECT IF	74
10.6	SPLIT FILE	74
10.7	TEMPORARY	75
10.8	WEIGHT	75
11	Conditional and Looping Constructs	77
11.1	BREAK	77
11.2	DO IF	77
11.3	DO REPEAT	77
11.4	LOOP	78

12	Statistics	80
12.1	DESCRIPTIVES	80
12.2	FREQUENCIES	81
12.3	EXAMINE	83
12.4	CROSSTABS	84
12.5	T-TEST	86
12.5.1	One Sample Mode	87
12.5.2	Independent Samples Mode	87
12.5.3	Paired Samples Mode	88
12.6	ONEWAY	88
13	Utilities	89
13.1	COMMENT	89
13.2	DOCUMENT	89
13.3	DISPLAY DOCUMENTS	89
13.4	DISPLAY FILE LABEL	89
13.5	DROP DOCUMENTS	89
13.6	ECHO	90
13.7	ERASE	90
13.8	EXECUTE	90
13.9	FILE LABEL	90
13.10	FINISH	90
13.11	HOST	90
13.12	INCLUDE	90
13.13	PERMISSIONS	91
13.14	QUIT	91
13.15	SET	91
13.16	SHOW	96
13.17	SUBTITLE	96
13.18	TITLE	97
14	Not Implemented	98
15	Bugs	101
16	Function Index	102
17	Command Index	105
18	Concept Index	107
Appendix A	Installing PSPP	112
A.1	UNIX installation	112

Appendix B Configuring PSPP 113

B.1	Locating configuration files	113
B.2	Configuration techniques	114
B.3	Configuration files	114
B.4	Environment variables	115
B.4.1	Values of environment variables	115
B.4.2	Environment substitutions	115
B.4.3	Predefined environment variables	116
B.5	Output devices	116
B.5.1	Driver categories	116
B.5.2	Macro definitions	117
B.5.3	Driver definitions	117
B.5.4	Dimensions	118
B.5.5	Paper sizes	119
B.5.6	How lines are divided into types	119
B.5.7	How lines are divided into tokens	120
B.6	The PostScript driver class	121
B.6.1	PostScript output options	121
B.6.2	PostScript page options	122
B.6.3	PostScript file options	122
B.6.4	PostScript font options	123
B.6.5	PostScript line options	124
B.6.6	The PostScript prologue	124
B.6.7	PostScript encodings	126
B.7	The ASCII driver class	126
B.7.1	ASCII output options	126
B.7.2	ASCII page options	127
B.7.3	ASCII font options	128
B.8	The HTML driver class	130
B.8.1	The HTML prologue	131
B.9	Miscellaneous configuration	131
B.10	Improving output quality	133

Appendix C Portable File Format 134

C.1	Portable File Characters	134
C.2	Portable File Structure	134
C.3	Portable File Header	135
C.4	Version and Date Info Record	137
C.5	Identification Records	138
C.6	Variable Count Record	138
C.7	Case Weight Variable Record	138
C.8	Variable Records	138
C.9	Value Label Records	139
C.10	Portable File Data	139

Appendix D	Data File Format	140
D.1	File Header Record	140
D.2	Variable Record	142
D.3	Value Label Record	145
D.4	Value Label Variable Record	145
D.5	Document Record	145
D.6	Machine <code>int32</code> Info Record	146
D.7	Machine <code>flt64</code> Info Record	147
D.8	Auxilliary Variable Parameter Record	148
D.9	Long Variable Names Record	149
D.10	Miscellaneous Informational Records	149
D.11	Dictionary Termination Record	150
D.12	Data Record	150
Appendix E	q2c Input Format	152
E.1	Invoking q2c	152
E.2	q2c Input Structure	152
E.3	Grammar Rules	153
Appendix F	GNU Free Documentation License	157
F.1	ADDENDUM: How to use this License for your documents	163

1 Introduction

PSPP is a tool for statistical analysis of sampled data. It reads a syntax file and a data file, analyzes the data, and writes the results to a listing file or to standard output.

The language accepted by PSPP is similar to those accepted by SPSS statistical products. The details of PSPP's language are given later in this manual.

PSPP produces output in two forms: tables and charts. Both of these can be written in several formats; currently, ASCII, PostScript, and HTML are supported. In the future, more drivers, such as PCL and X Window System drivers, may be developed. For now, Ghostscript, available from the Free Software Foundation, may be used to convert PostScript chart output to other formats.

The current version of PSPP, 0.4.0, is woefully incomplete in terms of its statistical procedure support. PSPP is a work in progress. The author hopes to support fully support all features in the products that PSPP replaces, eventually. The author welcomes questions, comments, donations, and code submissions. See [Chapter 15 \[Submitting Bug Reports\]](#), [page 101](#), for instructions on contacting the author.

2 Your rights and obligations

PSPP is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of this program that they might get from you.

Specifically, we want to make sure that you have the right to give away copies of PSPP, that you receive source code or else can get it if you want it, that you can change these programs or use pieces of them in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of PSPP, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for PSPP. If these programs are modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

The precise conditions of the licenses for PSPP are found in the General Public Licenses that accompany them. This manual specifically is covered by the GNU Free Documentation License (see [Appendix F \[GNU Free Documentation License\]](#), page 157).

3 Invoking PSPP

```
pspp [ -B dir | --config-dir=dir ] [ -o device | --device=device ]
      [ -d var[=value] | --define=var[=value] ] [ -u var | --undef=var ]
      [ -f file | --out-file=file ] [ -p | --pipe ] [ -I- | --no-include ]
      [ -I dir | --include=dir ] [ -i | --interactive ]
      [ -n | --edit | --dry-run | --just-print | --recon ]
      [ -r | --no-statrc ] [ -h | --help ] [ -l | --list ]
      [ -c command | --command command ] [ -s | --safer ]
      [ --testing-mode ] [ -V | --version ] [ -v | --verbose ]
      [ key=value ] file...
```

3.1 Non-option Arguments

Syntax files and output device substitutions can be specified on PSPP's command line:

file

A file by itself on the command line will be executed as a syntax file. PSPP terminates after the syntax file runs, unless the `-i` or `--interactive` option is given (see [Section 3.4 \[Language control options\]](#), page 5).

file1 file2

When two or more filenames are given on the command line, the first syntax file is executed, then PSPP's dictionary is cleared, then the second syntax file is executed.

file1 + file2

If syntax files' names are delimited by a plus sign ('+'), then the dictionary is not cleared between their executions, as if they were concatenated together into a single file.

key=value

Defines an output device macro *key* to expand to *value*, overriding any macro having the same *key* defined in the device configuration file. See [Section B.5.2 \[Macro definitions\]](#), page 117.

There is one other way to specify a syntax file, if your operating system supports it. If you have a syntax file 'foobar.stat', put the notation

```
#! /usr/local/bin/pspp
```

at the top, and mark the file as executable with `chmod +x foobar.stat`. (If PSPP is not installed in '/usr/local/bin', then insert its actual installation directory into the syntax file instead.) Now you should be able to invoke the syntax file just by typing its name. You can include any options on the command line as usual. PSPP entirely ignores any lines beginning with '#!'.

3.2 Configuration Options

Configuration options are used to change PSPP's configuration for the current run. The configuration options are:

-a {compatible|enhanced}
--algorithm={compatible|enhanced}
 If you chose **compatible**, then PSPP will use the same algorithms as used by some proprietary statistical analysis packages. This is not recommended, as these algorithms are inferior and in some cases completely broken. The default setting is **enhanced**. Certain commands have subcommands which allow you to override this setting on a per command basis.

-B dir
--config-dir=dir
 Sets the configuration directory to *dir*. See [Section B.1 \[File locations\]](#), page 113.

-o device
--device=device
 Selects the output device with name *device*. If this option is given more than once, then all devices mentioned are selected. This option disables all devices besides those mentioned on the command line.

-d var[=value]
--define=var[=value]
 Defines an ‘environment variable’ named *var* having the optional value *value* specified. See [Section B.4.1 \[Variable values\]](#), page 115.

-u var
--undef=var
 Undefines the ‘environment variable’ named *var*. See [Section B.4.1 \[Variable values\]](#), page 115.

3.3 Input and output options

Input and output options affect how PSPP reads input and writes output. These are the input and output options:

-f file
--out-file=file
 This overrides the output file name for devices designated as listing devices. If a file named *file* already exists, it is overwritten.

-p
--pipe
 Allows PSPP to be used as a filter by causing the syntax file to be read from stdin and output to be written to stdout. Conflicts with the **-f file** and **--file=file** options.

-I-
--no-include
 Clears all directories from the include path. This includes all directories put in the include path by default. See [Section B.9 \[Miscellaneous configuring\]](#), page 131.

-I *dir*
--include=*dir*
 Appends directory *dir* to the path that is searched for include files in PSPP syntax files.

-c *command*
--command=*command*
 Execute literal command *command*. The command is executed before startup syntax files, if any.

--testing-mode
 Invoke heuristics to assist with testing PSPP. For use by **make check** and similar scripts.

3.4 Language control options

Language control options control how PSPP syntax files are parsed and interpreted. The available language control options are:

-i
--interactive
 When a syntax file is specified on the command line, PSPP normally terminates after processing it. Giving this option will cause PSPP to bring up a command prompt after processing the syntax file.

In addition, this forces syntax files to be interpreted in interactive mode, rather than the default batch mode. See [Section B.5.7 \[Tokenizing lines\], page 120](#), for information on the differences between batch mode and interactive mode command interpretation.

-n
--edit
--dry-run
--just-print
--recon

Only the syntax of any syntax file specified or of commands entered at the command line is checked. Transformations are not performed and procedures are not executed. Not yet implemented.

-r
--no-statrc
 Prevents the execution of the PSPP startup syntax file. Not yet implemented, as startup syntax files aren't, either.

-s
--safer

Disables certain unsafe operations. This includes the ERASE and HOST commands, as well as use of pipes as input and output files.

3.5 Informational options

Informational options cause information about PSPP to be written to the terminal. Here are the available options:

-h

--help

Prints a message describing PSPP command-line syntax and the available device driver classes, then terminates.

-l

--list

Lists the available device driver classes, then terminates.

-x {compatible|enhanced}

--syntax={compatible|enhanced}

If you chose **compatible**, then PSPP will only accept command syntax that is compatible with the proprietary program SPSS. If you choose **enhanced** then additional syntax will be available. The default is **enhanced**.

-V

--version

Prints a brief message listing PSPP's version, warranties you don't have, copying conditions and copyright, and e-mail address for bug reports, then terminates.

-v

--verbose

Increments PSPP's verbosity level. Higher verbosity levels cause PSPP to display greater amounts of information about what it is doing. Often useful for debugging PSPP's configuration.

This option can be given multiple times to set the verbosity level to that value. The default verbosity level is 0, in which no informational messages will be displayed.

Higher verbosity levels cause messages to be displayed when the corresponding events take place.

1

Driver and subsystem initializations.

2

Completion of driver initializations. Beginning of driver closings.

3

Completion of driver closings.

4

Files searched for; success of searches.

5

Individual directories included in file searches.

Each verbosity level also includes messages from lower verbosity levels.

4 The PSPP language

Please note: PSPP is not even close to completion. Only a few statistical procedures are implemented. PSPP is a work in progress.

This chapter discusses elements common to many PSPP commands. Later chapters will describe individual commands in detail.

4.1 Tokens

PSPP divides most syntax file lines into series of short chunks called *tokens*. Tokens are then grouped to form commands, each of which tells PSPP to take some action—read in data, write out data, perform a statistical procedure, etc. Each type of token is described below.

Identifiers Identifiers are names that typically specify variables, commands, or subcommands. The first character in an identifier must be a letter, '#', or '@'. The remaining characters in the identifier must be letters, digits, or one of the following special characters:

. _ \$ # @

Identifiers may be any length, but only the first 64 bytes are significant. Identifiers are not case-sensitive: `foobar`, `Foobar`, `FooBar`, `FOOBAR`, and `Fo0baR` are different representations of the same identifier.

Some identifiers are reserved. Reserved identifiers may not be used in any context besides those explicitly described in this manual. The reserved identifiers are:

ALL AND BY EQ GE GT LE LT NE NOT OR TO WITH

Keywords Keywords are a subclass of identifiers that form a fixed part of command syntax. For example, command and subcommand names are keywords. Keywords may be abbreviated to their first 3 characters if this abbreviation is unambiguous. (Unique abbreviations of 3 or more characters are also accepted: `'FRE'`, `'FREQ'`, and `'FREQUENCIES'` are equivalent when the last is a keyword.)

Reserved identifiers are always used as keywords. Other identifiers may be used both as keywords and as user-defined identifiers, such as variable names.

Numbers Numbers are expressed in decimal. A decimal point is optional. Numbers may be expressed in scientific notation by adding `'e'` and a base-10 exponent, so that `'1.234e3'` has the value 1234. Here are some more examples of valid numbers:

-5 3.14159265359 1e100 -.707 8945.

Negative numbers are expressed with a `'-'` prefix. However, in situations where a literal `'-'` token is expected, what appears to be a negative number is treated as `'-'` followed by a positive number.

No white space is allowed within a number token, except for horizontal white space between `'-'` and the rest of the number.

The last example above, `'8945.'` will be interpreted as two tokens, `'8945'` and `'.'`, if it is the last token on a line. See [Section 4.2 \[Forming commands of tokens\]](#), page 9.

Strings Strings are literal sequences of characters enclosed in pairs of single quotes (‘ ’) or double quotes (‘ ” ’). To include the character used for quoting in the string, double it, e.g. ‘ ‘it’ ’s an apostrophe ’ ’. White space and case of letters are significant inside strings.

Strings can be concatenated using ‘+’, so that ‘ “a” + ’b’ + ’c ’ ’ is equivalent to ‘ ’abc ’ ’. Concatenation is useful for splitting a single string across multiple source lines. The maximum length of a string, after concatenation, is 255 characters.

Strings may also be expressed as hexadecimal, octal, or binary character values by prefixing the initial quote character by ‘X’, ‘O’, or ‘B’ or their lowercase equivalents. Each pair, triplet, or octet of characters, according to the radix, is transformed into a single character with the given value. If there is an incomplete group of characters, the missing final digits are assumed to be ‘0’. These forms of strings are nonportable because numeric values are associated with different characters by different operating systems. Therefore, their use should be confined to syntax files that will not be widely distributed.

The character with value 00 is reserved for internal use by PSPP. Its use in strings causes an error and replacement by a space character.

Punctuators and Operators

These tokens are the punctuators and operators:

, / = () + - * / ** < <= <> > >= ~ = & | .

Most of these appear within the syntax of commands, but the period (‘.’) punctuator is used only at the end of a command. It is a punctuator only as the last character on a line (except white space). When it is the last non-space character on a line, a period is not treated as part of another token, even if it would otherwise be part of, e.g., an identifier or a floating-point number.

Actually, the character that ends a command can be changed with SET’s END-CMD subcommand (see [Section 13.15 \[SET\], page 91](#)), but we do not recommend doing so. Throughout the remainder of this manual we will assume that the default setting is in effect.

4.2 Forming commands of tokens

Most PSPP commands share a common structure. A command begins with a command name, such as FREQUENCIES, DATA LIST, or N OF CASES. The command name may be abbreviated to its first word, and each word in the command name may be abbreviated to its first three or more characters, where these abbreviations are unambiguous.

The command name may be followed by one or more *subcommands*. Each subcommand begins with a subcommand name, which may be abbreviated to its first three letters. Some subcommands accept a series of one or more specifications, which follow the subcommand name, optionally separated from it by an equals sign (‘=’). Specifications may be separated from each other by commas or spaces. Each subcommand must be separated from the next (if any) by a forward slash (‘/’).

There are multiple ways to mark the end of a command. The most common way is to end the last line of the command with a period (‘.’) as described in the previous section (see

[Section 4.1 \[Tokens\], page 8](#)). A blank line, or one that consists only of white space or comments, also ends a command by default, although you can use the `NULLLINE` subcommand of `SET` to disable this feature (see [Section 13.15 \[SET\], page 91](#)).

In batch mode only, that is, when reading commands from a file instead of an interactive user, any line that contains a non-space character in the leftmost column begins a new command. Thus, each command consists of a flush-left line followed by any number of lines indented from the left margin. In this mode, a plus sign, minus sign, or period (`+`, `-`, or `.`) as the first character in a line is ignored and causes that line to begin a new command, which allows for visual indentation of a command without that command being considered part of the previous command.

Sometimes, one encounters syntax files that are intended to be interpreted in interactive mode rather than batch mode. When this occurs, use the `-i` command line option to force interpretation in interactive mode (see [Section 3.4 \[Language control options\], page 5](#)).

4.3 Types of Commands

Commands in PSPP are divided roughly into six categories:

Utility commands

Set or display various global options that affect PSPP operations. May appear anywhere in a syntax file. See [Chapter 13 \[Utility commands\], page 89](#).

File definition commands

Give instructions for reading data from text files or from special binary “system files”. Most of these commands replace any previous data or variables with new data or variables. At least one file definition command must appear before the first command in any of the categories below. See [Chapter 6 \[Data Input and Output\], page 39](#).

Input program commands

Though rarely used, these provide tools for reading data files in arbitrary textual or binary formats. See [Section 6.7 \[INPUT PROGRAM\], page 44](#).

Transformations

Perform operations on data and write data to output files. Transformations are not carried out until a procedure is executed.

Restricted transformations

Transformations that cannot appear in certain contexts. See [Section 4.4 \[Order of Commands\], page 10](#), for details.

Procedures

Analyze data, writing results of analyses to the listing file. Cause transformations specified earlier in the file to be performed. In a more general sense, a *procedure* is any command that causes the active file (the data) to be read.

4.4 Order of Commands

PSPP does not place many restrictions on ordering of commands. The main restriction is that variables must be defined before they are otherwise referenced. This section describes the details of command ordering, but most users will have no need to refer to them.

PSPP possesses five internal states, called initial, INPUT PROGRAM, FILE TYPE, transformation, and procedure states. (Please note the distinction between the INPUT PROGRAM and FILE TYPE *commands* and the INPUT PROGRAM and FILE TYPE *states*.)

PSPP starts in the initial state. Each successful completion of a command may cause a state transition. Each type of command has its own rules for state transitions:

Utility commands

- Valid in any state.
- Do not cause state transitions. Exception: when N OF CASES is executed in the procedure state, it causes a transition to the transformation state.

DATA LIST

- Valid in any state.
- When executed in the initial or procedure state, causes a transition to the transformation state.
- Clears the active file if executed in the procedure or transformation state.

INPUT PROGRAM

- Invalid in INPUT PROGRAM and FILE TYPE states.
- Causes a transition to the INPUT PROGRAM state.
- Clears the active file.

FILE TYPE

- Invalid in INPUT PROGRAM and FILE TYPE states.
- Causes a transition to the FILE TYPE state.
- Clears the active file.

Other file definition commands

- Invalid in INPUT PROGRAM and FILE TYPE states.
- Cause a transition to the transformation state.
- Clear the active file, except for ADD FILES, MATCH FILES, and UPDATE.

Transformations

- Invalid in initial and FILE TYPE states.
- Cause a transition to the transformation state.

Restricted transformations

- Invalid in initial, INPUT PROGRAM, and FILE TYPE states.
- Cause a transition to the transformation state.

Procedures

- Invalid in initial, INPUT PROGRAM, and FILE TYPE states.
- Cause a transition to the procedure state.

4.5 Handling missing observations

PSPP includes special support for unknown numeric data values. Missing observations are assigned a special value, called the *system-missing value*. This “value” actually indicates the absence of a value; it means that the actual value is unknown. Procedures automatically exclude from analyses those observations or cases that have missing values. Details of missing value exclusion depend on the procedure and can often be controlled by the user; refer to descriptions of individual procedures for details.

The system-missing value exists only for numeric variables. String variables always have a defined value, even if it is only a string of spaces.

Variables, whether numeric or string, can have designated *user-missing values*. Every user-missing value is an actual value for that variable. However, most of the time user-missing values are treated in the same way as the system-missing value. String variables that are wider than a certain width, usually 8 characters (depending on computer architecture), cannot have user-missing values.

For more information on missing values, see the following sections: [Section 4.6 \[Variables\], page 12](#), [Section 8.6 \[MISSING VALUES\], page 60](#), [Chapter 5 \[Expressions\], page 21](#). See also the documentation on individual procedures for information on how they handle missing values.

4.6 Variables

Variables are the basic unit of data storage in PSPP. All the variables in a file taken together, apart from any associated data, are said to form a *dictionary*. Some details of variables are described in the sections below.

4.6.1 Attributes of Variables

Each variable has a number of attributes, including:

Name	<p>An identifier, up to 64 bytes long. Each variable must have a different name. See Section 4.1 [Tokens], page 8.</p> <p>Some system variable names begin with ‘\$’, but user-defined variables’ names may not begin with ‘\$’.</p> <p>The final character in a variable name should not be ‘.’, because such an identifier will be misinterpreted when it is the final token on a line: <code>F00.</code> will be divided into two separate tokens, ‘F00’ and ‘.’, indicating end-of-command. See Section 4.1 [Tokens], page 8.</p> <p>The final character in a variable name should not be ‘_’, because some such identifiers are used for special purposes by PSPP procedures.</p> <p>As with all PSPP identifiers, variable names are not case-sensitive. PSPP capitalizes variable names on output the same way they were capitalized at their point of definition in the input.</p>
Type	Numeric or string.
Width	(string variables only) String variables with a width of 8 characters or fewer are called <i>short string variables</i> . Short string variables can be used in many

procedures where *long string variables* (those with widths greater than 8) are not allowed.

Certain systems may consider strings longer than 8 characters to be short strings. Eight characters represents a minimum figure for the maximum length of a short string.

Position Variables in the dictionary are arranged in a specific order. DISPLAY can be used to show this order: see [Section 8.2 \[DISPLAY\]](#), page 58.

Initialization

Either reinitialized to 0 or spaces for each case, or left at its existing value. See [Section 8.5 \[LEAVE\]](#), page 59.

Missing values

Optionally, up to three values, or a range of values, or a specific value plus a range, can be specified as *user-missing values*. There is also a *system-missing value* that is assigned to an observation when there is no other obvious value for that observation. Observations with missing values are automatically excluded from analyses. User-missing values are actual data values, while the system-missing value is not a value at all. See [Section 4.5 \[Missing Observations\]](#), page 12.

Variable label

A string that describes the variable. See [Section 8.13 \[VARIABLE LABELS\]](#), page 62.

Value label

Optionally, these associate each possible value of the variable with a string. See [Section 8.11 \[VALUE LABELS\]](#), page 61.

Print format

Display width, format, and (for numeric variables) number of decimal places. This attribute does not affect how data are stored, just how they are displayed. Example: a width of 8, with 2 decimal places. See [Section 8.9 \[PRINT FORMATS\]](#), page 61.

Write format

Similar to print format, but used by certain commands that are designed to write to binary files. See [Section 8.18 \[WRITE FORMATS\]](#), page 63.

4.6.2 Variables Automatically Defined by PSPP

There are seven system variables. These are not like ordinary variables because system variables are not always stored. They can be used only in expressions. These system variables, whose values and output formats cannot be modified, are described below.

- \$CASENUM** Case number of the case at the moment. This changes as cases are shuffled around.
- \$DATE** Date the PSPP process was started, in format A9, following the pattern DD MMM YY.
- \$JDATE** Number of days between 15 Oct 1582 and the time the PSPP process was started.

\$LENGTH	Page length, in lines, in format F11.
\$SYSMIS	System missing value, in format F1.
\$TIME	Number of seconds between midnight 14 Oct 1582 and the time the active file was read, in format F20.
\$WIDTH	Page width, in characters, in format F3.

4.6.3 Lists of variable names

To refer to a set of variables, list their names one after another. Optionally, their names may be separated by commas. To include a range of variables from the dictionary in the list, write the name of the first and last variable in the range, separated by **T0**. For instance, if the dictionary contains six variables with the names **ID**, **X1**, **X2**, **GOAL**, **MET**, and **NEXTGOAL**, in that order, then **X2 T0 MET** would include variables **X2**, **GOAL**, and **MET**.

Commands that define variables, such as **DATA LIST**, give **T0** an alternate meaning. With these commands, **T0** define sequences of variables whose names end in consecutive integers. The syntax is two identifiers that begin with the same root and end with numbers, separated by **T0**. The syntax **X1 T0 X5** defines 5 variables, named **X1**, **X2**, **X3**, **X4**, and **X5**. The syntax **ITEM0008 T0 ITEM0013** defines 6 variables, named **ITEM0008**, **ITEM0009**, **ITEM0010**, **ITEM0011**, **ITEM0012**, and **ITEM0013**. The syntaxes **QUES001 T0 QUES9** and **QUES6 T0 QUES3** are invalid.

After a set of variables has been defined with **DATA LIST** or another command with this method, the same set can be referenced on later commands using the same syntax.

4.6.4 Input and Output Formats

Data that PSPP inputs and outputs must have one of a number of formats. These formats are described, in general, by a format specification of the form **NAMEw.d**, where *name* is the format name and *w* is a field width. *d* is the optional desired number of decimal places, if appropriate. If *d* is not included then it is assumed to be 0. Some formats do not allow *d* to be specified.

When **DATA LIST** or another command specifies an input format, that format is converted to an output format for the purposes of **PRINT** and other data output commands. For most purposes, input and output formats are the same; the salient differences are described below.

Below are listed the input and output formats supported by PSPP. If an input format is mapped to a different output format by default, then that mapping is indicated with \Rightarrow . Each format has the listed bounds on input width (*iw*) and output width (*ow*).

The standard numeric input and output formats are given in the following table:

Fw.d: 1 <= *iw*, *ow* <= 40

Standard decimal format with *d* decimal places. If the number is too large to fit within the field width, it is expressed in scientific notation (**1.2+34**) if *w* >= 6, with always at least two digits in the exponent. When used as an input format, scientific notation is allowed but an **E** or an **F** must be used to introduce the exponent.

The default output format is the same as the input format, except if *d* > 1. In that case the output *w* is always made to be at least 2 + *d*.

Ew.d: 1 <= iw <= 40; 6 <= ow <= 40

For input this is equivalent to F format except that no E or F is required to introduce the exponent. For output, produces scientific notation in the form 1.2+34. There are always at least two digits given in the exponent.

The default output w is the largest of the input w , the input $d + 7$, and 10. The default output d is the input d , but at least 3.

COMMAw.d: 1 <= iw,ow <= 40

Equivalent to F format, except that groups of three digits are comma-separated on output. If the number is too large to express in the field width, then first commas are eliminated, then if there is still not enough space the number is expressed in scientific notation given that $w \geq 6$. Commas are allowed and ignored when this is used as an input format.

DOTw.d: 1 <= iw,ow <= 40

Equivalent to COMMA format except that the roles of comma and decimal point are interchanged. However: If SET /DECIMAL=DOT is in effect, then COMMA uses ‘,’ for a decimal point and DOT uses ‘.’ for a decimal point.

DOLLARw.d: 1 <= iw <= 40; 2 <= ow <= 40

Equivalent to COMMA format, except that the number is prefixed by a dollar sign (\$) if there is room. On input the value is allowed to be prefixed by a dollar sign, which is ignored.

The default output w is the input w , but at least 2.

PCTw.d: 2 <= iw,ow <= 40

Equivalent to F format, except that the number is suffixed by a percent sign (%) if there is room. On input the value is allowed to be suffixed by a percent sign, which is ignored.

The default output w is the input w , but at least 2.

Nw.d: 1 <= iw,ow <= 40

Only digits are allowed within the field width. The decimal point is assumed to be d digits from the right margin.

The default output format is F with the same w and d , except if $d > 1$. In that case the output w is always made to be at least $2 + d$.

Zw.d \Rightarrow F: 1 <= iw,ow <= 40

Zoned decimal input. If you need to use this then you know how.

IBw.d \Rightarrow F: 1 <= iw,ow <= 8

Integer binary format. The field is interpreted as a fixed-point positive or negative binary number in two's-complement notation. The location of the decimal point is implied. Endianness is the same as the host machine.

The default output format is F8.2 if d is 0. Otherwise it is F, with output w as $9 + \text{input } d$ and output d as input d .

PIB \Rightarrow F: 1 <= iw,ow <= 8

Positive integer binary format. The field is interpreted as a fixed-point positive binary number. The location of the decimal point is implied. Endianness is the same as the host machine.

The default output format follows the rules for IB format.

Pw.d \Rightarrow F: 1 \leq iw, ow \leq 16

Binary coded decimal format. Each byte from left to right, except the rightmost, represents two digits. The upper nibble of each byte is more significant. The upper nibble of the final byte is the least significant digit. The lower nibble of the final byte is the sign; a value of D represents a negative sign and all other values are considered positive. The decimal point is implied.

The default output format follows the rules for IB format.

PKw.d \Rightarrow F: 1 \leq iw, ow \leq 16

Positive binary code decimal format. Same as P but the last byte is the same as the others.

The default output format follows the rules for IB format.

RBw \Rightarrow F: 2 \leq iw, ow \leq 8

Binary C architecture-dependent “double” format. For a standard IEEE754 implementation w should be 8.

The default output format follows the rules for IB format.

PIBHEXw.d \Rightarrow F: 2 \leq iw, ow \leq 16

PIB format encoded as textual hex digit pairs. w must be even.

The input width is mapped to a default output width as follows: 2 \Rightarrow 4, 4 \Rightarrow 6, 6 \Rightarrow 9, 8 \Rightarrow 11, 10 \Rightarrow 14, 12 \Rightarrow 16, 14 \Rightarrow 18, 16 \Rightarrow 21. No allowances are made for decimal places.

RBHEXw \Rightarrow F: 4 \leq iw, ow \leq 16

RB format encoded as textual hex digits pairs. w must be even.

The default output format is F8.2.

CCAw.d: 1 \leq ow \leq 40

CCBw.d: 1 \leq ow \leq 40

CCCw.d: 1 \leq ow \leq 40

CCDw.d: 1 \leq ow \leq 40

CCEw.d: 1 \leq ow \leq 40

User-defined custom currency formats. May not be used as an input format.

See [Section 13.15 \[SET\], page 91](#), for more details.

The date and time numeric input and output formats accept a number of possible formats. Before describing the formats themselves, some definitions of the elements that make up their formats will be helpful:

leader All formats accept an optional white space leader.

day An integer between 1 and 31 representing the day of month.

day-count An integer representing a number of days.

date-delimiter

One or more characters of white space or the following characters: - / . ,

month A month name in one of the following forms:

	<ul style="list-style-type: none"> • An integer between 1 and 12. • Roman numerals representing an integer between 1 and 12. • At least the first three characters of an English month name (January, February, ...).
<i>year</i>	An integer year number between 1582 and 19999, or between 1 and 199. Years between 1 and 199 will have 1900 added.
<i>julian</i>	A single number with a year number in the first 2, 3, or 4 digits (as above) and the day number within the year in the last 3 digits.
<i>quarter</i>	An integer between 1 and 4 representing a quarter.
<i>q-delimiter</i>	The letter 'Q' or 'q'.
<i>week</i>	An integer between 1 and 53 representing a week within a year.
<i>wk-delimiter</i>	The letters 'wk' in any case.
<i>time-delimiter</i>	At least one characters of white space or ':' or '.'.
<i>hour</i>	An integer greater than 0 representing an hour.
<i>minute</i>	An integer between 0 and 59 representing a minute within an hour.
<i>opt-second</i>	Optionally, a time-delimiter followed by a real number representing a number of seconds.
<i>hour24</i>	An integer between 0 and 23 representing an hour within a day.
<i>weekday</i>	At least the first two characters of an English day word.
<i>spaces</i>	Any amount or no amount of white space.
<i>sign</i>	An optional positive or negative sign.
<i>trailer</i>	All formats accept an optional white space trailer.

The date input formats are strung together from the above pieces. On output, the date formats are always printed in a single canonical manner, based on field width. The date input and output formats are described below:

DATEw: 9 <= iw,ow <= 40

Date format. Input format: leader + day + date-delimiter + month + date-delimiter + year + trailer. Output format: DD-MMM-YY for w < 11, DD-MMM-YYYY otherwise.

EDATEw: 8 <= iw,ow <= 40

European date format. Input format same as DATE. Output format: DD.MM.YY for w < 10, DD.MM.YYYY otherwise.

SDATEw: 8 <= iw,ow <= 40

Standard date format. Input format: leader + year + date-delimiter + month + date-delimiter + day + trailer. Output format: YY/MM/DD for w < 10, YYYY/MM/DD otherwise.

ADATEw: 8 <= iw, ow <= 40

American date format. Input format: leader + month + date-delimiter + day + date-delimiter + year + trailer. Output format: MM/DD/YY for w < 10, MM/DD/YYYY otherwise.

JDATEw: 5 <= iw, ow <= 40

Julian date format. Input format: leader + julian + trailer. Output format: YYDDD for w < 7, YYYYDDD otherwise.

QYRw: 4 <= iw <= 40, 6 <= ow <= 40

Quarter/year format. Input format: leader + quarter + q-delimiter + year + trailer. Output format: 'Q Q YY', where the first 'Q' is one of the digits 1, 2, 3, 4, if w < 8, Q Q YYYY otherwise.

MOYRw: 6 <= iw, ow <= 40

Month/year format. Input format: leader + month + date-delimiter + year + trailer. Output format: 'MMM YY' for w < 8, 'MMM YYYY' otherwise.

WKYRw: 6 <= iw <= 40, 8 <= ow <= 40

Week/year format. Input format: leader + week + wk-delimiter + year + trailer. Output format: 'WW WK YY' for w < 10, 'WW WK YYYY' otherwise.

DATETIMEw.d: 17 <= iw, ow <= 40

Date and time format. Input format: leader + day + date-delimiter + month + date-delimiter + year + time-delimiter + hour24 + time-delimiter + minute + opt-second. Output format: 'DD-MMM-YYYY HH:MM'. If w > 19 then seconds ':SS' is added. If w > 22 and d > 0 then fractional seconds '.SS' are added.

TIMEw.d: 5 <= iw, ow <= 40

Time format. Input format: leader + sign + spaces + hour + time-delimiter + minute + opt-second. Output format: 'HH:MM'. Seconds and fractional seconds are available with w of at least 8 and 10, respectively.

DTIMEw.d: 1 <= iw <= 40, 8 <= ow <= 40

Time format with day count. Input format: leader + sign + spaces + day-count + time-delimiter + hour + time-delimiter + minute + opt-second. Output format: 'DD HH:MM'. Seconds and fractional seconds are available with w of at least 8 and 10, respectively.

WKDAYw: 2 <= iw, ow <= 40

A weekday as a number between 1 and 7, where 1 is Sunday. Input format: leader + weekday + trailer. Output format: as many characters, in all capital letters, of the English name of the weekday as will fit in the field width.

MONTHw: 3 <= iw, ow <= 40

A month as a number between 1 and 12, where 1 is January. Input format: leader + month + trailer. Output format: as many character, in all capital letters, of the English name of the month as will fit in the field width.

There are only two formats that may be used with string variables:

Aw: 1 <= iw <= 255, 1 <= ow <= 254

The entire field is treated as a string value.

AHEXw \Rightarrow A: 2 <= iw <= 254; 2 <= ow <= 510

The field is composed of characters in a string encoded as textual hex digit pairs.

The default output w is half the input w.

4.6.5 Scratch Variables

Most of the time, variables don't retain their values between cases. Instead, either they're being read from a data file or the active file, in which case they assume the value read, or, if created with COMPUTE or another transformation, they're initialized to the system-missing value or to blanks, depending on type.

However, sometimes it's useful to have a variable that keeps its value between cases. You can do this with LEAVE (see [Section 8.5 \[LEAVE\], page 59](#)), or you can use a *scratch variable*. Scratch variables are variables whose names begin with an octothorpe ('#').

Scratch variables have the same properties as variables left with LEAVE: they retain their values between cases, and for the first case they are initialized to 0 or blanks. They have the additional property that they are deleted before the execution of any procedure. For this reason, scratch variables can't be used for analysis. To use a scratch variable in an analysis, use COMPUTE (see [Section 9.3 \[COMPUTE\], page 67](#)) to copy its value into an ordinary variable, then use that ordinary variable in the analysis.

4.7 Files Used by PSPP

PSPP makes use of many files each time it runs. Some of these it reads, some it writes, some it creates. Here is a table listing the most important of these files:

command file

syntax file These names (synonyms) refer to the file that contains instructions that tell PSPP what to do. The syntax file's name is specified on the PSPP command line. Syntax files can also be pulled in with INCLUDE (see [Section 13.12 \[INCLUDE\], page 90](#)).

data file Data files contain raw data in ASCII format suitable for being read in by DATA LIST. Data can be embedded in the syntax file with BEGIN DATA and END DATA: this makes the syntax file a data file too.

listing file One or more output files are created by PSPP each time it is run. The output files receive the tables and charts produced by statistical procedures. The output files may be in any number of formats, depending on how PSPP is configured.

active file The active file is the "file" on which all PSPP procedures are performed. The active file contains variable definitions and cases. The active file is not necessarily a disk file: it is stored in memory if there is room.

4.8 Backus-Naur Form

The syntax of some parts of the PSPP language is presented in this manual using the formalism known as *Backus-Naur Form*, or BNF. The following table describes BNF:

- Words in all-uppercase are PSPP keyword tokens. In BNF, these are often called *terminals*. There are some special terminals, which are written in lowercase for clarity:

number A real number.

integer An integer number.

string A string.

var-name A single variable name.

=, /, +, -, etc.

Operators and punctuators.

. The end of the command. This is not necessarily an actual dot in the syntax file: See [Section 4.2 \[Commands\]](#), page 9, for more details.

- Other words in all lowercase refer to BNF definitions, called *productions*. These productions are also known as *nonterminals*. Some nonterminals are very common, so they are defined here in English for clarity:

var-list A list of one or more variable names or the keyword **ALL**.

expression

An expression. See [Chapter 5 \[Expressions\]](#), page 21, for details.

- ‘**::=**’ means “is defined as”. The left side of ‘**::=**’ gives the name of the nonterminal being defined. The right side of ‘**::=**’ gives the definition of that nonterminal. If the right side is empty, then one possible expansion of that nonterminal is nothing. A BNF definition is called a *production*.
- So, the key difference between a terminal and a nonterminal is that a terminal cannot be broken into smaller parts—in fact, every terminal is a single token (see [Section 4.1 \[Tokens\]](#), page 8). On the other hand, nonterminals are composed of a (possibly empty) sequence of terminals and nonterminals. Thus, terminals indicate the deepest level of syntax description. (In parsing theory, terminals are the leaves of the parse tree; nonterminals form the branches.)
- The first nonterminal defined in a set of productions is called the *start symbol*. The start symbol defines the entire syntax for that command.

5 Mathematical Expressions

Expressions share a common syntax each place they appear in PSPP commands. Expressions are made up of *operands*, which can be numbers, strings, or variable names, separated by *operators*. There are five types of operators: grouping, arithmetic, logical, relational, and functions.

Every operator takes one or more operands as input and yields exactly one result as output. Depending on the operator, operands accept strings or numbers as operands. With few exceptions, operands may be full-fledged expressions in themselves.

5.1 Boolean Values

Some PSPP operators and expressions work with Boolean values, which represent true/false conditions. Booleans have only three possible values: 0 (false), 1 (true), and system-missing (unknown). System-missing is neither true nor false and indicates that the true value is unknown.

Boolean-typed operands or function arguments must take on one of these three values. Other values are considered false, but provoke a warning when the expression is evaluated.

Strings and Booleans are not compatible, and neither may be used in place of the other.

5.2 Missing Values in Expressions

Most numeric operators yield system-missing when given any system-missing operand. A string operator given any system-missing operand typically results in the empty string. Exceptions are listed under particular operator descriptions.

String user-missing values are not treated specially in expressions.

User-missing values for numeric variables are always transformed into the system-missing value, except inside the arguments to the `VALUE` and `SYSMIS` functions.

The missing-value functions can be used to precisely control how missing values are treated in expressions. See [Section 5.7.4 \[Missing Value Functions\]](#), [page 24](#), for more details.

5.3 Grouping Operators

Parentheses (`()`) are the grouping operators. Surround an expression with parentheses to force early evaluation.

Parentheses also surround the arguments to functions, but in that situation they act as punctuators, not as operators.

5.4 Arithmetic Operators

The arithmetic operators take numeric operands and produce numeric results.

- | | |
|---------|--|
| $a + b$ | Yields the sum of a and b . |
| $a - b$ | Subtracts b from a and yields the difference. |
| $a * b$ | Yields the product of a and b . If either a or b is 0, then the result is 0, even if the other operand is missing. |

a / b	Divides a by b and yields the quotient. If a is 0, then the result is 0, even if b is missing. If b is zero, the result is system-missing.
$a ** b$	Yields the result of raising a to the power b . If a is negative and b is not an integer, the result is system-missing. The result of $0**0$ is system-missing as well.
$- a$	Reverses the sign of a .

5.5 Logical Operators

The logical operators take logical operands and produce logical results, meaning “true or false.” Logical operators are not true Boolean operators because they may also result in a system-missing value. See [Section 5.1 \[Boolean Values\]](#), page 21, for more information.

$a \text{ AND } b$	
$a \& b$	True if both a and b are true, false otherwise. If one operand is false, the result is false even if the other is missing. If both operands are missing, the result is missing.
$a \text{ OR } b$	
$a b$	True if at least one of a and b is true. If one operand is true, the result is true even if the other operand is missing. If both operands are missing, the result is missing.
$\text{NOT } a$	
$\sim a$	True if a is false. If the operand is missing, then the result is missing.

5.6 Relational Operators

The relational operators take numeric or string operands and produce Boolean results.

Strings cannot be compared to numbers. When strings of different lengths are compared, the shorter string is right-padded with spaces to match the length of the longer string.

The results of string comparisons, other than tests for equality or inequality, depend on the character set in use. String comparisons are case-sensitive.

$a \text{ EQ } b$	
$a = b$	True if a is equal to b .
$a \text{ LE } b$	
$a \leq b$	True if a is less than or equal to b .
$a \text{ LT } b$	
$a < b$	True if a is less than b .
$a \text{ GE } b$	
$a \geq b$	True if a is greater than or equal to b .
$a \text{ GT } b$	
$a > b$	True if a is greater than b .
$a \text{ NE } b$	
$a \neq b$	
$a <> b$	True if a is not equal to b .

5.7 Functions

PSPP functions provide mathematical abilities above and beyond those possible using simple operators. Functions have a common syntax: each is composed of a function name followed by a left parenthesis, one or more arguments, and a right parenthesis.

Function names are not reserved. Their names are specially treated only when followed by a left parenthesis, so that **EXP**(10) refers to the constant value **e** raised to the 10th power, but **EXP** by itself refers to the value of variable **EXP**.

The sections below describe each function in detail.

5.7.1 Mathematical Functions

Advanced mathematical functions take numeric arguments and produce numeric results.

EXP (*exponent*) [Function]

Returns e (approximately 2.71828) raised to power *exponent*.

LG10 (*number*) [Function]

Takes the base-10 logarithm of *number*. If *number* is not positive, the result is system-missing.

LN (*number*) [Function]

Takes the base- e logarithm of *number*. If *number* is not positive, the result is system-missing.

LNGAMMA (*number*) [Function]

Yields the base- e logarithm of the complete gamma of *number*. If *number* is a negative integer, the result is system-missing.

SQRT (*number*) [Function]

Takes the square root of *number*. If *number* is negative, the result is system-missing.

5.7.2 Miscellaneous Mathematical Functions

Miscellaneous mathematical functions take numeric arguments and produce numeric results.

ABS (*number*) [Function]

Results in the absolute value of *number*.

MOD (*numerator*, *denominator*) [Function]

Returns the remainder (modulus) of *numerator* divided by *denominator*. If *numerator* is 0, then the result is 0, even if *denominator* is missing. If *denominator* is 0, the result is system-missing.

MOD10 (*number*) [Function]

Returns the remainder when *number* is divided by 10. If *number* is negative, MOD10(*number*) is negative or zero.

RND (*number*) [Function]

Takes the absolute value of *number* and rounds it to an integer. Then, if *number* was negative originally, negates the result.

TRUNC (*number*) [Function]

Discards the fractional part of *number*; that is, rounds *number* towards zero.

5.7.3 Trigonometric Functions

Trigonometric functions take numeric arguments and produce numeric results.

ARCOS (*number*) [Function]

ACOS (*number*) [Function]

Takes the arccosine, in radians, of *number*. Results in system-missing if *number* is not between -1 and 1 inclusive. This function is a PSPP extension.

ARSIN (*number*) [Function]

ASIN (*number*) [Function]

Takes the arcsine, in radians, of *number*. Results in system-missing if *number* is not between -1 and 1 inclusive.

ARTAN (*number*) [Function]

ATAN (*number*) [Function]

Takes the arctangent, in radians, of *number*.

COS (*angle*) [Function]

Takes the cosine of *angle* which should be in radians.

SIN (*angle*) [Function]

Takes the sine of *angle* which should be in radians.

TAN (*angle*) [Function]

Takes the tangent of *angle* which should be in radians. Results in system-missing at values of *angle* that are too close to odd multiples of $\pi/2$. Portability: none.

5.7.4 Missing-Value Functions

Missing-value functions take various numeric arguments and yield various types of results. Except where otherwise stated below, the normal rules of evaluation apply within expression arguments to these functions. In particular, user-missing values for numeric variables are converted to system-missing values.

MISSING (*expr*) [Function]

Returns 1 if *expr* has the system-missing value, 0 otherwise.

NMISS (*expr* [, *expr*]...) [Function]

Each argument must be a numeric expression. Returns the number of system-missing values in the list, which may include variable ranges using the **var1 TO var2** syntax.

NVALID (*expr* [, *expr*]...) [Function]

Each argument must be a numeric expression. Returns the number of values in the list that are not system-missing. The list may include variable ranges using the **var1 TO var2** syntax.

SYSMIS (*expr*) [Function]

When *expr* is simply the name of a numeric variable, returns 1 if the variable has the system-missing value, 0 if it is user-missing or not missing. If given *expr* takes another form, results in 1 if the value is system-missing, 0 otherwise.

VALUE (*variable*) [Function]

Prevents the user-missing values of *variable* from being transformed into system-missing values, and always results in the actual value of *variable*, whether it is valid, user-missing, or system-missing.

5.7.5 Set-Membership Functions

Set membership functions determine whether a value is a member of a set. They take a set of numeric arguments or a set of string arguments, and produce Boolean results.

String comparisons are performed according to the rules given in [Section 5.6 \[Relational Operators\]](#), page 22.

ANY (*value*, *set* [, *set*]...) [Function]

Results in true if *value* is equal to any of the *set* values. Otherwise, results in false. If *value* is system-missing, returns system-missing. System-missing values in *set* do not cause ANY to return system-missing.

RANGE (*value*, *low*, *high* [, *low*, *high*]...) [Function]

Results in true if *value* is in any of the intervals bounded by *low* and *high* inclusive. Otherwise, results in false. Each *low* must be less than or equal to its corresponding *high* value. *low* and *high* must be given in pairs. If *value* is system-missing, returns system-missing. System-missing values in *set* do not cause RANGE to return system-missing.

5.7.6 Statistical Functions

Statistical functions compute descriptive statistics on a list of values. Some statistics can be computed on numeric or string values; other can only be computed on numeric values. Their results have the same type as their arguments. The current case's weighting factor (see [Section 10.8 \[WEIGHT\]](#), page 75) has no effect on statistical functions.

These functions' argument lists may include entire ranges of variables using the **var1 TO var2** syntax.

Unlike most functions, statistical functions can return non-missing values even when some of their arguments are missing. Most statistical functions, by default, require only 1 non-missing value to have a non-missing return, but CFVAR, SD, and VARIANCE require 2. These defaults can be increased (but not decreased) by appending a dot and the minimum number of valid arguments to the function name. For example, MEAN.3(X, Y, Z) would only return non-missing if all of 'X', 'Y', and 'Z' were valid.

CFVAR (*number*, *number* [, ...]) [Function]

Results in the coefficient of variation of the values of *number*. (The coefficient of variation is the standard deviation divided by the mean.)

MAX (*value*, *value* [, ...]) [Function]

Results in the value of the greatest *value*. The *values* may be numeric or string.

MEAN (*number*, *number* [, ...]) [Function]

Results in the mean of the values of *number*.

MIN (*number*, *number* [, ...]) [Function]

Results in the value of the least *value*. The *values* may be numeric or string.

SD (*number*, *number*[, ...]) [Function]
Results in the standard deviation of the values of *number*.

SUM (*number*, *number*[, ...]) [Function]
Results in the sum of the values of *number*.

VARIANCE (*number*, *number*[, ...]) [Function]
Results in the variance of the values of *number*.

5.7.7 String Functions

String functions take various arguments and return various results.

CONCAT (*string*, *string*[, ...]) [Function]
Returns a string consisting of each *string* in sequence. **CONCAT**("abc", "def", "ghi") has a value of "abcdefghi". The resultant string is truncated to a maximum of 255 characters.

INDEX (*haystack*, *needle*) [Function]
Returns a positive integer indicating the position of the first occurrence of *needle* in *haystack*. Returns 0 if *haystack* does not contain *needle*. Returns system-missing if *needle* is an empty string.

INDEX (*haystack*, *needles*, *needle_len*) [Function]
Divides *needles* into one or more needles, each with length *needle_len*. Searches *haystack* for the first occurrence of each needle, and returns the smallest value. Returns 0 if *haystack* does not contain any part in *needle*. It is an error if *needle_len* does not evenly divide the length of *needles*. Returns system-missing if *needles* is an empty string.

LENGTH (*string*) [Function]
Returns the number of characters in *string*.

LOWER (*string*) [Function]
Returns a string identical to *string* except that all uppercase letters are changed to lowercase letters. The definitions of “uppercase” and “lowercase” are system-dependent.

LPAD (*string*, *length*) [Function]
If *string* is at least *length* characters in length, returns *string* unchanged. Otherwise, returns *string* padded with spaces on the left side to length *length*. Returns an empty string if *length* is system-missing, negative, or greater than 255.

LPAD (*string*, *length*, *padding*) [Function]
If *string* is at least *length* characters in length, returns *string* unchanged. Otherwise, returns *string* padded with *padding* on the left side to length *length*. Returns an empty string if *length* is system-missing, negative, or greater than 255, or if *padding* does not contain exactly one character.

LTRIM (*string*) [Function]
Returns *string*, after removing leading spaces. Other white space, such as tabs, carriage returns, line feeds, and vertical tabs, is not removed.

LTRIM (*string*, *padding*) [Function]

Returns *string*, after removing leading *padding* characters. If *padding* does not contain exactly one character, returns an empty string.

NUMBER (*string*, *format*) [Function]

Returns the number produced when *string* is interpreted according to format specifier *format*. If the format width *w* is less than the length of *string*, then only the first *w* characters in *string* are used, e.g. **NUMBER**("123", F3.0) and **NUMBER**("1234", F3.0) both have value 123. If *w* is greater than *string*'s length, then it is treated as if it were right-padded with spaces. If *string* is not in the correct format for *format*, system-missing is returned.

RINDEX (*string*, *format*) [Function]

Returns a positive integer indicating the position of the last occurrence of *needle* in *haystack*. Returns 0 if *haystack* does not contain *needle*. Returns system-missing if *needle* is an empty string.

RINDEX (*haystack*, *needle*, *needle_len*) [Function]

Divides *needle* into parts, each with length *needle_len*. Searches *haystack* for the last occurrence of each part, and returns the largest value. Returns 0 if *haystack* does not contain any part in *needle*. It is an error if *needle_len* does not evenly divide the length of *needle*. Returns system-missing if *needle* is an empty string.

RPAD (*string*, *length*) [Function]

If *string* is at least *length* characters in length, returns *string* unchanged. Otherwise, returns *string* padded with spaces on the right to length *length*. Returns an empty string if *length* is system-missing, negative, or greater than 255.

RPAD (*string*, *length*, *padding*) [Function]

If *string* is at least *length* characters in length, returns *string* unchanged. Otherwise, returns *string* padded with *padding* on the right to length *length*. Returns an empty string if *length* is system-missing, negative, or greater than 255, or if *padding* does not contain exactly one character.

RTRIM (*string*) [Function]

Returns *string*, after removing trailing spaces. Other types of white space are not removed.

RTRIM (*string*, *padding*) [Function]

Returns *string*, after removing trailing *padding* characters. If *padding* does not contain exactly one character, returns an empty string.

STRING (*number*, *format*) [Function]

Returns a string corresponding to *number* in the format given by format specifier *format*. For example, **STRING**(123.56, F5.1) has the value "123.6".

SUBSTR (*string*, *start*) [Function]

Returns a string consisting of the value of *string* from position *start* onward. Returns an empty string if *start* is system-missing, less than 1, or greater than the length of *string*.

SUBSTR (*string*, *start*, *count*) [Function]

Returns a string consisting of the first *count* characters from *string* beginning at position *start*. Returns an empty string if *start* or *count* is system-missing, if *start* is less than 1 or greater than the number of characters in *string*, or if *count* is less than 1. Returns a string shorter than *count* characters if *start* + *count* - 1 is greater than the number of characters in *string*. Examples: SUBSTR("abcdefg", 3, 2) has value "cd"; SUBSTR("nonsense", 4, 10) has the value "sense".

UPCASE (*string*) [Function]

Returns *string*, changing lowercase letters to uppercase letters.

5.7.8 Time & Date Functions

For compatibility, PSPP considers dates before 15 Oct 1582 invalid. Most time and date functions will not accept earlier dates.

5.7.8.1 How times & dates are defined and represented

Times and dates are handled by PSPP as single numbers. A *time* is an interval. PSPP measures times in seconds. Thus, the following intervals correspond with the numeric values given:

10 minutes	600
1 hour	3,600
1 day, 3 hours, 10 seconds	97,210
40 days	3,456,000

A *date*, on the other hand, is a particular instant in the past or the future. PSPP represents a date as a number of seconds since midnight preceding 14 Oct 1582. Because midnight preceding the dates given below correspond with the numeric PSPP dates given:

15 Oct 1582	86,400
4 Jul 1776	6,113,318,400
1 Jan 1900	10,010,390,400
1 Oct 1978	12,495,427,200
24 Aug 1995	13,028,601,600

Ordinary arithmetic operations on dates and times often produce sensible results. Adding a time to, or subtracting one from, a date produces a new date that much earlier or later. The difference of two dates yields the time between those dates. Adding two times produces the combined time. Multiplying a time by a scalar produces a time that many times longer. Since times and dates are just numbers, the ordinary addition and subtraction operators are employed for these purposes.

Adding two dates does not produce a useful result.

As the table shows, dates and times may have very large values. Thus, it is not a good idea to take powers of these values; also, the accuracy of some procedures may be affected. If necessary, convert times or dates in seconds to some other unit, like days or years, before performing analysis.

5.7.8.2 Functions that Produce Times

These functions take numeric arguments and return numeric values that represent times.

TIME.DAYS (*ndays*) [Function]
Returns a time corresponding to *ndays* days.

TIME.HMS (*nhours*, *nmins*, *nsecs*) [Function]
Returns a time corresponding to *nhours* hours, *nmins* minutes, and *nsecs* seconds. The arguments may not have mixed signs: if any of them are positive, then none may be negative, and vice versa.

5.7.8.3 Functions that Examine Times

These functions take numeric arguments in PSPP time format and give numeric results.

CTIME.DAYS (*time*) [Function]
Results in the number of days and fractional days in *time*.

CTIME.HOURS (*time*) [Function]
Results in the number of hours and fractional hours in *time*.

CTIME.MINUTES (*time*) [Function]
Results in the number of minutes and fractional minutes in *time*.

CTIME.SECONDS (*time*) [Function]
Results in the number of seconds and fractional seconds in *time*. (**CTIME.SECONDS** does nothing; **CTIME.SECONDS**(*x*) is equivalent to *x*.)

5.7.8.4 Functions that Produce Dates

These functions take numeric arguments and give numeric results that represent dates. Arguments taken by these functions are:

<i>day</i>	Refers to a day of the month between 1 and 31. Day 0 is also accepted and refers to the final day of the previous month. Days 29, 30, and 31 are accepted even in months that have fewer days and refer to a day near the beginning of the following month.
<i>month</i>	Refers to a month of the year between 1 and 12. Months 0 and 13 are also accepted and refer to the last month of the preceding year and the first month of the following year, respectively.
<i>quarter</i>	Refers to a quarter of the year between 1 and 4. The quarters of the year begin on the first day of months 1, 4, 7, and 10.
<i>week</i>	Refers to a week of the year between 1 and 53.
<i>yday</i>	Refers to a day of the year between 1 and 366.
<i>year</i>	Refers to a year, 1582 or greater. Years between 0 and 99 are treated according to the epoch set on SET EPOCH, by default beginning 69 years before the current date (see [SET EPOCH] , page 93).

If these functions' arguments are out-of-range, they are correctly normalized before conversion to date format. Non-integers are rounded toward zero.

DATE.DMY (*day*, *month*, *year*) [Function]

DATE.MDY (*month*, *day*, *year*) [Function]

Results in a date value corresponding to the midnight before day *day* of month *month* of year *year*.

DATE.MOYR (*month*, *year*) [Function]

Results in a date value corresponding to the midnight before the first day of month *month* of year *year*.

DATE.QYR (*quarter*, *year*) [Function]

Results in a date value corresponding to the midnight before the first day of quarter *quarter* of year *year*.

DATE.WKYR (*week*, *year*) [Function]

Results in a date value corresponding to the midnight before the first day of week *week* of year *year*.

DATE.YRDAY (*year*, *yday*) [Function]

Results in a date value corresponding to the day *yday* of year *year*.

5.7.8.5 Functions that Examine Dates

These functions take numeric arguments in PSPP date or time format and give numeric results. These names are used for arguments:

date A numeric value in PSPP date format.

time A numeric value in PSPP time format.

time-or-date

A numeric value in PSPP time or date format.

XDATE.DATE (*time-or-date*) [Function]

For a time, results in the time corresponding to the number of whole days *date-or-time* includes. For a date, results in the date corresponding to the latest midnight at or before *date-or-time*; that is, gives the date that *date-or-time* is in.

XDATE.HOUR (*time-or-date*) [Function]

For a time, results in the number of whole hours beyond the number of whole days represented by *date-or-time*. For a date, results in the hour (as an integer between 0 and 23) corresponding to *date-or-time*.

XDATE.JDAY (*date*) [Function]

Results in the day of the year (as an integer between 1 and 366) corresponding to *date*.

XDATE.MDAY (*date*) [Function]

Results in the day of the month (as an integer between 1 and 31) corresponding to *date*.

XDATE.MINUTE (*time-or-date*) [Function]

Results in the number of minutes (as an integer between 0 and 59) after the last hour in *time-or-date*.

XDATE.MONTH (*date*) [Function]
Results in the month of the year (as an integer between 1 and 12) corresponding to *date*.

XDATE.QUARTER (*date*) [Function]
Results in the quarter of the year (as an integer between 1 and 4) corresponding to *date*.

XDATE.SECOND (*time-or-date*) [Function]
Results in the number of whole seconds after the last whole minute (as an integer between 0 and 59) in *time-or-date*.

XDATE.TDAY (*date*) [Function]
Results in the number of whole days from 14 Oct 1582 to *date*.

XDATE.TIME (*date*) [Function]
Results in the time of day at the instant corresponding to *date*, as a time value. This is the number of seconds since midnight on the day corresponding to *date*.

XDATE.WEEK (*date*) [Function]
Results in the week of the year (as an integer between 1 and 53) corresponding to *date*.

XDATE.WKDAY (*date*) [Function]
Results in the day of week (as an integer between 1 and 7) corresponding to *date*, where 1 represents Sunday.

XDATE.YEAR (*date*) [Function]
Returns the year (as an integer 1582 or greater) corresponding to *date*.

5.7.9 Miscellaneous Functions

Miscellaneous functions take various arguments and produce various results.

LAG (*variable* [, *ncases*]) [Function]
variable must be a numeric or string variable name. **LAG** results in the value of that variable for the case *ncases* before the current one. In case-selection procedures, **LAG** results in the value of the variable for the last case selected. Results in system-missing (for numeric variables) or blanks (for string variables) for the first case or before any cases are selected.

If omitted, *ncases* defaults to 1. Otherwise, *ncases* must be a small positive constant integer. There is no explicit limit, but use of a large value will increase memory consumption.

YRMODA (*year*, *month*, *day*) [Function]
year is a year, either between 0 and 99 or at least 1582. Unlike other PSPP date functions, years between 0 and 99 always correspond to 1900 through 1999. *month* is a month between 1 and 13. *day* is a day between 0 and 31. A *day* of 0 refers to the last day of the previous month, and a *month* of 13 refers to the first month of the next year. *year* must be in range. *year*, *month*, and *day* must all be integers.

YRMODA results in the number of days between 15 Oct 1582 and the date specified, plus one. The date passed to **YRMODA** must be on or after 15 Oct 1582. 15 Oct 1582 has a value of 1.

5.7.10 Statistical Distribution Functions

PSPP can calculate several functions of standard statistical distributions. These functions are named systematically based on the function and the distribution. The table below describes the statistical distribution functions in general:

PDF.dist (*x* [, *param. . .*])

Probability density function for *dist*. The domain of *x* depends on *dist*. For continuous distributions, the result is the density of the probability function at *x*, and the range is nonnegative real numbers. For discrete distributions, the result is the probability of *x*.

CDF.dist (*x* [, *param. . .*])

Cumulative distribution function for *dist*, that is, the probability that a random variate drawn from the distribution is less than *x*. The domain of *x* depends *dist*. The result is a probability.

SIG.dist (*x* [, *param. . .*])

Tail probability function for *dist*, that is, the probability that a random variate drawn from the distribution is greater than *x*. The domain of *x* depends *dist*. The result is a probability. Only a few distributions include an SIG function.

IDF.dist (*p* [, *param. . .*])

Inverse distribution function for *dist*, the value of *x* for which the CDF would yield *p*. The value of *p* is a probability. The range depends on *dist* and is identical to the domain for the corresponding CDF.

RV.dist ([*param. . .*])

Random variate function for *dist*. The range depends on the distribution.

NPDF.dist (*x* [, *param. . .*])

Noncentral probability density function. The result is the density of the given noncentral distribution at *x*. The domain of *x* depends on *dist*. The range is nonnegative real numbers. Only a few distributions include an NPDF function.

NCDF.dist (*x* [, *param. . .*])

Noncentral cumulative distribution function for *dist*, that is, the probability that a random variate drawn from the given noncentral distribution is less than *x*. The domain of *x* depends *dist*. The result is a probability. Only a few distributions include an NCDF function.

The individual distributions are described individually below.

5.7.10.1 Continuous Distributions

The following continuous distributions are available:

PDF.BETA (*x*)

[Function]

CDF.BETA (*x*, *a*, *b*)

[Function]

IDF.BETA (*p*, *a*, *b*) [Function]

RV.BETA (*a*, *b*) [Function]

NPDF.BETA (*x*, *a*, *b*, *lambda*) [Function]

NCDF.BETA (*x*, *a*, *b*, *lambda*) [Function]

Beta distribution with shape parameters *a* and *b*. The noncentral distribution takes an additional parameter *lambda*. Constraints: $a > 0$, $b > 0$, $lambda \geq 0$, $0 \leq x \leq 1$, $0 \leq p \leq 1$.

PDF.BVNOR (*x0*, *x1*, *rho*) [Function]

CDF.BVNOR (*x0*, *x1*, *rho*) [Function]

Bivariate normal distribution of two standard normal variables with correlation coefficient *rho*. Two variates *x0* and *x1* must be provided. Constraints: $0 \leq rho \leq 1$, $0 \leq p \leq 1$.

PDF.CAUCHY (*x*, *a*, *b*) [Function]

CDF.CAUCHY (*x*, *a*, *b*) [Function]

IDF.CAUCHY (*p*, *a*, *b*) [Function]

RV.CAUCHY (*a*, *b*) [Function]

Cauchy distribution with location parameter *a* and scale parameter *b*. Constraints: $b > 0$, $0 < p < 1$.

PDF.CHISQ (*x*, *df*) [Function]

CDF.CHISQ (*x*, *df*) [Function]

SIG.CHISQ (*x*, *df*) [Function]

IDF.CHISQ (*p*, *df*) [Function]

RV.CHISQ (*df*) [Function]

NPDF.CHISQ (*x*, *df*, *lambda*) [Function]

NCDF.CHISQ (*x*, *df*, *lambda*) [Function]

Chi-squared distribution with *df* degrees of freedom. The noncentral distribution takes an additional parameter *lambda*. Constraints: $df > 0$, $lambda > 0$, $x \geq 0$, $0 \leq p < 1$.

PDF.EXP (*x*, *a*) [Function]

CDF.EXP (*x*, *a*) [Function]

IDF.EXP (*p*, *a*) [Function]

RV.EXP (*a*) [Function]

Exponential distribution with scale parameter *a*. The inverse of *a* represents the rate of decay. Constraints: $a > 0$, $x \geq 0$, $0 \leq p < 1$.

PDF.XPOWER (*x*, *a*, *b*) [Function]

RV.XPOWER (*a*, *b*) [Function]

Exponential power distribution with positive scale parameter *a* and nonnegative power parameter *b*. Constraints: $a > 0$, $b \geq 0$, $x \geq 0$, $0 \leq p \leq 1$. This distribution is a PSPP extension.

PDF.F (*x*, *df1*, *df2*) [Function]

CDF.F (*x*, *df1*, *df2*) [Function]

SIG.F (*x*, *df1*, *df2*) [Function]

IDF.F (*p*, *df1*, *df2*) [Function]

RV.F (*df1*, *df2*) [Function]

NPDF.F (*x*, *df1*, *df2*, *lambda*) [Function]

NCDF.F (*x*, *df1*, *df2*, *lambda*) [Function]

F-distribution of two chi-squared deviates with *df1* and *df2* degrees of freedom. The noncentral distribution takes an additional parameter *lambda*. Constraints: *df1* > 0, *df2* > 0, *lambda* >= 0, *x* >= 0, 0 <= *p* < 1.

PDF.GAMMA (*x*, *a*, *b*) [Function]

CDF.GAMMA (*x*, *a*, *b*) [Function]

IDF.GAMMA (*p*, *a*, *b*) [Function]

RV.GAMMA (*a*, *b*) [Function]

Gamma distribution with shape parameter *a* and scale parameter *b*. Constraints: *a* > 0, *b* > 0, *x* >= 0, 0 <= *p* < 1.

PDF.HALFNRM (*x*, *a*, *b*) [Function]

CDF.HALFNRM (*x*, *a*, *b*) [Function]

IDF.HALFNRM (*p*, *a*, *b*) [Function]

RV.HALFNRM (*a*, *b*) [Function]

Half-normal distribution with location parameter *a* and shape parameter *b*. Constraints: *b* > 0, 0 < *p* < 1.

PDF.IGAUSS (*x*, *a*, *b*) [Function]

CDF.IGAUSS (*x*, *a*, *b*) [Function]

IDF.IGAUSS (*p*, *a*, *b*) [Function]

RV.IGAUSS (*a*, *b*) [Function]

Inverse Gaussian distribution with parameters *a* and *b*. Constraints: *a* > 0, *b* > 0, *x* > 0, 0 <= *p* < 1.

PDF.LANDAU (*x*) [Function]

RV.LANDAU () [Function]

Landau distribution.

PDF.LAPLACE (*x*, *a*, *b*) [Function]

CDF.LAPLACE (*x*, *a*, *b*) [Function]

IDF.LAPLACE (*p*, *a*, *b*) [Function]

RV.LAPLACE (*a*, *b*) [Function]

Laplace distribution with location parameter *a* and scale parameter *b*. Constraints: *b* > 0, 0 < *p* < 1.

RV.LEVY (*c*, *alpha*) [Function]

Levy symmetric alpha-stable distribution with scale *c* and exponent *alpha*. Constraints: 0 < *alpha* <= 2.

RV.LVSKEW (*c*, *alpha*, *beta*) [Function]

Levy skew alpha-stable distribution with scale *c*, exponent *alpha*, and skewness parameter *beta*. Constraints: 0 < *alpha* <= 2, -1 <= *beta* <= 1.

PDF.LOGISTIC (*x*, *a*, *b*) [Function]

CDF.LOGISTIC (*x*, *a*, *b*) [Function]

IDF.LOGISTIC (*p*, *a*, *b*) [Function]

RV.LOGISTIC (*a*, *b*) [Function]

Logistic distribution with location parameter *a* and scale parameter *b*. Constraints: $b > 0$, $0 < p < 1$.

PDF.LNORMAL (*x*, *a*, *b*) [Function]

CDF.LNORMAL (*x*, *a*, *b*) [Function]

IDF.LNORMAL (*p*, *a*, *b*) [Function]

RV.LNORMAL (*a*, *b*) [Function]

Lognormal distribution with parameters *a* and *b*. Constraints: $a > 0$, $b > 0$, $x \geq 0$, $0 \leq p < 1$.

PDF.NORMAL (*x*, *mu*, *sigma*) [Function]

CDF.NORMAL (*x*, *mu*, *sigma*) [Function]

IDF.NORMAL (*p*, *mu*, *sigma*) [Function]

RV.NORMAL (*mu*, *sigma*) [Function]

Normal distribution with mean *mu* and standard deviation *sigma*. Constraints: $b > 0$, $0 < p < 1$. Three additional functions are available as shorthand:

CDFNORM (*x*) [Function]

Equivalent to CDF.NORMAL(*x*, 0, 1).

PROBIT (*p*) [Function]

Equivalent to IDF.NORMAL(*p*, 0, 1).

NORMAL (*sigma*) [Function]

Equivalent to RV.NORMAL(0, *sigma*).

PDF.NTAIL (*x*, *a*, *sigma*) [Function]

RV.NTAIL (*a*, *sigma*) [Function]

Normal tail distribution with lower limit *a* and standard deviation *sigma*. This distribution is a PSPP extension. Constraints: $a > 0$, $x > a$, $0 < p < 1$.

PDF.PARETO (*x*, *a*, *b*) [Function]

CDF.PARETO (*x*, *a*, *b*) [Function]

IDF.PARETO (*p*, *a*, *b*) [Function]

RV.PARETO (*a*, *b*) [Function]

Pareto distribution with threshold parameter *a* and shape parameter *b*. Constraints: $a > 0$, $b > 0$, $x \geq a$, $0 \leq p < 1$.

PDF.RAYLEIGH (*x*, *sigma*) [Function]

CDF.RAYLEIGH (*x*, *sigma*) [Function]

IDF.RAYLEIGH (*p*, *sigma*) [Function]

RV.RAYLEIGH (*sigma*) [Function]

Rayleigh distribution with scale parameter *sigma*. This distribution is a PSPP extension. Constraints: $sigma > 0$, $x > 0$.

PDF.RTAIL (*x*, *a*, *sigma*) [Function]

RV. RTAIL (*a*, *sigma*) [Function]

Rayleigh tail distribution with lower limit *a* and scale parameter *sigma*. This distribution is a PSPP extension. Constraints: $a > 0$, $sigma > 0$, $x > a$.

CDF.SMOD (*x*, *a*, *b*) [Function]

IDF.SMOD (*p*, *a*, *b*) [Function]

Studentized maximum modulus distribution with parameters *a* and *b*. Constraints: $a > 0$, $b > 0$, $x > 0$, $0 \leq p < 1$.

CDF.SRANGE (*x*, *a*, *b*) [Function]

IDF.SRANGE (*p*, *a*, *b*) [Function]

Studentized range distribution with parameters *a* and *b*. Constraints: $a \geq 1$, $b \geq 1$, $x > 0$, $0 \leq p < 1$.

PDF.T (*x*, *df*) [Function]

CDF.T (*x*, *df*) [Function]

IDF.T (*p*, *df*) [Function]

RV.T (*df*) [Function]

NPDF.T (*x*, *df*, *lambda*) [Function]

NCDF.T (*x*, *df*, *lambda*) [Function]

T-distribution with *df* degrees of freedom. The noncentral distribution takes an additional parameter *lambda*. Constraints: $df > 0$, $0 < p < 1$.

PDF.T1G (*x*, *a*, *b*) [Function]

CDF.T1G (*x*, *a*, *b*) [Function]

IDF.T1G (*p*, *a*, *b*) [Function]

Type-1 Gumbel distribution with parameters *a* and *b*. This distribution is a PSPP extension. Constraints: $0 < p < 1$.

PDF.T2G (*x*, *a*, *b*) [Function]

CDF.T2G (*x*, *a*, *b*) [Function]

IDF.T2G (*p*, *a*, *b*) [Function]

Type-2 Gumbel distribution with parameters *a* and *b*. This distribution is a PSPP extension. Constraints: $x > 0$, $0 < p < 1$.

PDF.UNIFORM (*x*, *a*, *b*) [Function]

CDF.UNIFORM (*x*, *a*, *b*) [Function]

IDF.UNIFORM (*p*, *a*, *b*) [Function]

RV.UNIFORM (*a*, *b*) [Function]

Uniform distribution with parameters *a* and *b*. Constraints: $a \leq x \leq b$, $0 \leq p \leq 1$. An additional function is available as shorthand:

UNIFORM (*b*) [Function]

Equivalent to RV.UNIFORM(0, *b*).

PDF.WEIBULL (*x*, *a*, *b*) [Function]

CDF.WEIBULL (*x*, *a*, *b*) [Function]

IDF.WEIBULL (*p*, *a*, *b*) [Function]

RV.WEIBULL (*a*, *b*) [Function]

Weibull distribution with parameters *a* and *b*. Constraints: $a > 0$, $b > 0$, $x \geq 0$, $0 \leq p < 1$.

5.7.10.2 Discrete Distributions

The following discrete distributions are available:

PDF.BERNOULLI (x) [Function]
 CDF.BERNOULLI (x, p) [Function]
 RV.BERNOULLI (p) [Function]

Bernoulli distribution with probability of success p . Constraints: $x = 0$ or 1 , $0 \leq p \leq 1$.

PDF.BINOMIAL (x, n, p) [Function]
 CDF.BINOMIAL (x, n, p) [Function]
 RV.BINOMIAL (n, p) [Function]

Binomial distribution with n trials and probability of success p . Constraints: integer $n > 0$, $0 \leq p \leq 1$, integer $x \leq n$.

PDF.GEOM (x, n, p) [Function]
 CDF.GEOM (x, n, p) [Function]
 RV.GEOM (n, p) [Function]

Geometric distribution with probability of success p . Constraints: $0 \leq p \leq 1$, integer $x > 0$.

PDF.HYPER (x, a, b, c) [Function]
 CDF.HYPER (x, a, b, c) [Function]
 RV.HYPER (a, b, c) [Function]

Hypergeometric distribution when b objects out of a are drawn and c of the available objects are distinctive. Constraints: integer $a > 0$, integer $b \leq a$, integer $c \leq a$, integer $x \geq 0$.

PDF.LOG (x, p) [Function]
 RV.LOG (p) [Function]

Logarithmic distribution with probability parameter p . Constraints: $0 \leq p < 1$, $x \geq 1$.

PDF.NEGBIN (x, n, p) [Function]
 CDF.NEGBIN (x, n, p) [Function]
 RV.NEGBIN (n, p) [Function]

Negative binomial distribution with number of successes parameter n and probability of success parameter p . Constraints: integer $n \geq 0$, $0 < p \leq 1$, integer $x \geq 1$.

PDF.POISSON (x, mu) [Function]
 CDF.POISSON (x, mu) [Function]
 RV.POISSON (mu) [Function]

Poisson distribution with mean mu . Constraints: $mu > 0$, integer $x \geq 0$.

5.8 Operator Precedence

The following table describes operator precedence. Smaller-numbered levels in the table have higher precedence. Within a level, operations are always performed from left to right. The first occurrence of ‘-’ represents unary negation, the second binary subtraction.

1. ()
2. **
3. -
4. * /
5. + -
6. EQ GE GT LE LT NE
7. AND NOT OR

6 Data Input and Output

Data are the focus of the PSPP language. Each datum belongs to a *case* (also called an *observation*). Each case represents an individual or ‘experimental unit’. For example, in the results of a survey, the names of the respondents, their sex, age *etc.* and their responses are all data and the data pertaining to single respondent is a case. This chapter examines the PSPP commands for defining variables and reading and writing data.

Please note: Data is not actually read until a procedure is executed. These commands tell PSPP how to read data, but they do not *cause* PSPP to read data.

6.1 BEGIN DATA

BEGIN DATA.

...

END DATA.

BEGIN DATA and END DATA can be used to embed raw ASCII data in a PSPP syntax file. DATA LIST or another input procedure must be used before BEGIN DATA (see [Section 6.3 \[DATA LIST\]](#), [page 39](#)). BEGIN DATA and END DATA must be used together. END DATA must appear by itself on a single line, with no leading white space and exactly one space between the words END and DATA, like this:

END DATA.

6.2 CLEAR TRANSFORMATIONS

CLEAR TRANSFORMATIONS.

CLEAR TRANSFORMATIONS clears out all pending transformations. It does not cancel the current input program. It is valid only when PSPP is interactive, not in syntax files.

6.3 DATA LIST

Used to read text or binary data, DATA LIST is the most fundamental data-reading command. Even the more sophisticated input methods use DATA LIST commands as a building block. Understanding DATA LIST is important to understanding how to use PSPP to read your data files.

There are two major variants of DATA LIST, which are fixed format and free format. In addition, free format has a minor variant, list format, which is discussed in terms of its differences from vanilla free format.

Each form of DATA LIST is described in detail below.

6.3.1 DATA LIST FIXED

```
DATA LIST [FIXED]
      {TABLE,NOTABLE}
      FILE='filename'
      RECORDS=record_count
      END=end_var
```

```
/[line_no] var_spec...
```

where each `var_spec` takes one of the forms

```
var_list start-end [type_spec]
```

```
var_list (fortran_spec)
```

DATA LIST FIXED is used to read data files that have values at fixed positions on each line of single-line or multiline records. The keyword FIXED is optional.

The FILE subcommand must be used if input is to be taken from an external file. It may be used to specify a filename as a string or a file handle (see [Section 6.6 \[FILE HANDLE\]](#), [page 43](#)). If the FILE subcommand is not used, then input is assumed to be specified within the command file using BEGIN DATA...END DATA (see [Section 6.1 \[BEGIN DATA\]](#), [page 39](#)).

The optional RECORDS subcommand, which takes a single integer as an argument, is used to specify the number of lines per record. If RECORDS is not specified, then the number of lines per record is calculated from the list of variable specifications later in DATA LIST.

The END subcommand is only useful in conjunction with INPUT PROGRAM. See [Section 6.7 \[INPUT PROGRAM\]](#), [page 44](#), for details.

DATA LIST can optionally output a table describing how the data file will be read. The TABLE subcommand enables this output, and NOTABLE disables it. The default is to output the table.

The list of variables to be read from the data list must come last. Each line in the data record is introduced by a slash ('/'). Optionally, a line number may follow the slash. Following, any number of variable specifications may be present.

Each variable specification consists of a list of variable names followed by a description of their location on the input line. Sets of variables may be specified using the DATA LIST TO convention (see [Section 4.6.3 \[Sets of Variables\]](#), [page 14](#)). There are two ways to specify the location of the variable on the line: columnar style and FORTRAN style.

In columnar style, the starting column and ending column for the field are specified after the variable name, separated by a dash ('-'). For instance, the third through fifth columns on a line would be specified '3-5'. By default, variables are considered to be in 'F' format (see [Section 4.6.4 \[Input/Output Formats\]](#), [page 14](#)). (This default can be changed; see [Section 13.15 \[SET\]](#), [page 91](#) for more information.)

In columnar style, to use a variable format other than the default, specify the format type in parentheses after the column numbers. For instance, for alphanumeric 'A' format, use '(A)'.

In addition, implied decimal places can be specified in parentheses after the column numbers. As an example, suppose that a data file has a field in which the characters '1234' should be interpreted as having the value 12.34. Then this field has two implied decimal places, and the corresponding specification would be '(2)'. If a field that has implied decimal places contains a decimal point, then the implied decimal places are not applied.

Changing the variable format and adding implied decimal places can be done together; for instance, '(N,5)'.

When using columnar style, the input and output width of each variable is computed from the field width. The field width must be evenly divisible into the number of variables specified.

FORTTRAN style is an altogether different approach to specifying field locations. With this approach, a list of variable input format specifications, separated by commas, are placed after the variable names inside parentheses. Each format specifier advances as many characters into the input line as it uses.

Implied decimal places also exist in FORTRAN style. A format specification with d decimal places also has d implied decimal places.

In addition to the standard format specifiers (see [Section 4.6.4 \[Input/Output Formats\]](#), [page 14](#)), FORTRAN style defines some extensions:

- X** Advance the current column on this line by one character position.
- Tx** Set the current column on this line to column x , with column numbers considered to begin with 1 at the left margin.
- NEWRECx** Skip forward x lines in the current record, resetting the active column to the left margin.

Repeat count

Any format specifier may be preceded by a number. This causes the action of that format specifier to be repeated the specified number of times.

(spec1, ..., specN)

Group the given specifiers together. This is most useful when preceded by a repeat count. Groups may be nested arbitrarily.

FORTTRAN and columnar styles may be freely intermixed. Columnar style leaves the active column immediately after the ending column specified. Record motion using **NEWREC** in FORTRAN style also applies to later FORTRAN and columnar specifiers.

Examples

1.

```
DATA LIST TABLE /NAME 1-10 (A) INF01 TO INF03 12-17 (1).
```

```
BEGIN DATA.
```

```
John Smith 102311
```

```
Bob Arnold 122015
```

```
Bill Yates 918 6
```

```
END DATA.
```

Defines the following variables:

- **NAME**, a 10-character-wide long string variable, in columns 1 through 10.
- **INF01**, a numeric variable, in columns 12 through 13.
- **INF02**, a numeric variable, in columns 14 through 15.
- **INF03**, a numeric variable, in columns 16 through 17.

The **BEGIN DATA/END DATA** commands cause three cases to be defined:

Case	NAME	INF01	INF02	INF03
1	John Smith	10	23	11
2	Bob Arnold	12	20	15
3	Bill Yates	9	18	6

The `TABLE` keyword causes PSPP to print out a table describing the four variables defined.

2.

```
DAT LIS FIL="survey.dat"
      /ID 1-5 NAME 7-36 (A) SURNAME 38-67 (A) MINITIAL 69 (A)
      /Q01 TO Q50 7-56
      /.
```

Defines the following variables:

- `ID`, a numeric variable, in columns 1-5 of the first record.
- `NAME`, a 30-character long string variable, in columns 7-36 of the first record.
- `SURNAME`, a 30-character long string variable, in columns 38-67 of the first record.
- `MINITIAL`, a 1-character short string variable, in column 69 of the first record.
- Fifty variables `Q01`, `Q02`, `Q03`, . . . , `Q49`, `Q50`, all numeric, `Q01` in column 7, `Q02` in column 8, . . . , `Q49` in column 55, `Q50` in column 56, all in the second record.

Cases are separated by a blank record.

Data is read from file ‘`survey.dat`’ in the current directory.

This example shows keywords abbreviated to their first 3 letters.

6.3.2 DATA LIST FREE

```
DATA LIST FREE
      [({TAB,'c'}, . . .)]
      [{NOTABLE, TABLE}]
      FILE='filename'
      END=end_var
      /var_spec . .
```

where each `var_spec` takes one of the forms

```
var_list [(type_spec)]
var_list *
```

In free format, the input data is, by default, structured as a series of fields separated by spaces, tabs, commas, or line breaks. Each field’s content may be unquoted, or it may be quoted with a pairs of apostrophes (‘’) or double quotes (“”). Unquoted white space separates fields but is not part of any field. Any mix of spaces, tabs, and line breaks is equivalent to a single space for the purpose of separating fields, but consecutive commas will skip a field.

Alternatively, delimiters can be specified explicitly, as a parenthesized, comma-separated list of single-character strings immediately following `FREE`. The word `TAB` may also be used to specify a tab character as a delimiter. When delimiters are specified explicitly, only the given characters, plus line breaks, separate fields. Furthermore, leading spaces at the

beginnings of fields are not trimmed, consecutive delimiters define empty fields, and no form of quoting is allowed.

The NOTABLE and TABLE subcommands are as in DATA LIST FIXED above. NO-TABLE is the default.

The FILE and END subcommands are as in DATA LIST FIXED above.

The variables to be parsed are given as a single list of variable names. This list must be introduced by a single slash ('/'). The set of variable names may contain format specifications in parentheses (see [Section 4.6.4 \[Input/Output Formats\]](#), page 14). Format specifications apply to all variables back to the previous parenthesized format specification.

In addition, an asterisk may be used to indicate that all variables preceding it are to have input/output format 'F8.0'.

Specified field widths are ignored on input, although all normal limits on field width apply, but they are honored on output.

6.3.3 DATA LIST LIST

```
DATA LIST LIST
  [({TAB,'c'}, ...)]
  [{NOTABLE, TABLE}]
  FILE='filename'
  END=end_var
  /var_spec. . .
```

where each var_spec takes one of the forms

```
var_list [(type_spec)]
var_list *
```

With one exception, DATA LIST LIST is syntactically and semantically equivalent to DATA LIST FREE. The exception is that each input line is expected to correspond to exactly one input record. If more or fewer fields are found on an input line than expected, an appropriate diagnostic is issued.

6.4 END CASE

END CASE.

END CASE is used only within INPUT PROGRAM to output the current case. See [Section 6.7 \[INPUT PROGRAM\]](#), page 44, for details.

6.5 END FILE

END FILE.

END FILE is used only within INPUT PROGRAM to terminate the current input program. See [Section 6.7 \[INPUT PROGRAM\]](#), page 44.

6.6 FILE HANDLE

```
FILE HANDLE handle_name
  /NAME='filename'
```

```

/MODE={CHARACTER,IMAGE}
/LRECL=rec_len
/TABWIDTH=tab_width

```

Use FILE HANDLE to associate a file handle name with a file and its attributes, so that later commands can refer to the file by its handle name. Because names of text files can be specified directly on commands that access files, FILE HANDLE is only needed when a file is not an ordinary file containing lines of text. However, FILE HANDLE may be used even for text files, and it may be easier to specify a file's name once and later refer to it by an abstract handle.

Specify the file handle name as an identifier. Any given identifier may only appear once in a PSPP run. File handles may not be reassigned to a different file. The file handle name must immediately follow the FILE HANDLE command name.

The NAME subcommand specifies the name of the file associated with the handle. It is the only required subcommand.

MODE specifies a file mode. In CHARACTER mode, the default, the data file is opened in ANSI C text mode, so that local end of line conventions are followed, and each text line is read as one record. In CHARACTER mode, most input programs will expand tabs to spaces (DATA LIST FREE with explicitly specified delimiters is an exception). By default, each tab is 4 characters wide, but an alternate width may be specified on TABWIDTH. A tab width of 0 suppresses tab expansion entirely.

By contrast, in BINARY mode, the data file is opened in ANSI C binary mode and records are a fixed length. In BINARY mode, LRECL specifies the record length in bytes, with a default of 1024. Tab characters are never expanded to spaces in binary mode.

6.7 INPUT PROGRAM

```

INPUT PROGRAM.
... input commands ...
END INPUT PROGRAM.

```

INPUT PROGRAM...END INPUT PROGRAM specifies a complex input program. By placing data input commands within INPUT PROGRAM, PSPP programs can take advantage of more complex file structures than available with only DATA LIST.

The first sort of extended input program is to simply put multiple DATA LIST commands within the INPUT PROGRAM. This will cause all of the data files to be read in parallel. Input will stop when end of file is reached on any of the data files.

Transformations, such as conditional and looping constructs, can also be included within INPUT PROGRAM. These can be used to combine input from several data files in more complex ways. However, input will still stop when end of file is reached on any of the data files.

To prevent INPUT PROGRAM from terminating at the first end of file, use the END subcommand on DATA LIST. This subcommand takes a variable name, which should be a numeric scratch variable (see [Section 4.6.5 \[Scratch Variables\]](#), page 19). (It need not be a scratch variable but otherwise the results can be surprising.) The value of this variable is set to 0 when reading the data file, or 1 when end of file is encountered.

Two additional commands are useful in conjunction with INPUT PROGRAM. END CASE is the first. Normally each loop through the INPUT PROGRAM structure produces one case. END CASE controls exactly when cases are output. When END CASE is used, looping from the end of INPUT PROGRAM to the beginning does not cause a case to be output.

END FILE is the second. When the END subcommand is used on DATA LIST, there is no way for the INPUT PROGRAM construct to stop looping, so an infinite loop results. END FILE, when executed, stops the flow of input data and passes out of the INPUT PROGRAM structure.

All this is very confusing. A few examples should help to clarify.

```
INPUT PROGRAM.
    DATA LIST NOTABLE FILE='a.data'/X 1-10.
    DATA LIST NOTABLE FILE='b.data'/Y 1-10.
END INPUT PROGRAM.
LIST.
```

The example above reads variable X from file 'a.data' and variable Y from file 'b.data'. If one file is shorter than the other then the extra data in the longer file is ignored.

```
INPUT PROGRAM.
    NUMERIC #A #B.

    DO IF NOT #A.
        DATA LIST NOTABLE END=#A FILE='a.data'/X 1-10.
    END IF.
    DO IF NOT #B.
        DATA LIST NOTABLE END=#B FILE='b.data'/Y 1-10.
    END IF.
    DO IF #A AND #B.
        END FILE.
    END IF.
    END CASE.
END INPUT PROGRAM.
LIST.
```

The above example reads variable X from 'a.data' and variable Y from 'b.data'. If one file is shorter than the other then the missing field is set to the system-missing value alongside the present value for the remaining length of the longer file.

```
INPUT PROGRAM.
    NUMERIC #A #B.

    DO IF #A.
        DATA LIST NOTABLE END=#B FILE='b.data'/X 1-10.
    DO IF #B.
        END FILE.
    ELSE.
        END CASE.
    END IF.
```

```

ELSE.
    DATA LIST NOTABLE END=#A FILE='a.data'/X 1-10.
    DO IF NOT #A.
        END CASE.
    END IF.
END IF.
END INPUT PROGRAM.
LIST.

```

The above example reads data from file 'a.data', then from 'b.data', and concatenates them into a single active file.

```

INPUT PROGRAM.
    NUMERIC #EOF.

    LOOP IF NOT #EOF.
        DATA LIST NOTABLE END=#EOF FILE='a.data'/X 1-10.
        DO IF NOT #EOF.
            END CASE.
        END IF.
    END LOOP.

    COMPUTE #EOF = 0.
    LOOP IF NOT #EOF.
        DATA LIST NOTABLE END=#EOF FILE='b.data'/X 1-10.
        DO IF NOT #EOF.
            END CASE.
        END IF.
    END LOOP.

END FILE.
END INPUT PROGRAM.
LIST.

```

The above example does the same thing as the previous example, in a different way.

```

INPUT PROGRAM.
    LOOP #I=1 TO 50.
        COMPUTE X=UNIFORM(10).
        END CASE.
    END LOOP.
END FILE.
END INPUT PROGRAM.
LIST/FORMAT=NUMBERED.

```

The above example causes an active file to be created consisting of 50 random variates between 0 and 10.

6.8 LIST

```
LIST
```

```

/VARIABLES=var_list
/CASES=FROM start_index TO end_index BY incr_index
/FORMAT={UNNUMBERED,NUMBERED} {WRAP,SINGLE}
        {NOWEIGHT,WEIGHT}

```

The LIST procedure prints the values of specified variables to the listing file.

The VARIABLES subcommand specifies the variables whose values are to be printed. Keyword VARIABLES is optional. If VARIABLES subcommand is not specified then all variables in the active file are printed.

The CASES subcommand can be used to specify a subset of cases to be printed. Specify FROM and the case number of the first case to print, TO and the case number of the last case to print, and BY and the number of cases to advance between printing cases, or any subset of those settings. If CASES is not specified then all cases are printed.

The FORMAT subcommand can be used to change the output format. NUMBERED will print case numbers along with each case; UNNUMBERED, the default, causes the case numbers to be omitted. The WRAP and SINGLE settings are currently not used. WEIGHT will cause case weights to be printed along with variable values; NOWEIGHT, the default, causes case weights to be omitted from the output.

Case numbers start from 1. They are counted after all transformations have been considered.

LIST attempts to fit all the values on a single line. If needed to make them fit, variable names are displayed vertically. If values cannot fit on a single line, then a multi-line format will be used.

LIST is a procedure. It causes the data to be read.

6.9 MATRIX DATA

MATRIX DATA

```

/VARIABLES=var_list
/FILE='filename'
/FORMAT={LIST,FREE} {LOWER,UPPER,FULL} {DIAGONAL,NODIAGONAL}
/SPLIT={new_var,var_list}
/FACTORS=var_list
/CELLS=n_cells
/N=n
/CONTENTS={N_VECTOR,N_SCALAR,N_MATRIX,MEAN,STDDEV,COUNT,MSE,
          DFE,MAT,COV,CORR,PROX}

```

MATRIX DATA command reads square matrices in one of several textual formats. MATRIX DATA clears the dictionary and replaces it and reads a data file.

Use VARIABLES to specify the variables that form the rows and columns of the matrices. You may not specify a variable named VARNAME_. You should specify VARIABLES first.

Specify the file to read on FILE, either as a file name string or a file handle (see [Section 6.6 \[FILE HANDLE\]](#), page 43). If FILE is not specified then matrix data must immediately follow MATRIX DATA with a BEGIN DATA...END DATA construct (see [Section 6.1 \[BEGIN DATA\]](#), page 39).

The **FORMAT** subcommand specifies how the matrices are formatted. **LIST**, the default, indicates that there is one line per row of matrix data; **FREE** allows single matrix rows to be broken across multiple lines. This is analogous to the difference between **DATA LIST FREE** and **DATA LIST LIST** (see [Section 6.3 \[DATA LIST\]](#), page 39). **LOWER**, the default, indicates that the lower triangle of the matrix is given; **UPPER** indicates the upper triangle; and **FULL** indicates that the entire matrix is given. **DIAGONAL**, the default, indicates that the diagonal is part of the data; **NODIAGONAL** indicates that it is omitted. **DIAGONAL/NODIAGONAL** have no effect when **FULL** is specified.

The **SPLIT** subcommand is used to specify **SPLIT FILE** variables for the input matrices (see [Section 10.6 \[SPLIT FILE\]](#), page 74). Specify either a single variable not specified on **VARIABLES**, or one or more variables that are specified on **VARIABLES**. In the former case, the **SPLIT** values are not present in the data and **ROWTYPE_** may not be specified on **VARIABLES**. In the latter case, the **SPLIT** values are present in the data.

Specify a list of factor variables on **FACTORS**. Factor variables must also be listed on **VARIABLES**. Factor variables are used when there are some variables where, for each possible combination of their values, statistics on the matrix variables are included in the data.

If **FACTORS** is specified and **ROWTYPE_** is not specified on **VARIABLES**, the **CELLS** subcommand is required. Specify the number of factor variable combinations that are given. For instance, if factor variable **A** has 2 values and factor variable **B** has 3 values, specify 6.

The **N** subcommand specifies a population number of observations. When **N** is specified, one **N** record is output for each **SPLIT FILE**.

Use **CONTENTS** to specify what sort of information the matrices include. Each possible option is described in more detail below. When **ROWTYPE_** is specified on **VARIABLES**, **CONTENTS** is optional; otherwise, if **CONTENTS** is not specified then **/CONTENTS=CORR** is assumed.

N

N_VECTOR

Number of observations as a vector, one value for each variable.

N_SCALAR

Number of observations as a single value.

N_MATRIX

Matrix of counts.

MEAN

Vector of means.

STDDEV

Vector of standard deviations.

COUNT

Vector of counts.

MSE

Vector of mean squared errors.

DFE

Vector of degrees of freedom.

MAT

Generic matrix.

COV

Covariance matrix.

CORR

Correlation matrix.

PROX Proximities matrix.

The exact semantics of the matrices read by MATRIX DATA are complex. Right now MATRIX DATA isn't too useful due to a lack of procedures accepting or producing related data, so these semantics aren't documented. Later, they'll be described here in detail.

6.10 NEW FILE

NEW FILE.

NEW FILE command clears the current active file.

6.11 PRINT

PRINT

```
OUTFILE='filename'
RECORDS=n_lines
{NOTABLE, TABLE}
/[line_no] arg...
```

arg takes one of the following forms:

```
'string' [start-end]
var_list start-end [type-spec]
var_list (fortran-spec)
var_list *
```

The PRINT transformation writes variable data to an output file. PRINT is executed when a procedure causes the data to be read. Follow PRINT by EXECUTE to print variable data without invoking a procedure (see [Section 13.8 \[EXECUTE\]](#), page 90).

All PRINT subcommands are optional.

The OUTFILE subcommand specifies the file to receive the output. The file may be a file name as a string or a file handle (see [Section 6.6 \[FILE HANDLE\]](#), page 43). If OUTFILE is not present then output will be sent to PSPP's output listing file.

The RECORDS subcommand specifies the number of lines to be output. The number of lines may optionally be surrounded by parentheses.

TABLE will cause the PRINT command to output a table to the listing file that describes what it will print to the output file. NOTABLE, the default, suppresses this output table.

Introduce the strings and variables to be printed with a slash ('/'). Optionally, the slash may be followed by a number indicating which output line will be specified. In the absence of this line number, the next line number will be specified. Multiple lines may be specified using multiple slashes with the intended output for a line following its respective slash.

Literal strings may be printed. Specify the string itself. Optionally the string may be followed by a column number or range of column numbers, specifying the location on the line for the string to be printed. Otherwise, the string will be printed at the current position on the line.

Variables to be printed can be specified in the same ways as available for DATA LIST FIXED (see [Section 6.3.1 \[DATA LIST FIXED\]](#), page 39). In addition, a variable list may be followed by an asterisk (*), which indicates that the variables should be printed in their

dictionary print formats, separated by spaces. A variable list followed by a slash or the end of command will be interpreted the same way.

If a FORTRAN type specification is used to move backwards on the current line, then text is written at that point on the line, the line will be truncated to that length, although additional text being added will again extend the line to that length.

6.12 PRINT EJECT

```
PRINT EJECT
  OUTFILE='filename'
  RECORDS=n_lines
  {NOTABLE, TABLE}
  /[line_no] arg...
```

arg takes one of the following forms:

```
'string' [start-end]
var_list start-end [type_spec]
var_list (fortran_spec)
var_list *
```

PRINT EJECT writes data to an output file. Before the data is written, the current page in the listing file is ejected.

See [Section 6.11 \[PRINT\]](#), [page 49](#), for more information on syntax and usage.

6.13 PRINT SPACE

```
PRINT SPACE OUTFILE='filename' n_lines.
```

PRINT SPACE prints one or more blank lines to an output file.

The OUTFILE subcommand is optional. It may be used to direct output to a file specified by file name as a string or file handle (see [Section 6.6 \[FILE HANDLE\]](#), [page 43](#)). If OUTFILE is not specified then output will be directed to the listing file.

n_lines is also optional. If present, it is an expression (see [Chapter 5 \[Expressions\]](#), [page 21](#)) specifying the number of blank lines to be printed. The expression must evaluate to a nonnegative value.

6.14 REREAD

```
REREAD FILE=handle COLUMN=column.
```

The REREAD transformation allows the previous input line in a data file already processed by DATA LIST or another input command to be re-read for further processing.

The FILE subcommand, which is optional, is used to specify the file to have its line re-read. The file must be specified in the form of a file handle (see [Section 6.6 \[FILE HANDLE\]](#), [page 43](#)). If FILE is not specified then the last file specified on DATA LIST will be assumed (last file specified lexically, not in terms of flow-of-control).

By default, the line re-read is re-read in its entirety. With the COLUMN subcommand, a prefix of the line can be exempted from re-reading. Specify an expression (see [Chapter 5](#)

[Expressions], page 21) evaluating to the first column that should be included in the re-read line. Columns are numbered from 1 at the left margin.

Issuing **REREAD** multiple times will not back up in the data file. Instead, it will re-read the same line multiple times.

6.15 REPEATING DATA

REPEATING DATA

```

/STARTS=start-end
/OCCURS=n_occurs
/FILE='filename'
/LENGTH=length
/CONTINUED[=cont_start-cont_end]
/ID=id_start-id_end=id_var
/{TABLE,NOTABLE}
/DATA=var_spec...

```

where each `var_spec` takes one of the forms

```

var_list start-end [type_spec]
var_list (fortran_spec)

```

REPEATING DATA parses groups of data repeating in a uniform format, possibly with several groups on a single line. Each group of data corresponds with one case. **REPEATING DATA** may only be used within an **INPUT PROGRAM** structure (see [Section 6.7 \[INPUT PROGRAM\]](#), page 44). When used with **DATA LIST**, it can be used to parse groups of cases that share a subset of variables but differ in their other data.

The **STARTS** subcommand is required. Specify a range of columns, using literal numbers or numeric variable names. This range specifies the columns on the first line that are used to contain groups of data. The ending column is optional. If it is not specified, then the record width of the input file is used. For the inline file (see [Section 6.1 \[BEGIN DATA\]](#), page 39) this is 80 columns; for a file with fixed record widths it is the record width; for other files it is 1024 characters by default.

The **OCCURS** subcommand is required. It must be a number or the name of a numeric variable. Its value is the number of groups present in the current record.

The **DATA** subcommand is required. It must be the last subcommand specified. It is used to specify the data present within each repeating group. Column numbers are specified relative to the beginning of a group at column 1. Data is specified in the same way as with **DATA LIST FIXED** (see [Section 6.3.1 \[DATA LIST FIXED\]](#), page 39).

All other subcommands are optional.

FILE specifies the file to read, either a file name as a string or a file handle (see [Section 6.6 \[FILE HANDLE\]](#), page 43). If **FILE** is not present then the default is the last file handle used on **DATA LIST** (lexically, not in terms of flow of control).

By default **REPEATING DATA** will output a table describing how it will parse the input data. Specifying **NOTABLE** will disable this behavior; specifying **TABLE** will explicitly enable it.

The `LENGTH` subcommand specifies the length in characters of each group. If it is not present then length is inferred from the `DATA` subcommand. `LENGTH` can be a number or a variable name.

Normally all the data groups are expected to be present on a single line. Use the `CONTINUED` command to indicate that data can be continued onto additional lines. If data on continuation lines starts at the left margin and continues through the entire field width, no column specifications are necessary on `CONTINUED`. Otherwise, specify the possible range of columns in the same way as on `STARTS`.

When data groups are continued from line to line, it is easy for cases to get out of sync through careless hand editing. The `ID` subcommand allows a case identifier to be present on each line of repeating data groups. `REPEATING DATA` will check for the same identifier on each line and report mismatches. Specify the range of columns that the identifier will occupy, followed by an equals sign (`=`) and the identifier variable name. The variable must already have been declared with `NUMERIC` or another command.

`REPEATING DATA` should be the last command given within an `INPUT PROGRAM`. It should not be enclosed within a `LOOP` structure (see [Section 11.4 \[LOOP\]](#), page 78). Use `DATA LIST` before, not after, `REPEATING DATA`.

6.16 WRITE

```
WRITE
  OUTFILE='filename'
  RECORDS=n_lines
  {NOTABLE, TABLE}
  /[line_no] arg. . .
```

`arg` takes one of the following forms:

```
'string' [start-end]
var_list start-end [type-spec]
var_list (fortran-spec)
var_list *
```

`WRITE` writes text or binary data to an output file.

See [Section 6.11 \[PRINT\]](#), page 49, for more information on syntax and usage. The main difference between `PRINT` and `WRITE` is that `WRITE` uses write formats by default, where `PRINT` uses print formats.

The sole additional difference is that if `WRITE` is used to send output to a binary file, carriage control characters will not be output. See [Section 6.6 \[FILE HANDLE\]](#), page 43, for information on how to declare a file as binary.

7 System Files and Portable Files

The commands in this chapter read, write, and examine system files and portable files.

7.1 APPLY DICTIONARY

APPLY DICTIONARY FROM='filename'.

APPLY DICTIONARY applies the variable labels, value labels, and missing values from variables in a system file to corresponding variables in the active file. In some cases it also updates the weighting variable.

Specify a system file with a file name string or as a file handle (see [Section 6.6 \[FILE HANDLE\]](#), page 43). The dictionary in the system file will be read, but it will not replace the active file dictionary. The system file's data will not be read.

Only variables with names that exist in both the active file and the system file are considered. Variables with the same name but different types (numeric, string) will cause an error message. Otherwise, the system file variables' attributes will replace those in their matching active file variables, as described below.

If a system file variable has a variable label, then it will replace the active file variable's variable label. If the system file variable does not have a variable label, then the active file variable's variable label, if any, will be retained.

If the active file variable is numeric or short string, then value labels and missing values, if any, will be copied to the active file variable. If the system file variable does not have value labels or missing values, then those in the active file variable, if any, will not be disturbed.

Finally, weighting of the active file is updated (see [Section 10.8 \[WEIGHT\]](#), page 75). If the active file has a weighting variable, and the system file does not, or if the weighting variable in the system file does not exist in the active file, then the active file weighting variable, if any, is retained. Otherwise, the weighting variable in the system file becomes the active file weighting variable.

APPLY DICTIONARY takes effect immediately. It does not read the active file. The system file is not modified.

7.2 EXPORT

```
EXPORT
  /OUTFILE='filename'
  /DROP=var_list
  /KEEP=var_list
  /RENAME=(src_names=target_names)...
```

The EXPORT procedure writes the active file dictionary and data to a specified portable file.

The OUTFILE subcommand, which is the only required subcommand, specifies the portable file to be written as a file name string or a file handle (see [Section 6.6 \[FILE HANDLE\]](#), page 43).

DROP, KEEP, and RENAME follow the same format as the SAVE procedure (see [Section 7.6 \[SAVE\]](#), page 56).

EXPORT is a procedure. It causes the active file to be read.

7.3 GET

```
GET
    /FILE='filename'
    /DROP=var_list
    /KEEP=var_list
    /RENAME=(src_names=target_names)...
```

GET clears the current dictionary and active file and replaces them with the dictionary and data from a specified system file.

The FILE subcommand is the only required subcommand. Specify the system file to be read as a string file name or a file handle (see [Section 6.6 \[FILE HANDLE\]](#), page 43).

By default, all the variables in a system file are read. The DROP subcommand can be used to specify a list of variables that are not to be read. By contrast, the KEEP subcommand can be used to specify variable that are to be read, with all other variables not read.

Normally variables in a system file retain the names that they were saved under. Use the RENAME subcommand to change these names. Specify, within parentheses, a list of variable names followed by an equals sign (=) and the names that they should be renamed to. Multiple parenthesized groups of variable names can be included on a single RENAME subcommand. Variables' names may be swapped using a RENAME subcommand of the form `/RENAME=(A B=B A)`.

Alternate syntax for the RENAME subcommand allows the parentheses to be eliminated. When this is done, only a single variable may be renamed at once. For instance, `/RENAME=A=B`. This alternate syntax is deprecated.

DROP, KEEP, and RENAME are performed in left-to-right order. They each may be present any number of times. GET never modifies a system file on disk. Only the active file read from the system file is affected by these subcommands.

GET does not cause the data to be read, only the dictionary. The data is read later, when a procedure is executed.

7.4 IMPORT

```
IMPORT
    /FILE='filename'
    /TYPE={COMM,TAPE}
    /DROP=var_list
    /KEEP=var_list
    /RENAME=(src_names=target_names)...
```

The IMPORT transformation clears the active file dictionary and data and replaces them with a dictionary and data from a portable file on disk.

The FILE subcommand, which is the only required subcommand, specifies the portable file to be read as a file name string or a file handle (see [Section 6.6 \[FILE HANDLE\]](#), page 43).

The TYPE subcommand is currently not used.

DROP, KEEP, and RENAME follow the syntax used by GET (see [Section 7.3 \[GET\]](#), page 54).

IMPORT does not cause the data to be read, only the dictionary. The data is read later, when a procedure is executed.

7.5 MATCH FILES

MATCH FILES

```
/{FILE, TABLE}={*, 'filename'}
/DROP=var_list
/KEEP=var_list
/RENAME=(src_names=target_names)...
/IN=var_name

/BY var_list
/FIRST=var_name
/LAST=var_name
/MAP
```

MATCH FILES merges one or more system files, optionally including the active file. Records with the same values for BY variables are combined into a single record. Records with different values are output in order. Thus, multiple sorted system files are combined into a single sorted system file based on the value of the BY variables. The results of the merge become the new active file.

The BY subcommand specifies a list of variables that are used to match records from each of the system files. Variables specified must exist in all the files specified on FILE and TABLE. BY should usually be specified. If TABLE or IN is used then BY is required.

Specify FILE with a system file as a file name string or file handle (see [Section 6.6 \[FILE HANDLE\]](#), page 43), or with an asterisk (*) to indicate the current active file. The files specified on FILE are merged together based on the BY variables, or combined case-by-case if BY is not specified. Normally at least two FILE subcommands should be specified.

Specify TABLE with a system file to use it as a *table lookup file*. Records in table lookup files are not used up after they've been used once. This means that data in table lookup files can correspond to any number of records in FILE files. Table lookup files correspond to lookup tables in traditional relational database systems. It is incorrect to have records with duplicate BY values in table lookup files.

Any number of FILE and TABLE subcommands may be specified. Each instance of FILE or TABLE can be followed by any sequence of DROP, KEEP, or RENAME subcommands. These have the same form and meaning as the corresponding subcommands of GET (see [Section 7.3 \[GET\]](#), page 54), but apply only to variables in the given file.

Each FILE or TABLE may optionally be followed by an IN subcommand, which creates a numeric variable with the specified name and format F1.0. The IN variable takes value 1 in a case if the given file contributed a row to the merged file, 0 otherwise. The DROP, KEEP, and RENAME subcommands do not affect IN variables.

Variables belonging to files that are not present for the current case are set to the system-missing value for numeric variables or spaces for string variables.

FIRST, LAST, and MAP are currently ignored.

MATCH FILES may not be specified following TEMPORARY (see [Section 10.7 \[TEMPORARY\]](#), page 75) if the active file is used as an input source.

7.6 SAVE

```
SAVE
  /OUTFILE='filename'
  /{COMPRESSED,UNCOMPRESSED}
  /DROP=var_list
  /KEEP=var_list
  /VERSION=version
  /RENAME=(src_names=target_names)...
```

The SAVE procedure causes the dictionary and data in the active file to be written to a system file.

OUTFILE is the only required subcommand. Specify the system file to be written as a string file name or a file handle (see [Section 6.6 \[FILE HANDLE\]](#), page 43).

The COMPRESS and UNCOMPRESS subcommand determine whether the saved system file is compressed. By default, system files are compressed. This default can be changed with the SET command (see [Section 13.15 \[SET\]](#), page 91).

By default, all the variables in the active file dictionary are written to the system file. The DROP subcommand can be used to specify a list of variables not to be written. In contrast, KEEP specifies variables to be written, with all variables not specified not written.

Normally variables are saved to a system file under the same names they have in the active file. Use the RENAME subcommand to change these names. Specify, within parentheses, a list of variable names followed by an equals sign ('=') and the names that they should be renamed to. Multiple parenthesized groups of variable names can be included on a single RENAME subcommand. Variables' names may be swapped using a RENAME subcommand of the form '/RENAME=(A B=B A)'.

Alternate syntax for the RENAME subcommand allows the parentheses to be eliminated. When this is done, only a single variable may be renamed at once. For instance, '/RENAME=A=B'. This alternate syntax is deprecated.

DROP, KEEP, and RENAME are performed in left-to-right order. They each may be present any number of times. SAVE never modifies the active file. DROP, KEEP, and RENAME only affect the system file written to disk.

The VERSION subcommand specifies the version of the file format. Valid versions are '3' and '3x'. Version 3x system files are identical to version 3 files, except that variable names greater than 8 bytes will be truncated. The default version is 3. The VERSION subcommand is optional. There is no need ever to use it.

SAVE causes the data to be read. It is a procedure.

7.7 SYSFILE INFO

```
SYSFILE INFO FILE='filename'.
```

SYSFILE INFO reads the dictionary in a system file and displays the information in its dictionary.

Specify a file name or file handle. SYSFILE INFO reads that file as a system file and displays information on its dictionary.

SYSFILE INFO does not affect the current active file.

7.8 XSAVE

```
XSAVE
  /OUTFILE='filename'
  /{COMPRESSED,UNCOMPRESSED}
  /DROP=var_list
  /KEEP=var_list
  /RENAME=(src_names=target_names)...
```

The XSAVE transformation writes the active file dictionary and data to a system file stored on disk.

XSAVE is a transformation, not a procedure. It is executed when the data is read by a procedure or procedure-like command. In all other respects, XSAVE is identical to SAVE. See [Section 7.6 \[SAVE\]](#), [page 56](#), for more information on syntax and usage.

8 Manipulating variables

The variables in the active file dictionary are important. There are several utility functions for examining and adjusting them.

8.1 ADD VALUE LABELS

```
ADD VALUE LABELS
    /var_list value 'label' [value 'label'] . . .
```

ADD VALUE LABELS has the same syntax and purpose as VALUE LABELS (see [Section 8.11 \[VALUE LABELS\], page 61](#)), but it does not clear value labels from the variables before adding the ones specified.

8.2 DISPLAY

```
DISPLAY {NAMES,INDEX,LABELS,VARIABLES,DICTIONARY,SCRATCH}
    [SORTED] [var_list]
```

DISPLAY displays requested information on variables. Variables can optionally be sorted alphabetically. The entire dictionary or just specified variables can be described.

One of the following keywords can be present:

- NAMES The variables' names are displayed.
- INDEX The variables' names are displayed along with a value describing their position within the active file dictionary.
- LABELS Variable names, positions, and variable labels are displayed.
- VARIABLES
 Variable names, positions, print and write formats, and missing values are displayed.
- DICTIONARY
 Variable names, positions, print and write formats, missing values, variable labels, and value labels are displayed.
- SCRATCH
 Variable names are displayed, for scratch variables only (see [Section 4.6.5 \[Scratch Variables\], page 19](#)).

If SORTED is specified, then the variables are displayed in ascending order based on their names; otherwise, they are displayed in the order that they occur in the active file dictionary.

8.3 DISPLAY VECTORS

```
DISPLAY VECTORS.
```

DISPLAY VECTORS lists all the currently declared vectors.

8.4 FORMATS

FORMATS var_list (fmt_spec).

FORMATS set both print and write formats for the specified numeric variables to the specified format specification. See [Section 4.6.4 \[Input/Output Formats\]](#), page 14.

Specify a list of variables followed by a format specification in parentheses. The print and write formats of the specified variables will be changed.

Additional lists of variables and formats may be included if they are delimited by a slash ('/').

FORMATS takes effect immediately. It is not affected by conditional and looping structures such as DO IF or LOOP.

8.5 LEAVE

LEAVE var_list.

LEAVE prevents the specified variables from being reinitialized whenever a new case is processed.

Normally, when a data file is processed, every variable in the active file is initialized to the system-missing value or spaces at the beginning of processing for each case. When a variable has been specified on LEAVE, this is not the case. Instead, that variable is initialized to 0 (not system-missing) or spaces for the first case. After that, it retains its value between cases.

This becomes useful for counters. For instance, in the example below the variable SUM maintains a running total of the values in the ITEM variable.

```
DATA LIST /ITEM 1-3.
COMPUTE SUM=SUM+ITEM.
PRINT /ITEM SUM.
LEAVE SUM
BEGIN DATA.
123
404
555
999
END DATA.
```

Partial output from this example:

```
123    123.00
404    527.00
555   1082.00
999   2081.00
```

It is best to use LEAVE command immediately before invoking a procedure command, because the left status of variables is reset by certain transformations—for instance, COMPUTE and IF. Left status is also reset by all procedure invocations.

8.6 MISSING VALUES

`MISSING VALUES var_list (missing_values).`

`missing_values` takes one of the following forms:

```
num1
num1, num2
num1, num2, num3
num1 THRU num2
num1 THRU num2, num3
string1
string1, string2
string1, string2, string3
```

As part of a range, `LO` or `LOWEST` may take the place of `num1`;

`HI` or `HIGHEST` may take the place of `num2`.

`MISSING VALUES` sets user-missing values for numeric and short string variables. Long string variables may not have missing values.

Specify a list of variables, followed by a list of their user-missing values in parentheses. Up to three discrete values may be given, or, for numeric variables only, a range of values optionally accompanied by a single discrete value. Ranges may be open-ended on one end, indicated through the use of the keyword `LO` or `LOWEST` or `HI` or `HIGHEST`.

The `MISSING VALUES` command takes effect immediately. It is not affected by conditional and looping constructs such as `DO IF` or `LOOP`.

8.7 MODIFY VARS

`MODIFY VARS`

```
/REORDER={FORWARD,BACKWARD} {POSITIONAL,ALPHA} (var_list) . . .
/RENAME=(old_names=new_names) . . .
/{DROP,KEEP}=var_list
/MAP
```

`MODIFY VARS` reorders, renames, and deletes variables in the active file.

At least one subcommand must be specified, and no subcommand may be specified more than once. `DROP` and `KEEP` may not both be specified.

The `REORDER` subcommand changes the order of variables in the active file. Specify one or more lists of variable names in parentheses. By default, each list of variables is rearranged into the specified order. To put the variables into the reverse of the specified order, put keyword `BACKWARD` before the parentheses. To put them into alphabetical order in the dictionary, specify keyword `ALPHA` before the parentheses. `BACKWARD` and `ALPHA` may also be combined.

To rename variables in the active file, specify `RENAME`, an equals sign (`=`), and lists of the old variable names and new variable names separated by another equals sign within parentheses. There must be the same number of old and new variable names. Each old variable is renamed to the corresponding new variable name. Multiple parenthesized groups of variables may be specified.

The `DROP` subcommand deletes a specified list of variables from the active file.

The KEEP subcommand keeps the specified list of variables in the active file. Any unlisted variables are deleted from the active file.

MAP is currently ignored.

If either DROP or KEEP is specified, the data is read; otherwise it is not.

MODIFY VARS may not be specified following TEMPORARY (see [Section 10.7 \[TEMPORARY\]](#), page 75).

8.8 NUMERIC

NUMERIC /var_list [(fmt_spec)].

NUMERIC explicitly declares new numeric variables, optionally setting their output formats.

Specify a slash ('/'), followed by the names of the new numeric variables. If you wish to set their output formats, follow their names by an output format specification in parentheses (see [Section 4.6.4 \[Input/Output Formats\]](#), page 14); otherwise, the default is F8.2.

Variables created with NUMERIC are initialized to the system-missing value.

8.9 PRINT FORMATS

PRINT FORMATS var_list (fmt_spec).

PRINT FORMATS sets the print formats for the specified numeric variables to the specified format specification.

Its syntax is identical to that of FORMATS (see [Section 8.4 \[FORMATS\]](#), page 59), but PRINT FORMATS sets only print formats, not write formats.

8.10 RENAME VARIABLES

RENAME VARIABLES (old_names=new_names)

RENAME VARIABLES changes the names of variables in the active file. Specify lists of the old variable names and new variable names, separated by an equals sign ('='), within parentheses. There must be the same number of old and new variable names. Each old variable is renamed to the corresponding new variable name. Multiple parenthesized groups of variables may be specified.

RENAME VARIABLES takes effect immediately. It does not cause the data to be read.

RENAME VARIABLES may not be specified following TEMPORARY (see [Section 10.7 \[TEMPORARY\]](#), page 75).

8.11 VALUE LABELS

VALUE LABELS

/var_list value 'label' [value 'label'] . . .

VALUE LABELS allows values of numeric and short string variables to be associated with labels. In this way, a short value can stand for a long value.

To set up value labels for a set of variables, specify the variable names after a slash ('/'), followed by a list of values and their associated labels, separated by spaces. Long string variables may not be specified.

Before VALUE LABELS is executed, any existing value labels are cleared from the variables specified. Use ADD VALUE LABELS (see [Section 8.1 \[ADD VALUE LABELS\]](#), [page 58](#)) to add value labels without clearing those already present.

8.12 STRING

STRING /var_list (fmt-spec).

STRING creates new string variables for use in transformations.

Specify a slash ('/'), followed by the names of the string variables to create and the desired output format specification in parentheses (see [Section 4.6.4 \[Input/Output Formats\]](#), [page 14](#)). Variable widths are implicitly derived from the specified output formats.

Created variables are initialized to spaces.

8.13 VARIABLE LABELS

VARIABLE LABELS

```
var_list 'var_label'
[ /var_list 'var_label']
.
.
.
[ /var_list 'var_label']
```

VARIABLE LABELS associates explanatory names with variables. This name, called a *variable label*, is displayed by statistical procedures.

To assign a variable label to a group of variables, specify a list of variable names and the variable label as a string. To assign different labels to different variables in the same command, precede the subsequent variable list with a slash ('/').

8.14 VARIABLE ALIGNMENT

VARIABLE ALIGNMENT

```
var_list ( LEFT | RIGHT | CENTER )
[ /var_list ( LEFT | RIGHT | CENTER ) ]
.
.
.
[ /var_list ( LEFT | RIGHT | CENTER ) ]
```

VARIABLE ALIGNMENT sets the alignment of variables for display editing purposes. This only has effect for third party software. It does not affect the display of variables in the PSPP output.

8.15 VARIABLE WIDTH

VARIABLE WIDTH

```
var_list (width)
[ /var_list (width) ]
.
```

```

.
.
[ /var_list (width) ]

```

VARIABLE WIDTH sets the column width of variables for display editing purposes. This only affects third party software. It does not affect the display of variables in the PSPP output.

8.16 VARIABLE LEVEL

```

VARIABLE LEVEL
var_list ( SCALE | NOMINAL | ORDINAL )
[ /var_list ( SCALE | NOMINAL | ORDINAL ) ]
.
.
.
[ /var_list ( SCALE | NOMINAL | ORDINAL ) ]

```

VARIABLE LEVEL sets the measurement level of variables. Currently, this has no effect except for certain third party software.

8.17 VECTOR

Two possible syntaxes:

```

VECTOR vec_name=var_list.
VECTOR vec_name_list(count).

```

VECTOR allows a group of variables to be accessed as if they were consecutive members of an array with a vector(index) notation.

To make a vector out of a set of existing variables, specify a name for the vector followed by an equals sign ('=') and the variables that belong in the vector.

To make a vector and create variables at the same time, specify one or more vector names followed by a count in parentheses. This will cause variables named **vec1** through **vec count** to be created as numeric variables with print and write format F8.2. Variable names including numeric suffixes may not exceed 64 characters in length, and none of the variables may exist prior to VECTOR.

All the variables in a vector must be the same type.

Vectors created with VECTOR disappear after any procedure or procedure-like command is executed. The variables contained in the vectors remain, unless they are scratch variables (see [Section 4.6.5 \[Scratch Variables\]](#), [page 19](#)).

Variables within a vector may be referenced in expressions using **vector(index)** syntax.

8.18 WRITE FORMATS

```

WRITE FORMATS var_list (fmt-spec).

```

WRITE FORMATS sets the write formats for the specified numeric variables to the specified format specification. Its syntax is identical to that of FORMATS (see [Section 8.4 \[FORMATS\]](#), [page 59](#)), but WRITE FORMATS sets only write formats, not print formats.

9 Data transformations

The PSPP procedures examined in this chapter manipulate data and prepare the active file for later analyses. They do not produce output, as a rule.

9.1 AGGREGATE

```
AGGREGATE
  OUTFILE={*, 'filename'}
  /PRESORTED
  /DOCUMENT
  /MISSING=COLUMNWISE
  /BREAK=var_list
  /dest_var['label']...=agr_func(src_vars, args...).
```

AGGREGATE summarizes groups of cases into single cases. Cases are divided into groups that have the same values for one or more variables called *break variables*. Several functions are available for summarizing case contents.

The OUTFILE subcommand is required and must appear first. Specify a system file by file name string or file handle (see [Section 6.6 \[FILE HANDLE\]](#), page 43). The aggregated cases are written to this file. If '*' is specified, then the aggregated cases replace the active file.

By default, the active file will be sorted based on the break variables before aggregation takes place. If the active file is already sorted or otherwise grouped in terms of the break variables, specify PRESORTED to save time.

Specify DOCUMENT to copy the documents from the active file into the aggregate file (see [Section 13.2 \[DOCUMENT\]](#), page 89). Otherwise, the aggregate file will not contain any documents, even if the aggregate file replaces the active file.

Normally, only a single case (for SD and SD., two cases) need be non-missing in each group for the aggregate variable to be non-missing. Specifying /MISSING=COLUMNWISE inverts this behavior, so that the aggregate variable becomes missing if any aggregated value is missing.

If PRESORTED, DOCUMENT, or MISSING are specified, they must appear between OUTFILE and BREAK.

At least one break variable must be specified on BREAK, a required subcommand. The values of these variables are used to divide the active file into groups to be summarized. In addition, at least one *dest_var* must be specified.

One or more sets of aggregation variables must be specified. Each set comprises a list of aggregation variables, an equals sign ('='), the name of an aggregation function (see the list below), and a list of source variables in parentheses. Some aggregation functions expect additional arguments following the source variable names.

Aggregation variables typically are created with no variable label, value labels, or missing values. Their default print and write formats depend on the aggregation function used, with details given in the table below. A variable label for an aggregation variable may be specified just after the variable's name in the aggregation variable list.

Each set must have exactly as many source variables as aggregation variables. Each aggregation variable receives the results of applying the specified aggregation function to the corresponding source variable. The MEAN, SD, and SUM aggregation functions may only be applied to numeric variables. All the rest may be applied to numeric and short and long string variables.

The available aggregation functions are as follows:

FGT(var_name, value)

Fraction of values greater than the specified constant. The default format is F5.3.

FIN(var_name, low, high)

Fraction of values within the specified inclusive range of constants. The default format is F5.3.

FLT(var_name, value)

Fraction of values less than the specified constant. The default format is F5.3.

FIRST(var_name)

First non-missing value in break group. The aggregation variable receives the complete dictionary information from the source variable. The sort performed by AGGREGATE (and by SORT CASES) is stable, so that the first case with particular values for the break variables before sorting will also be the first case in that break group after sorting.

FOUT(var_name, low, high)

Fraction of values strictly outside the specified range of constants. The default format is F5.3.

LAST(var_name)

Last non-missing value in break group. The aggregation variable receives the complete dictionary information from the source variable. The sort performed by AGGREGATE (and by SORT CASES) is stable, so that the last case with particular values for the break variables before sorting will also be the last case in that break group after sorting.

MAX(var_name)

Maximum value. The aggregation variable receives the complete dictionary information from the source variable.

MEAN(var_name)

Arithmetic mean. Limited to numeric values. The default format is F8.2.

MIN(var_name)

Minimum value. The aggregation variable receives the complete dictionary information from the source variable.

N(var_name)

Number of non-missing values. The default format is F7.0 if weighting is not enabled, F8.2 if it is (see [Section 10.8 \[WEIGHT\]](#), page 75).

N

Number of cases aggregated to form this group. The default format is F7.0 if weighting is not enabled, F8.2 if it is (see [Section 10.8 \[WEIGHT\]](#), page 75).

- NMISS(var_name)
 Number of missing values. The default format is F7.0 if weighting is not enabled, F8.2 if it is (see [Section 10.8 \[WEIGHT\]](#), page 75).
- NU(var_name)
 Number of non-missing values. Each case is considered to have a weight of 1, regardless of the current weighting variable (see [Section 10.8 \[WEIGHT\]](#), page 75). The default format is F7.0.
- NU
 Number of cases aggregated to form this group. Each case is considered to have a weight of 1, regardless of the current weighting variable. The default format is F7.0.
- NUMISS(var_name)
 Number of missing values. Each case is considered to have a weight of 1, regardless of the current weighting variable. The default format is F7.0.
- PGT(var_name, value)
 Percentage between 0 and 100 of values greater than the specified constant. The default format is F5.1.
- PIN(var_name, low, high)
 Percentage of values within the specified inclusive range of constants. The default format is F5.1.
- PLT(var_name, value)
 Percentage of values less than the specified constant. The default format is F5.1.
- POUT(var_name, low, high)
 Percentage of values strictly outside the specified range of constants. The default format is F5.1.
- SD(var_name)
 Standard deviation of the mean. Limited to numeric values. The default format is F8.2.
- SUM(var_name)
 Sum. Limited to numeric values. The default format is F8.2.

Aggregation functions compare string values in terms of internal character codes. On most modern computers, this is a form of ASCII.

The aggregation functions listed above exclude all user-missing values from calculations. To include user-missing values, insert a period (‘.’) at the end of the function name. (e.g. ‘SUM.’). (Be aware that specifying such a function as the last token on a line will cause the period to be interpreted as the end of the command.)

AGGREGATE both ignores and cancels the current SPLIT FILE settings (see [Section 10.6 \[SPLIT FILE\]](#), page 74).

9.2 AUTORECODE

```
AUTORECODE VARIABLES=src_vars INTO dest_vars
/DESCENDING
/PRINT
```

The AUTORECODE procedure considers the n values that a variable takes on and maps them onto values $1 \dots n$ on a new numeric variable.

Subcommand VARIABLES is the only required subcommand and must come first. Specify VARIABLES, an equals sign ('='), a list of source variables, INTO, and a list of target variables. There must be the same number of source and target variables. The target variables must not already exist.

By default, increasing values of a source variable (for a string, this is based on character code comparisons) are recoded to increasing values of its target variable. To cause increasing values of a source variable to be recoded to decreasing values of its target variable (n down to 1), specify DESCENDING.

PRINT is currently ignored.

AUTORECODE is a procedure. It causes the data to be read.

9.3 COMPUTE

```
COMPUTE variable = expression.
or
COMPUTE vector(index) = expression.
```

COMPUTE assigns the value of an expression to a target variable. For each case, the expression is evaluated and its value assigned to the target variable. Numeric and short and long string variables may be assigned. When a string expression's width differs from the target variable's width, the string result of the expression is truncated or padded with spaces on the right as necessary. The expression and variable types must match.

For numeric variables only, the target variable need not already exist. Numeric variables created by COMPUTE are assigned an F8.2 output format. String variables must be declared before they can be used as targets for COMPUTE.

The target variable may be specified as an element of a vector (see [Section 8.17 \[VECTOR\]](#), page 63). In this case, a vector index expression must be specified in parentheses following the vector name. The index expression must evaluate to a numeric value that, after rounding down to the nearest integer, is a valid index for the named vector.

Using COMPUTE to assign to a variable specified on LEAVE (see [Section 8.5 \[LEAVE\]](#), page 59) resets the variable's left state. Therefore, LEAVE should be specified following COMPUTE, not before.

COMPUTE is a transformation. It does not cause the active file to be read.

When COMPUTE is specified following TEMPORARY (see [Section 10.7 \[TEMPORARY\]](#), page 75), the LAG function may not be used (see [\[LAG\]](#), page 31).

9.4 COUNT

```
COUNT var_name = var... (value...).
```

Each value takes one of the following forms:

```
number
string
num1 THRU num2
MISSING
SYSMIS
```

In addition, num1 and num2 can be LO or LOWEST, or HI or HIGHEST, respectively.

COUNT creates or replaces a numeric *target* variable that counts the occurrence of a *criterion* value or set of values over one or more *test* variables for each case.

The target variable values are always nonnegative integers. They are never missing. The target variable is assigned an F8.2 output format. See [Section 4.6.4 \[Input/Output Formats\]](#), page 14. Any variables, including long and short string variables, may be test variables.

User-missing values of test variables are treated just like any other values. They are **not** treated as system-missing values. User-missing values that are criterion values or inside ranges of criterion values are counted as any other values. However (for numeric variables), keyword MISSING may be used to refer to all system- and user-missing values.

COUNT target variables are assigned values in the order specified. In the command `COUNT A=A B(1) /B=A B(2) .`, the following actions occur:

- The number of occurrences of 1 between A and B is counted.
- A is assigned this value.
- The number of occurrences of 1 between B and the **new** value of A is counted.
- B is assigned this value.

Despite this ordering, all COUNT criterion variables must exist before the procedure is executed—they may not be created as target variables earlier in the command! Break such a command into two separate commands.

The examples below may help to clarify.

A. Assuming Q0, Q2, . . . , Q9 are numeric variables, the following commands:

1. Count the number of times the value 1 occurs through these variables for each case and assigns the count to variable QCOUNT.
2. Print out the total number of times the value 1 occurs throughout *all* cases using DESCRIPTIVES. See [Section 12.1 \[DESCRIPTIVES\]](#), page 80, for details.

```
COUNT QCOUNT=Q0 TO Q9(1).
DESCRIPTIVES QCOUNT /STATISTICS=SUM.
```

B. Given these same variables, the following commands:

1. Count the number of valid values of these variables for each case and assigns the count to variable QVALID.
2. Multiplies each value of QVALID by 10 to obtain a percentage of valid values, using COMPUTE. See [Section 9.3 \[COMPUTE\]](#), page 67, for details.
3. Print out the percentage of valid values across all cases, using DESCRIPTIVES. See [Section 12.1 \[DESCRIPTIVES\]](#), page 80, for details.

```

COUNT QVALID=Q0 TO Q9 (LO THRU HI).
COMPUTE QVALID=QVALID*10.
DESCRIPTIVES QVALID /STATISTICS=MEAN.

```

9.5 FLIP

FLIP /VARIABLES=var_list /NEWNAMES=var_name.

FLIP transposes rows and columns in the active file. It causes cases to be swapped with variables, and vice versa.

All variables in the transposed active file are numeric. String variables take on the system-missing value in the transposed file.

No subcommands are required. If specified, the VARIABLES subcommand selects variables to be transformed into cases, and variables not specified are discarded. If the VARIABLES subcommand is omitted, all variables are selected for transposition.

The variables specified by NEWNAMES, which must be a string variable, is used to give names to the variables created by FLIP. Only the first 8 characters of the variable are used. If NEWNAMES is not specified then the default is a variable named CASE_LBL, if it exists. If it does not then the variables created by FLIP are named VAR000 through VAR999, then VAR1000, VAR1001, and so on.

When a NEWNAMES variable is available, the names must be canonicalized before becoming variable names. Invalid characters are replaced by letter ‘V’ in the first position, or by ‘_’ in subsequent positions. If the name thus generated is not unique, then numeric extensions are added, starting with 1, until a unique name is found or there are no remaining possibilities. If the latter occurs then the FLIP operation aborts.

The resultant dictionary contains a CASE_LBL variable, a string variable of width 8, which stores the names of the variables in the dictionary before the transposition. Variables names longer than 8 characters are truncated. If the active file is subsequently transposed using FLIP, this variable can be used to recreate the original variable names.

FLIP honors N OF CASES (see [Section 10.2 \[N OF CASES\], page 72](#)). It ignores TEMPORARY (see [Section 10.7 \[TEMPORARY\], page 75](#)), so that “temporary” transformations become permanent.

9.6 IF

IF condition variable=expression.

or

IF condition vector(index)=expression.

The IF transformation conditionally assigns the value of a target expression to a target variable, based on the truth of a test expression.

Specify a boolean-valued expression (see [Chapter 5 \[Expressions\], page 21](#)) to be tested following the IF keyword. This expression is evaluated for each case. If the value is true, then the value of the expression is computed and assigned to the specified variable. If the value is false or missing, nothing is done. Numeric and short and long string variables may be assigned. When a string expression’s width differs from the target variable’s width, the string result of the expression is truncated or padded with spaces on the right as necessary. The expression and variable types must match.

The target variable may be specified as an element of a vector (see [Section 8.17 \[VECTOR\]](#), [page 63](#)). In this case, a vector index expression must be specified in parentheses following the vector name. The index expression must evaluate to a numeric value that, after rounding down to the nearest integer, is a valid index for the named vector.

Using IF to assign to a variable specified on LEAVE (see [Section 8.5 \[LEAVE\]](#), [page 59](#)) resets the variable's left state. Therefore, **LEAVE** should be specified following IF, not before.

When IF is specified following TEMPORARY (see [Section 10.7 \[TEMPORARY\]](#), [page 75](#)), the LAG function may not be used (see [\[LAG\]](#), [page 31](#)).

9.7 RECODE

RECODE var_list (src_value...=dest_value)... [INTO var_list].

src_value may take the following forms:

```
number
string
num1 THRU num2
MISSING
SYSMIS
ELSE
```

Open-ended ranges may be specified using LO or LOWEST for num1 or HI or HIGHEST for num2.

dest_value may take the following forms:

```
num
string
SYSMIS
COPY
```

RECODE translates data from one range of values to another, via flexible user-specified mappings. Data may be remapped in-place or copied to new variables. Numeric, short string, and long string data can be recoded.

Specify the list of source variables, followed by one or more mapping specifications each enclosed in parentheses. If the data is to be copied to new variables, specify INTO, then the list of target variables. String target variables must already have been declared using **STRING** or another transformation, but numeric target variables can be created on the fly. There must be exactly as many target variables as source variables. Each source variable is remapped into its corresponding target variable.

When INTO is not used, the input and output variables must be of the same type. Otherwise, string values can be recoded into numeric values, and vice versa. When this is done and there is no mapping for a particular value, either a value consisting of all spaces or the system-missing value is assigned, depending on variable type.

Mappings are considered from left to right. The first src_value that matches the value of the source variable causes the target variable to receive the value indicated by the dest_value. Literal number, string, and range src_value's should be self-explanatory. **MISSING** as a src_value matches any user- or system-missing value. **SYSMIS** matches the system missing

value only. ELSE is a catch-all that matches anything. It should be the last `src_value` specified.

Numeric and string `dest_value`'s should also be self-explanatory. COPY causes the input values to be copied to the output. This is only value if the source and target variables are of the same type. SYSMIS indicates the system-missing value.

If the source variables are strings and the target variables are numeric, then there is one additional mapping available: (CONVERT), which must be the last specified mapping. CONVERT causes a number specified as a string to be converted to a numeric value. If the string cannot be parsed as a number, then the system-missing value is assigned.

Multiple recodings can be specified on a single RECODE invocation. Introduce additional recodings with a slash ('/') to separate them from the previous recodings.

9.8 SORT CASES

`SORT CASES BY var_list.`

`SORT CASES` sorts the active file by the values of one or more variables.

Specify BY and a list of variables to sort by. By default, variables are sorted in ascending order. To override sort order, specify (D) or (DOWN) after a list of variables to get descending order, or (A) or (UP) for ascending order. These apply to the entire list of variables preceding them.

The sort algorithms used by `SORT CASES` are stable. That is, records that have equal values of the sort variables will have the same relative order before and after sorting. As a special case, re-sorting an already sorted file will not affect the ordering of cases.

`SORT CASES` is a procedure. It causes the data to be read.

`SORT CASES` attempts to sort the entire active file in main memory. If workspace is exhausted, it falls back to a merge sort algorithm that involves creates numerous temporary files.

`SORT CASES` may not be specified following `TEMPORARY`.

10 Selecting data for analysis

This chapter documents PSPP commands that temporarily or permanently select data records from the active file for analysis.

10.1 FILTER

```
FILTER BY var_name.  
FILTER OFF.
```

FILTER allows a boolean-valued variable to be used to select cases from the data stream for processing.

To set up filtering, specify BY and a variable name. Keyword BY is optional but recommended. Cases which have a zero or system- or user-missing value are excluded from analysis, but not deleted from the data stream. Cases with other values are analyzed. To filter based on a different condition, use transformations such as COMPUTE or RECODE to compute a filter variable of the required form, then specify that variable on FILTER.

FILTER OFF turns off case filtering.

Filtering takes place immediately before cases pass to a procedure for analysis. Only one filter variable may be active at a time. Normally, case filtering continues until it is explicitly turned off with FILTER OFF. However, if FILTER is placed after TEMPORARY, it filters only the next procedure or procedure-like command.

10.2 N OF CASES

```
N [OF CASES] num_of_cases [ESTIMATED].
```

Sometimes you may want to disregard cases of your input. N can do this. N 100 tells PSPP to disregard all cases after the first 100.

If the value specified for N is greater than the number of cases read in, the value is ignored.

N does not discard cases or prevent them from being read. It just causes cases beyond the last one specified to be ignored by data analysis commands.

A later N command can increase or decrease the number of cases selected. (To select all the cases without knowing how many there are, specify a very high number: 100000 or whatever you think is large enough.)

Transformation procedures performed after N is executed *do* cause cases to be discarded.

SAMPLE, PROCESS IF, and SELECT IF have precedence over N—the same results are obtained by both of the following fragments, given the same random number seeds:

```
...set up, read in data...  
N 100.  
SAMPLE .5.  
...analyze data...  
  
...set up, read in data...  
SAMPLE .5.  
N 100.
```


`...analyze data...`

Both fragments above first randomly sample approximately half of the cases, then select the first 100 of those sampled.

N with the **ESTIMATED** keyword gives an estimated number of cases before DATA LIST or another command to read in data. **ESTIMATED** never limits the number of cases processed by procedures. PSPP currently does not make use of case count estimates.

When N is specified after **TEMPORARY**, it affects only the next procedure (see [Section 10.7 \[TEMPORARY\]](#), page 75).

10.3 PROCESS IF

PROCESS IF *expression*.

PROCESS IF temporarily eliminates cases from the data stream. Its effects are active only through the execution of the next procedure or procedure-like command.

Specify a boolean expression (see [Chapter 5 \[Expressions\]](#), page 21). If the value of the expression is true for a particular case, the case will be analyzed. If the expression has a false or missing value, then the case will be deleted from the data stream for this procedure only.

Regardless of its placement relative to other commands, **PROCESS IF** always takes effect immediately before data passes to the procedure. Only one **PROCESS IF** command may be in effect at any given time.

The effects of **PROCESS IF** are similar, but not identical, to the effects of executing **TEMPORARY**, then **SELECT IF** (see [Section 10.5 \[SELECT IF\]](#), page 74).

The filtering performed by **PROCESS IF** takes place immediately before cases pass to a procedure for analysis. Because **PROCESS IF** affects only a single procedure, its placement relative to **TEMPORARY** is unimportant.

PROCESS IF is deprecated. It is included for compatibility with old command files. New syntax files should use **SELECT IF** or **FILTER** instead.

10.4 SAMPLE

SAMPLE *num1* [**FROM** *num2*].

SAMPLE randomly samples a proportion of the cases in the active file. Unless it follows **TEMPORARY**, it operates as a transformation, permanently removing cases from the active file.

The proportion to sample can be expressed as a single number between 0 and 1. If *k* is the number specified, and *N* is the number of currently-selected cases in the active file, then after **SAMPLE k.**, approximately *k***N* cases will be selected.

The proportion to sample can also be specified in the style **SAMPLE m FROM N**. With this style, cases are selected as follows:

1. If *N* is equal to the number of currently-selected cases in the active file, exactly *m* cases will be selected.
2. If *N* is greater than the number of currently-selected cases in the active file, an equivalent proportion of cases will be selected.

3. If N is less than the number of currently-selected cases in the active, exactly m cases will be selected *from the first N cases in the active file*.

SAMPLE and SELECT IF are performed in the order specified by the syntax file.

SAMPLE is always performed before N OF CASES, regardless of ordering in the syntax file (see [Section 10.2 \[N OF CASES\]](#), page 72).

The same values for SAMPLE may result in different samples. To obtain the same sample, use the SET command to set the random number seed to the same value before each SAMPLE. Different samples may still result when the file is processed on systems with differing endianness or floating-point formats. By default, the random number seed is based on the system time.

10.5 SELECT IF

SELECT IF expression.

SELECT IF selects cases for analysis based on the value of a boolean expression. Cases not selected are permanently eliminated from the active file, unless TEMPORARY is in effect (see [Section 10.7 \[TEMPORARY\]](#), page 75).

Specify a boolean expression (see [Chapter 5 \[Expressions\]](#), page 21). If the value of the expression is true for a particular case, the case will be analyzed. If the expression has a false or missing value, then the case will be deleted from the data stream.

Place SELECT IF as early in the command file as possible. Cases that are deleted early can be processed more efficiently in time and space.

When SELECT IF is specified following TEMPORARY (see [Section 10.7 \[TEMPORARY\]](#), page 75), the LAG function may not be used (see [\[LAG\]](#), page 31).

10.6 SPLIT FILE

SPLIT FILE [{LAYERED, SEPARATE}] BY var_list.
SPLIT FILE OFF.

SPLIT FILE allows multiple sets of data present in one data file to be analyzed separately using single statistical procedure commands.

Specify a list of variable names to analyze multiple sets of data separately. Groups of adjacent cases having the same values for these variables are analyzed by statistical procedure commands as one group. An independent analysis is carried out for each group of cases, and the variable values for the group are printed along with the analysis.

When a list of variable names is specified, one of the keywords LAYERED or SEPARATE may also be specified. If provided, either keyword are ignored.

Groups are formed only by *adjacent* cases. To create a split using a variable where like values are not adjacent in the working file, you should first sort the data by that variable (see [Section 9.8 \[SORT CASES\]](#), page 71).

Specify OFF to disable SPLIT FILE and resume analysis of the entire active file as a single group of data.

When SPLIT FILE is specified after TEMPORARY, it affects only the next procedure (see [Section 10.7 \[TEMPORARY\]](#), page 75).

10.7 TEMPORARY

TEMPORARY.

TEMPORARY is used to make the effects of transformations following its execution temporary. These transformations will affect only the execution of the next procedure or procedure-like command. Their effects will not be saved to the active file.

The only specification on TEMPORARY is the command name.

TEMPORARY may not appear within a DO IF or LOOP construct. It may appear only once between procedures and procedure-like commands.

Scratch variables cannot be used following TEMPORARY.

An example may help to clarify:

```
DATA LIST /X 1-2.
BEGIN DATA.
  2
  4
10
15
20
24
END DATA.
COMPUTE X=X/2.
TEMPORARY.
COMPUTE X=X+3.
DESCRIPTIVES X.
DESCRIPTIVES X.
```

The data read by the first DESCRIPTIVES are 4, 5, 8, 10.5, 13, 15. The data read by the first DESCRIPTIVES are 1, 2, 5, 7.5, 10, 12.

10.8 WEIGHT

WEIGHT BY var_name.
WEIGHT OFF.

WEIGHT assigns cases varying weights, changing the frequency distribution of the active file. Execution of WEIGHT is delayed until data have been read.

If a variable name is specified, WEIGHT causes the values of that variable to be used as weighting factors for subsequent statistical procedures. Use of keyword BY is optional but recommended. Weighting variables must be numeric. Scratch variables may not be used for weighting (see [Section 4.6.5 \[Scratch Variables\]](#), page 19).

When OFF is specified, subsequent statistical procedures will weight all cases equally.

A positive integer weighting factor w on a case will yield the same statistical output as would replicating the case w times. A weighting factor of 0 is treated for statistical purposes as if the case did not exist in the input. Weighting values need not be integers, but negative and system-missing values for the weighting variable are interpreted as weighting factors of 0. User-missing values are not treated specially.

When WEIGHT is specified after TEMPORARY, it affects only the next procedure (see [Section 10.7 \[TEMPORARY\]](#), page 75).

WEIGHT does not cause cases in the active file to be replicated in memory.

11 Conditional and Looping Constructs

This chapter documents PSPP commands used for conditional execution, looping, and flow of control.

11.1 BREAK

BREAK.

BREAK terminates execution of the innermost currently executing LOOP construct.

BREAK is allowed only inside LOOP...END LOOP. See [Section 11.4 \[LOOP\]](#), page 78, for more details.

11.2 DO IF

DO IF condition.

```

    ...
    [ELSE IF condition.
        ...
    ]...
    [ELSE.
        ...]
    END IF.
```

DO IF allows one of several sets of transformations to be executed, depending on user-specified conditions.

If the specified boolean expression evaluates as true, then the block of code following DO IF is executed. If it evaluates as missing, then none of the code blocks is executed. If it is false, then the boolean expression on the first ELSE IF, if present, is tested in turn, with the same rules applied. If all expressions evaluate to false, then the ELSE code block is executed, if it is present.

When DO IF or ELSE IF is specified following TEMPORARY (see [Section 10.7 \[TEMPORARY\]](#), page 75), the LAG function may not be used (see [\[LAG\]](#), page 31).

11.3 DO REPEAT

```

DO REPEAT repvar_name=expansion...
    ...
END REPEAT [PRINT].
```

expansion takes one of the following forms:

```

var_list
num_or_range...
'string'...
```

num_or_range takes one of the following forms:

```

number
num1 TO num2
```

DO REPEAT repeats a block of code, textually substituting different variables, numbers, or strings into the block with each repetition.

Specify a repeat variable name followed by an equals sign ('=') and the list of replacements. Replacements can be a list of variables (which may be existing variables or new variables or a combination thereof), of numbers, or of strings. When new variable names are specified, DO REPEAT creates them as numeric variables. When numbers are specified, runs of integers may be indicated with TO notation, for instance '1 TO 5' and '1 2 3 4 5' would be equivalent. There is no equivalent notation for string values.

Multiple repeat variables can be specified. When this is done, each variable must have the same number of replacements.

The code within DO REPEAT is repeated as many times as there are replacements for each variable. The first time, the first value for each repeat variable is substituted; the second time, the second value for each repeat variable is substituted; and so on.

Repeat variable substitutions work like macros. They take place anywhere in a line that the repeat variable name occurs as a token, including command and subcommand names. For this reason it is not a good idea to select words commonly used in command and subcommand names as repeat variable identifiers.

If PRINT is specified on END REPEAT, the commands after substitutions are made are printed to the listing file, prefixed by a plus sign ('+').

11.4 LOOP

```
LOOP [index_var=start TO end [BY incr]] [IF condition].
```

```
...  
END LOOP [IF condition].
```

LOOP iterates a group of commands. A number of termination options are offered.

Specify index_var to make that variable count from one value to another by a particular increment. index_var must be a pre-existing numeric variable. start, end, and incr are numeric expressions (see [Chapter 5 \[Expressions\]](#), [page 21](#).)

During the first iteration, index_var is set to the value of start. During each successive iteration, index_var is increased by the value of incr. If end > start, then the loop terminates when index_var > end; otherwise it terminates when index_var < end. If incr is not specified then it defaults to +1 or -1 as appropriate.

If end > start and incr < 0, or if end < start and incr > 0, then the loop is never executed. index_var is nevertheless set to the value of start.

Modifying index_var within the loop is allowed, but it has no effect on the value of index_var in the next iteration.

Specify a boolean expression for the condition on LOOP to cause the loop to be executed only if the condition is true. If the condition is false or missing before the loop contents are executed the first time, the loop contents are not executed at all.

If index and condition clauses are both present on LOOP, the index clause is always evaluated first.

Specify a boolean expression for the condition on END LOOP to cause the loop to terminate if the condition is not true after the enclosed code block is executed. The condition is evaluated at the end of the loop, not at the beginning.

If the index clause and both condition clauses are not present, then the loop is executed MXLOOPS (see [Section 13.15 \[SET\], page 91](#)) times.

BREAK also terminates LOOP execution (see [Section 11.1 \[BREAK\], page 77](#)).

Loop index variables are by default reset to system-missing from one case to another, not left, unless a scratch variable is used as index. When loops are nested, this is usually undesired behavior, which can be corrected with LEAVE (see [Section 8.5 \[LEAVE\], page 59](#)) or by using a scratch variable as the loop index.

When LOOP or END LOOP is specified following TEMPORARY (see [Section 10.7 \[TEMPORARY\], page 75](#)), the LAG function may not be used (see [\[LAG\], page 31](#)).

12 Statistics

This chapter documents the statistical procedures that PSPP supports so far.

12.1 DESCRIPTIVES

DESCRIPTIVES

```

/VARIABLES=var_list
/MISSING={VARIABLE,LISTWISE} {INCLUDE,NOINCLUDE}
/FORMAT={LABELS,NOLABELS} {NOINDEX,INDEX} {LINE,SERIAL}
/SAVE
/STATISTICS={ALL,MEAN,SEMEAN,STDDEV,VARIANCE,KURTOSIS,
             SKEWNESS,RANGE,MINIMUM,MAXIMUM,SUM,DEFAULT,
             SESKEWNESS,SEKURTOSIS}
/SORT={NONE,MEAN,SEMEAN,STDDEV,VARIANCE,KURTOSIS,SKEWNESS,
       RANGE,MINIMUM,MAXIMUM,SUM,SESKEWNESS,SEKURTOSIS,NAME}
{A,D}

```

The DESCRIPTIVES procedure reads the active file and outputs descriptive statistics requested by the user. In addition, it can optionally compute Z-scores.

The VARIABLES subcommand, which is required, specifies the list of variables to be analyzed. Keyword VARIABLES is optional.

All other subcommands are optional:

The MISSING subcommand determines the handling of missing variables. If INCLUDE is set, then user-missing values are included in the calculations. If NOINCLUDE is set, which is the default, user-missing values are excluded. If VARIABLE is set, then missing values are excluded on a variable by variable basis; if LISTWISE is set, then the entire case is excluded whenever any value in that case has a system-missing or, if INCLUDE is set, user-missing value.

The FORMAT subcommand affects the output format. Currently the LABELS/NOLABELS and NOINDEX/INDEX settings are not used. When SERIAL is set, both valid and missing number of cases are listed in the output; when NOSERIAL is set, only valid cases are listed.

The SAVE subcommand causes DESCRIPTIVES to calculate Z scores for all the specified variables. The Z scores are saved to new variables. Variable names are generated by trying first the original variable name with Z prepended and truncated to a maximum of 8 characters, then the names ZSC000 through ZSC999, STDZ00 through STDZ09, ZZZZ00 through ZZZZ09, ZQZQ00 through ZQZQ09, in that sequence. In addition, Z score variable names can be specified explicitly on VARIABLES in the variable list by enclosing them in parentheses after each variable.

The STATISTICS subcommand specifies the statistics to be displayed:

ALL	All of the statistics below.
MEAN	Arithmetic mean.
SEMEAN	Standard error of the mean.
STDDEV	Standard deviation.

VARIANCE	Variance.
KURTOSIS	Kurtosis and standard error of the kurtosis.
SKEWNESS	Skewness and standard error of the skewness.
RANGE	Range.
MINIMUM	Minimum value.
MAXIMUM	Maximum value.
SUM	Sum.
DEFAULT	Mean, standard deviation of the mean, minimum, maximum.
SEKURTOSIS	Standard error of the kurtosis.
SESKWNESS	Standard error of the skewness.

The SORT subcommand specifies how the statistics should be sorted. Most of the possible values should be self-explanatory. NAME causes the statistics to be sorted by name. By default, the statistics are listed in the order that they are specified on the VARIABLES subcommand. The A and D settings request an ascending or descending sort order, respectively.

12.2 FREQUENCIES

```

FREQUENCIES
  /VARIABLES=var_list
  /FORMAT={TABLE,NOTABLE,LIMIT(limit)}
           {STANDARD,CONDENSE,ONEPAGE[(onepage_limit)]}
           {LABELS,NOLABELS}
           {AVALUE,DVALUE,AFREQ,DFREQ}
           {SINGLE,DOUBLE}
           {OLDPAGE,NEWPAGE}
  /MISSING={EXCLUDE,INCLUDE}
  /STATISTICS={DEFAULT,MEAN,SEMEAN,MEDIAN,MODE,STDDEV,VARIANCE,
              KURTOSIS,SKEWNESS,RANGE,MINIMUM,MAXIMUM,SUM,
              SESKEWNESS,SEKURTOSIS,ALL,NONE}
  /NTILES=ntiles
  /PERCENTILES=percent...

```

(These options are not currently implemented.)

```

  /BARCHART=...
  /HISTOGRAM=...
  /HBAR=...
  /GROUPED=...

```

(Integer mode.)

```

  /VARIABLES=var_list (low,high)...

```

The FREQUENCIES procedure outputs frequency tables for specified variables. FREQUENCIES can also calculate and display descriptive statistics (including median and mode) and percentiles.

In the future, FREQUENCIES will also support graphical output in the form of bar charts and histograms. In addition, it will be able to support percentiles for grouped data.

The VARIABLES subcommand is the only required subcommand. Specify the variables to be analyzed. In most cases, this is all that is required. This is known as *general mode*.

Occasionally, one may want to invoke a special mode called *integer mode*. Normally, in general mode, PSPP will automatically determine what values occur in the data. In integer mode, the user specifies the range of values that the data assumes. To invoke this mode, specify a range of data values in parentheses, separated by a comma. Data values inside the range are truncated to the nearest integer, then assigned to that value. If values occur outside this range, they are discarded.

The FORMAT subcommand controls the output format. It has several possible settings:

- TABLE, the default, causes a frequency table to be output for every variable specified. NOTABLE prevents them from being output. LIMIT with a numeric argument causes them to be output except when there are more than the specified number of values in the table.
- STANDARD frequency tables contain more complete information, but also to take up more space on the printed page. CONDENSE frequency tables are less informative but take up less space. ONEPAGE with a numeric argument will output standard frequency tables if there are the specified number of values or less, condensed tables otherwise. ONEPAGE without an argument defaults to a threshold of 50 values.
- LABELS causes value labels to be displayed in STANDARD frequency tables. NO-LABELS prevents this.
- Normally frequency tables are sorted in ascending order by value. This is AVALUE. DVALUE tables are sorted in descending order by value. AFREQ and DFREQ tables are sorted in ascending and descending order, respectively, by frequency count.
- SINGLE spaced frequency tables are closely spaced. DOUBLE spaced frequency tables have wider spacing.
- OLDPAGE and NEWPAGE are not currently used.

The MISSING subcommand controls the handling of user-missing values. When EXCLUDE, the default, is set, user-missing values are not included in frequency tables or statistics. When INCLUDE is set, user-missing are included. System-missing values are never included in statistics, but are listed in frequency tables.

The available STATISTICS are the same as available in DESCRIPTIVES (see [Section 12.1 \[DESCRIPTIVES\]](#), page 80), with the addition of MEDIAN, the data's median value, and MODE, the mode. (If there are multiple modes, the smallest value is reported.) By default, the mean, standard deviation of the mean, minimum, and maximum are reported for each variable.

PERCENTILES causes the specified percentiles to be reported. The percentiles should be presented at a list of numbers between 0 and 100 inclusive. The NTILES subcommand causes the percentiles to be reported at the boundaries of the data set divided into the specified number of ranges. For instance, /NTILES=4 would cause quartiles to be reported.

12.3 EXAMINE

```

EXAMINE
  VARIABLES=var_list [BY factor_list ]
  /STATISTICS={DESCRIPTIVES, EXTREME[(n)], ALL, NONE}
  /PLOT={STEMLEAF, BOXPLOT, NPLOT, SPREADLEVEL(n), HISTOGRAM,
  ALL, NONE}
  /CINTERVAL n
  /COMPARE={GROUPS,VARIABLES}
  /ID={case_number, var_name}
  /{TOTAL,NOTOTAL}
  /PERCENTILE=[value_list]={HAVERAGE, WAVERAGE, ROUND, AEM-
  PIRICAL, EMPIRICAL }
  /MISSING={LISTWISE, PAIRWISE} [{EXCLUDE, INCLUDE}]
  [{NOREPORT,REPORT}]

```

The EXAMINE command is used to test how closely a distribution is to a normal distribution. It also shows you outliers and extreme values.

The VARIABLES subcommand specifies the dependent variables and the independent variable to use as factors for the analysis. Variables listed before the first BY keyword are the dependent variables. The dependent variables may optionally be followed by a list of factors which tell PSPP how to break down the analysis for each dependent variable. The format for each factor is

```
var [BY var].
```

The STATISTICS subcommand specifies the analysis to be done. DESCRIPTIVES will produce a table showing some parametric and non-parametrics statistics. EXTREME produces a table showing extreme values of the dependent variable. A number in parentheses determines how many upper and lower extremes to show. The default number is 5.

The PLOT subcommand specifies which plots are to be produced if any.

The COMPARE subcommand is only relevant if producing boxplots, and it is only useful there is more than one dependent variable and at least one factor. If /COMPARE=GROUPS is specified, then one plot per dependent variable is produced, containing boxplots for all the factors. If /COMPARE=VARIABLES is specified, then one plot per factor is produced, each each containing one boxplot per dependent variable. If the /COMPARE subcommand is omitted, then PSPP uses the default value of /COMPARE=GROUPS.

The CINTERVAL subcommand specifies the confidence interval to use in calculation of the descriptives command. The default is 95%.

The PERCENTILES subcommand specifies which percentiles are to be calculated, and which algorithm to use for calculating them. The default is to calculate the 5, 10, 25, 50, 75, 90, 95 percentiles using the HAVERAGE algorithm.

The TOTAL and NOTOTAL subcommands are mutually exclusive. If NOTOTAL is given and factors have been specified in the VARIABLES subcommand, then statistics for the unfactored dependent variables are produced in addition to the factored variables. If there are no factors specified then TOTAL and NOTOTAL have no effect.

Warning! If many dependent variable are given, or factors are given for which there are many distinct values, then EXAMINE will produce a very large quantity of output.

12.4 CROSSTABS

CROSSTABS

```

/TABLES=var_list BY var_list [BY var_list] . . .
/MISSING={TABLE,INCLUDE,REPORT}
/WRITE={NONE,CELLS,ALL}
/FORMAT={TABLES,NOTABLES}
        {LABELS,NOLABELS,NOVALLABS}
        {PIVOT,NOPIVOT}
        {AVALUE,DVALUE}
        {NOINDEX,INDEX}
        {BOX,NOBOX}
/CELLS={COUNT,ROW,COLUMN,TOTAL,EXPECTED,RESIDUAL,SRESIDUAL,
        ASRESIDUAL,ALL,NONE}
/STATISTICS={CHISQ,PHI,CC,LAMBDA,UC,BTAU,CTAU,RISK,GAMMA,D,
        KAPPA,ETA,CORR,ALL,NONE}

```

(Integer mode.)

```

/VARIABLES=var_list (low,high) . . .

```

The CROSSTABS procedure displays crosstabulation tables requested by the user. It can calculate several statistics for each cell in the crosstabulation tables. In addition, a number of statistics can be calculated for each table itself.

The TABLES subcommand is used to specify the tables to be reported. Any number of dimensions is permitted, and any number of variables per dimension is allowed. The TABLES subcommand may be repeated as many times as needed. This is the only required subcommand in *general mode*.

Occasionally, one may want to invoke a special mode called *integer mode*. Normally, in general mode, PSPP automatically determines what values occur in the data. In integer mode, the user specifies the range of values that the data assumes. To invoke this mode, specify the VARIABLES subcommand, giving a range of data values in parentheses for each variable to be used on the TABLES subcommand. Data values inside the range are truncated to the nearest integer, then assigned to that value. If values occur outside this range, they are discarded. When it is present, the VARIABLES subcommand must precede the TABLES subcommand.

In general mode, numeric and string variables may be specified on TABLES. Although long string variables are allowed, only their initial short-string parts are used. In integer mode, only numeric variables are allowed.

The MISSING subcommand determines the handling of user-missing values. When set to TABLE, the default, missing values are dropped on a table by table basis. When set to INCLUDE, user-missing values are included in tables and statistics. When set to REPORT, which is allowed only in integer mode, user-missing values are included in tables but marked with an 'M' (for "missing") and excluded from statistical calculations.

Currently the WRITE subcommand is ignored.

The `FORMAT` subcommand controls the characteristics of the crosstabulation tables to be displayed. It has a number of possible settings:

- `TABLES`, the default, causes crosstabulation tables to be output. `NOTABLES` suppresses them.
- `LABELS`, the default, allows variable labels and value labels to appear in the output. `NOLABELS` suppresses them. `NOVALLABS` displays variable labels but suppresses value labels.
- `PIVOT`, the default, causes each `TABLES` subcommand to be displayed in a pivot table format. `NOPIVOT` causes the old-style crosstabulation format to be used.
- `AVALUE`, the default, causes values to be sorted in ascending order. `DVALUE` asserts a descending sort order.
- `INDEX/NOINDEX` is currently ignored.
- `BOX/NOBOX` is currently ignored.

The `CELLS` subcommand controls the contents of each cell in the displayed crosstabulation table. The possible settings are:

<code>COUNT</code>	Frequency count.
<code>ROW</code>	Row percent.
<code>COLUMN</code>	Column percent.
<code>TOTAL</code>	Table percent.
<code>EXPECTED</code>	Expected value.
<code>RESIDUAL</code>	Residual.
<code>SRESIDUAL</code>	Standardized residual.
<code>ASRESIDUAL</code>	Adjusted standardized residual.
<code>ALL</code>	All of the above.
<code>NONE</code>	Suppress cells entirely.

`/CELLS` without any settings specified requests `COUNT`, `ROW`, `COLUMN`, and `TOTAL`. If `CELLS` is not specified at all then only `COUNT` will be selected.

The `STATISTICS` subcommand selects statistics for computation:

<code>CHISQ</code>	Pearson chi-square, likelihood ratio, Fisher's exact test, continuity correction, linear-by-linear association.
<code>PHI</code>	Phi.
<code>CC</code>	Contingency coefficient.
<code>LAMBDA</code>	Lambda.
<code>UC</code>	Uncertainty coefficient.

BTAU	Tau-b.
CTAU	Tau-c.
RISK	Risk estimate.
GAMMA	Gamma.
D	Somers' D.
KAPPA	Cohen's Kappa.
ETA	Eta.
CORR	Spearman correlation, Pearson's r.
ALL	All of the above.
NONE	No statistics.

Selected statistics are only calculated when appropriate for the statistic. Certain statistics require tables of a particular size, and some statistics are calculated only in integer mode.

'/STATISTICS' without any settings selects CHISQ. If the STATISTICS subcommand is not given, no statistics are calculated.

Please note: Currently the implementation of CROSSTABS has the followings bugs:

- Pearson's R (but not Spearman) is off a little.
- T values for Spearman's R and Pearson's R are wrong.
- Significance of symmetric and directional measures is not calculated.
- Asymmetric ASEs and T values for lambda are wrong.
- ASE of Goodman and Kruskal's tau is not calculated.
- ASE of symmetric somers' d is wrong.
- Approximate T of uncertainty coefficient is wrong.

Fixes for any of these deficiencies would be welcomed.

12.5 T-TEST

T-TEST

```
/MISSING={ANALYSIS,LISTWISE} {EXCLUDE,INCLUDE}
/CRITERIA=CIN(confidence)
```

(One Sample mode.)

```
TESTVAL=test_value
/VARIABLES=var_list
```

(Independent Samples mode.)

```
GROUPS=var(value1 [, value2])
/VARIABLES=var_list
```

(Paired Samples mode.)

```
PAIRS=var_list [WITH var_list [(PAIRED)] ]
```

The T-TEST procedure outputs tables used in testing hypotheses about means. It operates in one of three modes:

- One Sample mode.
- Independent Groups mode.
- Paired mode.

Each of these modes are described in more detail below. There are two optional subcommands which are common to all modes.

The /CRITERIA subcommand tells PSPP the confidence interval used in the tests. The default value is 0.95.

The MISSING subcommand determines the handling of missing variables. If INCLUDE is set, then user-missing values are included in the calculations, but system-missing values are not. If EXCLUDE is set, which is the default, user-missing values are excluded as well as system-missing values. This is the default.

If LISTWISE is set, then the entire case is excluded from analysis whenever any variable specified in the /VARIABLES, /PAIRS or /GROUPS subcommands contains a missing value. If ANALYSIS is set, then missing values are excluded only in the analysis for which they would be needed. This is the default.

12.5.1 One Sample Mode

The TESTVAL subcommand invokes the One Sample mode. This mode is used to test a population mean against a hypothesised mean. The value given to the TESTVAL subcommand is the value against which you wish to test. In this mode, you must also use the /VARIABLES subcommand to tell PSPP which variables you wish to test.

12.5.2 Independent Samples Mode

The GROUPS subcommand invokes Independent Samples mode or ‘Groups’ mode. This mode is used to test whether two groups of values have the same population mean. In this mode, you must also use the /VARIABLES subcommand to tell PSPP the dependent variables you wish to test.

The variable given in the GROUPS subcommand is the independent variable which determines to which group the samples belong. The values in parentheses are the specific values of the independent variable for each group. If the parentheses are omitted and no values are given, the default values of 1.0 and 2.0 are assumed.

If the independent variable is numeric, it is acceptable to specify only one value inside the parentheses. If you do this, cases where the independent variable is less than or equal to this value belong to the first group, and cases greater than this value belong to the second group. When using this form of the GROUPS subcommand, missing values in the independent variable are excluded on a listwise basis, regardless of whether /MISSING=LISTWISE was specified.

12.5.3 Paired Samples Mode

The PAIRS subcommand introduces Paired Samples mode. Use this mode when repeated measures have been taken from the same samples. If the the WITH keyword is omitted, then tables for all combinations of variables given in the PAIRS subcommand are generated. If the WITH keyword is given, and the (PAIRED) keyword is also given, then the number of variables preceding WITH must be the same as the number following it. In this case, tables for each respective pair of variables are generated. In the event that the WITH keyword is given, but the (PAIRED) keyword is omitted, then tables for each combination of variable preceding WITH against variable following WITH are generated.

12.6 ONEWAY

ONEWAY

```
[/VARIABLES = ] var_list BY var
/MISSING={ANALYSIS,LISTWISE} {EXCLUDE,INCLUDE}
/CONTRASTS= value1 [, value2] ... [,valueN]
/STATISTICS={DESCRIPTIVES,HOMOGENEITY}
```

The ONEWAY procedure performs a one-way analysis of variance of variables factored by a single independent variable. It is used to compare the means of a population divided into more than two groups.

The variables to be analysed should be given in the **VARIABLES** subcommand. The list of variables must be followed by the **BY** keyword and the name of the independent (or factor) variable.

You can use the **STATISTICS** subcommand to tell PSPP to display ancilliary information. The options accepted are:

- **DESCRIPTIVES** Displays descriptive statistics about the groups factored by the independent variable.
- **HOMOGENEITY** Displays the Levene test of Homogeneity of Variance for the variables and their groups.

The **CONTRASTS** subcommand is used when you anticipate certain differences between the groups. The subcommand must be followed by a list of numerals which are the coefficients of the groups to be tested. The number of coefficients must correspond to the number of distinct groups (or values of the independent variable). If the total sum of the coefficients are not zero, then PSPP will display a warning, but will proceed with the analysis. The **CONTRASTS** subcommand may be given up to 10 times in order to specify different contrast tests.

13 Utilities

Commands that don't fit any other category are placed here.

Most of these commands are not affected by commands like IF and LOOP: they take effect only once, unconditionally, at the time that they are encountered in the input.

13.1 COMMENT

Two possible syntaxes:

COMMENT comment text

*comment text

COMMENT is ignored. It is used to provide information to the author and other readers of the PSPP syntax file.

COMMENT can extend over any number of lines. Don't forget to terminate it with a dot or a blank line.

13.2 DOCUMENT

DOCUMENT documentary_text.

DOCUMENT adds one or more lines of descriptive commentary to the active file. Documents added in this way are saved to system files. They can be viewed using SYSFILE INFO or DISPLAY DOCUMENTS. They can be removed from the active file with DROP DOCUMENTS.

Specify the documentary text following the DOCUMENT keyword. You can extend the documentary text over as many lines as necessary. Lines are truncated at 80 characters width. Don't forget to terminate the command with a dot or a blank line.

13.3 DISPLAY DOCUMENTS

DISPLAY DOCUMENTS.

DISPLAY DOCUMENTS displays the documents in the active file. Each document is preceded by a line giving the time and date that it was added. See [Section 13.2 \[DOCUMENT\]](#), page 89.

13.4 DISPLAY FILE LABEL

DISPLAY FILE LABEL.

DISPLAY FILE LABEL displays the file label contained in the active file, if any. See [Section 13.9 \[FILE LABEL\]](#), page 90.

13.5 DROP DOCUMENTS

DROP DOCUMENTS.

DROP DOCUMENTS removes all documents from the active file. New documents can be added with DOCUMENT (see [Section 13.2 \[DOCUMENT\]](#), page 89).

DROP DOCUMENTS changes only the active file. It does not modify any system files stored on disk.

13.6 ECHO

ECHO 'arbitrary text' .

Use ECHO to write arbitrary text to the output stream. The text should be enclosed in quotation marks following the normal rules for string tokens (see [Section 4.1 \[Tokens\]](#), page 8).

13.7 ERASE

ERASE FILE file_name.

ERASE FILE deletes a file from the local filesystem. file_name must be quoted. This command cannot be used if the SAFER setting is active.

13.8 EXECUTE

EXECUTE.

EXECUTE causes the active file to be read and all pending transformations to be executed.

13.9 FILE LABEL

FILE LABEL file_label.

FILE LABEL provides a title for the active file. This title will be saved into system files and portable files that are created during this PSPP run.

file_label need not be quoted. If quotes are included, they become part of the file label.

13.10 FINISH

FINISH.

FINISH terminates the current PSPP session and returns control to the operating system.

This command is not valid in interactive mode.

13.11 HOST

HOST.

HOST suspends the current PSPP session and temporarily returns control to the operating system. This command cannot be used if the SAFER setting is active.

13.12 INCLUDE

Two possible syntaxes:

INCLUDE 'filename'.
@filename.

INCLUDE causes the PSPP command processor to read an additional command file as if it were included bodily in the current command file.

Include files may be nested to any depth, up to the limit of available memory.

13.13 PERMISSIONS

PERMISSIONS

FILE='filename'

/PERMISSIONS = {READONLY,WRITEABLE}.

PERMISSIONS changes the permissions of a file. There is one mandatory subcommand which specifies the permissions to which the file should be changed. If you set a file's permission to READONLY, then the file will become unwritable either by you or anyone else on the system. If you set the permission to WRITEABLE, then the file will become writeable by you; the permissions afforded to others will be unchanged. This command cannot be used if the SAFER setting is active.

13.14 QUIT

Two possible syntaxes:

QUIT.

EXIT.

QUIT terminates the current PSPP session and returns control to the operating system. This command is not valid within a command file.

13.15 SET

SET

(data input)

/BLANKS={SYSMIS,'.',number}

/DECIMAL={DOT,COMMA}

/FORMAT=fmt_spec

/EPOCH={AUTOMATIC,year}

(program input)

/ENDCMD='.'

/NULLLINE={ON,OFF}

(interaction)

/CPROMPT='cprompt_string'

/DPROMPT='dprompt_string'

/ERRORBREAK={OFF,ON}

/MXERRS=max_errs

/MXWARNS=max_warnings

/PROMPT='prompt'

/VIEWLENGTH={MINIMUM,MEDIAN,MAXIMUM,n_lines}

/VIEWWIDTH=n_characters

(program execution)

/MEXPAND={ON,OFF}

/MITERATE=max_iterations

/MNEST=max_nest

```

/MPRINT={ON,OFF}
/MXLOOPS=max_loops
/SEED={RANDOM,seed_value}
/UNDEFINED={WARN,NOWARN}

(data output)
/CC{A,B,C,D,E}={'npre,pre,suf,nsuf','npre.pre.suf.nsuf'}
/DECIMAL={DOT,COMMA}
/FORMAT=fmt_spec

(output routing)
/ECHO={ON,OFF}
/ERRORS={ON,OFF,TERMINAL,LISTING,BOTH,NONE}
/INCLUDE={ON,OFF}
/MESSAGES={ON,OFF,TERMINAL,LISTING,BOTH,NONE}
/PRINTBACK={ON,OFF}
/RESULTS={ON,OFF,TERMINAL,LISTING,BOTH,NONE}

(output activation)
/LISTING={ON,OFF}
/PRINTER={ON,OFF}
/SCREEN={ON,OFF}

(output driver options)
/HEADERS={NO,YES,BLANK}
/LENGTH={NONE,length_in_lines}
/LISTING=filename
/MORE={ON,OFF}
/PAGER={OFF,"pager_name"}
/WIDTH={NARROW,WIDTH,n_characters}

(logging)
/JOURNAL={ON,OFF} [filename]
/LOG={ON,OFF} [filename]

(system files)
/COMPRESSION={ON,OFF}
/SCOMPRESSON={ON,OFF}

(security)
/SAFER=ON

(obsolete settings accepted for compatibility, but ignored)
/AUTOMENU={ON,OFF}
/BEEP={ON,OFF}
/BLOCK='c'
/BOXSTRING={'xxx','xxxxxxxxxxxx'}

```

```

/CASE={UPPER,UPLOW}
/COLOR=. . .
/CPI=cpi_value
/DISK={ON,OFF}
/EJECT={ON,OFF}
/HELPWINDOWS={ON,OFF}
/HIGHRES={ON,OFF}
/HISTOGRAM='c'
/LOWRES={AUTO,ON,OFF}
/LPI=lpi_value
/MENUS={STANDARD,EXTENDED}
/MXMEMORY=max_memory
/PTRANSLATE={ON,OFF}
/RCOLORS=. . .
/RUNREVIEW={AUTO,MANUAL}
/SCRIPTTAB='c'
/TB1={'xxx','xxxxxxxxxxx'}
/TBFonts='string'
/WORKDEV=drive_letter
/WORKSPACE=workspace_size
/XSORT={YES,NO}

```

SET allows the user to adjust several parameters relating to PSPP's execution. Since there are many subcommands to this command, its subcommands will be examined in groups.

On subcommands that take boolean values, ON and YES are synonym, and as are OFF and NO, when used as subcommand values.

The data input subcommands affect the way that data is read from data files. The data input subcommands are

BLANKS This is the value assigned to an item data item that is empty or contains only white space. An argument of SYSMIS or '.' will cause the system-missing value to be assigned to null items. This is the default. Any real value may be assigned.

DECIMAL

The default DOT setting causes the decimal point character to be '.'. A setting of COMMA causes the decimal point character to be ','.

FORMAT Allows the default numeric input/output format to be specified. The default is F8.2. See [Section 4.6.4 \[Input/Output Formats\]](#), page 14.

EPOCH Specifies the range of years used when a 2-digit year is read from a data file or used in a date construction expression (see [Section 5.7.8.4 \[Date Construction\]](#), page 29). If a 4-digit year is specified, then 2-digit years are interpreted starting from that year, known as the epoch. If AUTOMATIC (the default) is specified, then the epoch begins 69 years before the current date.

Program input subcommands affect the way that programs are parsed when they are typed interactively or run from a script. They are

ENDCMD This is a single character indicating the end of a command. The default is ‘.’. Don’t change this.

NULLINE Whether a blank line is interpreted as ending the current command. The default is ON.

Interaction subcommands affect the way that PSPP interacts with an online user. The interaction subcommands are

CPROMPT

The command continuation prompt. The default is ‘>’.

DPROMPT

Prompt used when expecting data input within BEGIN DATA (see [Section 6.1 \[BEGIN DATA\]](#), page 39). The default is ‘data>’.

ERRORBREAK

Whether an error causes PSPP to stop processing the current command file after finishing the current command. The default is OFF.

MXERRS The maximum number of errors before PSPP halts processing of the current command file. The default is 50.

MXWARNS

The maximum number of warnings + errors before PSPP halts processing the current command file. The default is 100.

PROMPT The command prompt. The default is ‘PSPP>’.

VIEWLENGTH

The length of the screen in lines. MINIMUM means 25 lines, MEDIAN and MAXIMUM mean 43 lines. Otherwise specify the number of lines. Normally PSPP should auto-detect your screen size so this shouldn’t have to be used.

VIEWWIDTH

The width of the screen in characters. Normally 80 or 132.

Program execution subcommands control the way that PSPP commands execute. The program execution subcommands are

MEXPAND

MITERATE

MNEST

MPRINT Currently not used.

MXLOOPS

The maximum number of iterations for an uncontrolled loop (see [Section 11.4 \[LOOP\]](#), page 78).

SEED The initial pseudo-random number seed. Set to a real number or to RANDOM, which will obtain an initial seed from the current time of day.

UNDEFINED

Currently not used.

Data output subcommands affect the format of output data. These subcommands are

CCA

CCB

CCC

CCD

CCE Set up custom currency formats. The argument is a string which must contain exactly three commas or exactly three periods. If commas, then the grouping character for the currency format is ‘,’ and the decimal point character is ‘.’; if periods, then the situation is reversed.

The commas or periods divide the string into four fields, which are, in order, the negative prefix, prefix, suffix, and negative suffix. When a value is formatted using the custom currency format, the prefix precedes the value formatted and the suffix follows it. In addition, if the value is negative, the negative prefix precedes the prefix and the negative suffix follows the suffix.

DECIMAL

The default DOT setting causes the decimal point character to be ‘.’. A setting of COMMA causes the decimal point character to be ‘,’.

FORMAT Allows the default numeric input/output format to be specified. The default is F8.2. See [Section 4.6.4 \[Input/Output Formats\]](#), page 14.

Output routing subcommands affect where the output of transformations and procedures is sent. These subcommands are

ECHO

If turned on, commands are written to the listing file as they are read from command files. The default is OFF.

ERRORS

INCLUDE

MESSAGES

PRINTBACK

RESULTS Currently not used.

Output activation subcommands affect whether output devices of particular types are enabled. These subcommands are

LISTING Enable or disable listing devices.

PRINTER

Enable or disable printer devices.

SCREEN Enable or disable screen devices.

Output driver option subcommands affect output drivers’ settings. These subcommands are

HEADERS

LENGTH

LISTING

MORE

PAGER

WIDTH

Logging subcommands affect logging of commands executed to external files. These subcommands are

JOURNAL

LOG Not currently used.

System file subcommands affect the default format of system files produced by PSPP. These subcommands are

COMPRESSION

Not currently used.

SCOMPRESSION

Whether system files created by SAVE or XSAVE are compressed by default. The default is ON.

Security subcommands affect the operations that commands are allowed to perform. The security subcommands are

SAFER Setting this option disables the following operations:

- The ERASE command.
- The HOST command.
- The PERMISSIONS command.
- Pipe filenames (filenames beginning or ending with '|').

Be aware that this setting does not guarantee safety (commands can still overwrite files, for instance) but it is an improvement. When set, this setting cannot be reset during the same session, for obvious security reasons.

13.16 SHOW

SHOW

/subcommand

SHOW can be used to display the current state of PSPP's execution parameters. All of the parameters which can be changed using SET See [Section 13.15 \[SET\], page 91](#), can be examined using SHOW, by using a subcommand with the same name. In addition, SHOW supports the following subcommands:

WARRANTY Show details of the lack of warranty for PSPP.

COPYING Display the terms of PSPP's copyright licence [Chapter 2 \[License\], page 2](#).

13.17 SUBTITLE

SUBTITLE 'subtitle_string'.

or

SUBTITLE subtitle_string.

SUBTITLE provides a subtitle to a particular PSPP run. This subtitle appears at the top of each output page below the title, if headers are enabled on the output device.

Specify a subtitle as a string in quotes. The alternate syntax that did not require quotes is now obsolete. If it is used then the subtitle is converted to all uppercase.

13.18 TITLE

TITLE 'title_string'.

or

TITLE title_string.

TITLE provides a title to a particular PSPP run. This title appears at the top of each output page, if headers are enabled on the output device.

Specify a title as a string in quotes. The alternate syntax that did not require quotes is now obsolete. If it is used then the title is converted to all uppercase.

14 Not Implemented

This chapter lists parts of the PSPP language that are not yet implemented.

ACF	Autocorrelation function
ADD FILES	Add files to dictionary
ALSCAL	Multidimensional scaling
ANOVA	Factorial analysis of variance
CASEPLOT	Plot time series
CASESTOVARS	Restructure complex data
CCF	Time series cross correlation
CLUSTER	Hierarchical clustering
CONJOINT	Analyse full concept data
COXREG	Cox proportional hazards regression
CREATE	Create time series data
CURVEFIT	Fit curve to line plot
DATE	Create time series data
DISCRIMINANT	Linear discriminant analysis
EDIT	obsolete
END FILE TYPE	Ends complex data input
FACTOR	Factor analysis
FILE TYPE	Complex data input
FIT	Goodness of Fit
GET TRANSLATE	Read other file formats
GLM	General Linear Model
GRAPH	Draw graphs
IGRAPH	Interactive graphs

INFO Local Documentation

KEYED DATA LIST
 Read nonsequential data

KM Kaplan-Meier

LOGISTIC REGRESSION
 Regression Analysis

MCONVERT
 Convert covariance/correlation matrices

MULT RESPONSE
 Multiple reponse analysis

MVA Missing value analysis

NLR Non Linear Regression

NONPAR CORR
 Nonparametric correlation

NPAR TESTS
 Nonparametric tests

NUMBERED

PACF Partial autocorrelation

PARTIAL CORR
 Partial correlation

POINT Marker in keyed file

PLOT Plot time series variables

PREDICT
 Specify forecast period

PRESERVE
 Push settings

PROCEDURE OUTPUT
 Specify output file

PROBIT Probit analysis

PROXIMITIES
 Pairwise similarity

QUICK CLUSTER
 Fast clustering

RANK Create rank scores

REFORMAT
 Read obsolete files

REGRESSION
 Compute regression coefficients

REPEATING DATA

Specify multiple cases per input record

REPORT Pretty print working file

RESTORE

Restore settings

ROC Receiver operating characteristic

RMV Replace missing values

SAVE TRANSLATE

Save to foreign format

SCRIPT Run script file

SPCHART

Plot control charts

SUMMARIZE

Univariate statistics

SURVIVAL

Survival analysis

TSET Set time sequence variables

TSHOW Show time sequence variables

TSPLOT Plot time sequence variables

UNIANOVA

Univariate analysis

UNNUMBERED

obsolete

UPDATE Update working file

VARSTOCASES

Restructure complex data

VERIFY Report time series

15 Bugs

PSPP does have bugs. We do our best to fix them, but our limited resources mean that some may remain for a long time. Our best alternative is to make you aware of PSPP's known bugs. To see a list, visit PSPP's project webpage at <https://savannah.gnu.org/projects/pspp>. You can also submit your own bug report there: click on “Bugs,” then on “Submit a Bug,” and fill out the form. Alternatively, PSPP bug reports may be sent by email to `<bug-gnu-pspp@gnu.org>`.

For known bugs in individual language features, see the documentation for that feature.

16 Function Index

A

ABS	23
ACOS	24
ANY	25
ARCOS	24
ARSIN	24
ARTAN	24
ASIN	24
ATAN	24

C

CDF.BERNOULLI	37
CDF.BETA	32
CDF.BINOMIAL	37
CDF.CAUCHY	33
CDF.CHISQ	33
CDF.EXP	33
CDF.F	33
CDF.GAMMA	34
CDF.GEOM	37
CDF.HALFNRM	34
CDF.HYPER	37
CDF.IGAUSS	34
CDF.LAPLACE	34
CDF.LNORMAL	35
CDF.LOGISTIC	34
CDF.NEGBIN	37
CDF.NORMAL	35
CDF.PARETO	35
CDF.POISSON	37
CDF.RAYLEIGH	35
CDF.SMOD	36
CDF.SRANGE	36
CDF.T	36
CDF.T1G	36
CDF.T2G	36
CDF.UNIFORM	36
CDF.VBNOR	33
CDF.WEIBULL	36
CDFNORM	35
CFVAR	25
CONCAT	26
COS	24
CTIME.DAYS	29
CTIME.HOURS	29
CTIME.MINUTES	29
CTIME.SECONDS	29

D

DATE.DMY	30
DATE.MDY	30
DATE.MOYR	30

DATE.QYR	30
DATE.WKYR	30
DATE.YRDAY	30

E

EXP	23
-----------	----

I

IDF.BETA	32
IDF.CAUCHY	33
IDF.CHISQ	33
IDF.EXP	33
IDF.F	33
IDF.GAMMA	34
IDF.HALFNRM	34
IDF.IGAUSS	34
IDF.LAPLACE	34
IDF.LNORMAL	35
IDF.LOGISTIC	34
IDF.NORMAL	35
IDF.PARETO	35
IDF.RAYLEIGH	35
IDF.SMOD	36
IDF.SRANGE	36
IDF.T	36
IDF.T1G	36
IDF.T2G	36
IDF.UNIFORM	36
IDF.WEIBULL	36
INDEX	26

L

LAG	31
LENGTH	26
LG10	23
LN	23
LNGAMMA	23
LOWER	26
LPAD	26
LTRIM	26, 27

M

MAX	25
MEAN	25
MIN	25
MISSING	24
MOD	23
MOD10	23

N

NCDF.BETA	33
NCDF.CHISQ	33
NCDF.F	34
NCDF.T	36
NMISS	24
NORMAL	35
NPDF.BETA	33
NPDF.CHISQ	33
NPDF.F	34
NPDF.T	36
NUMBER	27
NVALID	24

P

PDF.BERNOULLI	37
PDF.BETA	32
PDF.BINOMIAL	37
PDF.BVNOR	33
PDF.CAUCHY	33
PDF.CHISQ	33
PDF.EXP	33
PDF.F	33
PDF.GAMMA	34
PDF.GEOM	37
PDF.HALFNRN	34
PDF.HYPER	37
PDF.IGAUSS	34
PDF.LANDAU	34
PDF.LAPLACE	34
PDF.LNORMAL	35
PDF.LOG	37
PDF.LOGISTIC	34
PDF.NEGBIN	37
PDF.NORMAL	35
PDF.NTAIL	35
PDF.PARETO	35
PDF.POISSON	37
PDF.RAYLEIGH	35
PDF.RTAIL	35
PDF.T	36
PDF.T1G	36
PDF.T2G	36
PDF.UNIFORM	36
PDF.WEIBULL	36
PDF.XPOWER	33
PROBIT	35

R

RANGE	25
RINDEX	27
RND	23
RPAD	27
RTRIM	27
RV.BERNOULLI	37
RV.BETA	33

RV.BINOMIAL	37
RV.CAUCHY	33
RV.CHISQ	33
RV.EXP	33
RV.F	33
RV.GAMMA	34
RV.GEOM	37
RV.HALFNRN	34
RV.HYPER	37
RV.IGAUSS	34
RV.LANDAU	34
RV.LAPLACE	34
RV.LEVY	34
RV.LNORMAL	35
RV.LOG	37
RV.LOGISTIC	35
RV.LVSKWE	34
RV.NEGBIN	37
RV.NORMAL	35
RV.NTAIL	35
RV.PARETO	35
RV.POISSON	37
RV.RAYLEIGH	35
RV.RTAIL	35
RV.T	36
RV.UNIFORM	36
RV.WEIBULL	36
RV.XPOWER	33

S

SD	26
SIG.CHISQ	33
SIG.F	33
SIN	24
SQRT	23
STRING	27
SUBSTR	27, 28
SUM	26
SYSMIS	24

T

TAN	24
TIME.DAYS	29
TIME.HMS	29
TRUNC	23

U

UNIFORM	36
UPCASE	28

V

VALUE	25
VARIANCE	26

X

XDATE.DATE	30
XDATE.HOUR	30
XDATE.JDAY	30
XDATE.MDAY	30
XDATE.MINUTE	30
XDATE.MONTH	31
XDATE.QUARTER	31
XDATE.SECOND	31

XDATE.TDAY	31
XDATE.TIME	31
XDATE.WEEK	31
XDATE.WKDAY	31
XDATE.YEAR	31

Y

YRMODA	31
--------------	----

17 Command Index

*

* 89

@

@ 90

A

ADD VALUE LABELS 58
 AGGREGATE 64
 APPLY DICTIONARY 53
 AUTORECODE 67

B

BEGIN DATA 39
 BREAK 77

C

CLEAR TRANSFORMATIONS 39
 COMMENT 89
 COMPUTE 67
 COUNT 67
 CROSSTABS 84

D

DATA LIST 39
 DATA LIST FIXED 39
 DATA LIST FREE 42
 DATA LIST LIST 43
 DESCRIPTIVES 80
 DISPLAY 58
 DISPLAY DOCUMENTS 89
 DISPLAY FILE LABEL 89
 DISPLAY VECTORS 58
 DO IF 77
 DO REPEAT 77
 DOCUMENT 89
 DROP DOCUMENTS 89

E

ECHO 90
 END CASE 43
 END DATA 39
 END FILE 43
 ERASE 90
 EXAMINE 83
 EXECUTE 90
 EXPORT 53

F

FILE HANDLE 43
 FILE LABEL 90
 FILTER 72
 FINISH 90
 FLIP 69
 FORMATS 59
 FREQUENCIES 81

G

GET 54

H

HOST 90

I

IF 69
 IMPORT 54
 INCLUDE 90
 INPUT PROGRAM 44

L

LEAVE 59
 LIST 46
 LOOP 78

M

MATCH FILES 55
 MATRIX DATA 47
 MISSING VALUES 60
 MODIFY VARS 60

N

N OF CASES 72
 NEW FILE 49
 NUMERIC 61

O

ONEWAY 88

P

PERMISSIONS 91
 PRINT 49
 PRINT EJECT 50
 PRINT FORMATS 61
 PRINT SPACE 50

PROCESS IF 73

Q

QUIT 91

R

RECODE 70

RENAME VARIABLES 61

REPEATING DATA 51

REREAD 50

S

SAMPLE 73

SAVE 56

SELECT IF 74

SET 91

SHOW 96

SORT CASES 71

SPLIT FILE 74

STRING 62

SUBTITLE 96

SYSFILE INFO 56

T

T-TEST 86

TEMPORARY 75

TITLE 97

V

VALUE LABELS 61

VARIABLE ALIGNMENT 62

VARIABLE LABELS 62

VARIABLE LEVEL 63

VARIABLE WIDTH 62

VECTOR 63

W

WEIGHT 75

WRITE 52

WRITE FORMATS 63

X

XSAVE 57

18 Concept Index

"		
'"		9
\$		
\$CASENUM		13
\$DATE		13
\$JDATE		13
\$LENGTH		13
\$SYSMIS		14
\$TIME		14
\$WIDTH		14
&		
'&'		22
,		
','		9
(
(23
'()'		21
)		
)		23
*		
'*'		21
'**'		22
+		
'+'		21
-		
'-'		21, 22
.		
'.'		12
.		20
/		
'/'		21
'usr/local/bin/'		112
'usr/local/info/'		112
'usr/local/share/pspp/'		112
<		
<		22
<=		22
<>		22
=		
'='		22
>		
'>'		22
>=		22
_		
'_'		12
`		
"is defined as"		20
' '		22
~		
'~'		22
~=		22
0		
0		9
A		
absolute value		23
active file		19
addition		21
analysis of variance		88
AND		22
ANOVA		88
arccosine		24
arcsine		24
arctangent		24
arguments, invalid		29
arguments, minimum valid		25
arguments, of date construction functions		29
arguments, of date extraction functions		30
arithmetic operators		21

attributes of variables 12

B

Backus-Naur Form 19
BNF 19
Boolean 21, 22

C

case conversion 28
case-sensitivity 8, 9
cases 39
changing file permissions 91
characters, reserved 9
coefficient of variation 25
command file 19
command line, options 3
command syntax, description of 19
commands, ordering 10
commands, structure 9
compiler, gcc 112
compiler, recommended 112
compiling 112
concatenation 26
conditionals 77
'config.h' 112
configuration 113
configure, GNU 112
constructing dates 29
constructing times 28
control flow 77
convention, TO 14
cosine 24
cross-case function 31

D

data 39
data file 19
data, embedding in syntax files 39
Data, embedding in syntax files 39
data, fixed-format, reading 39
data, reading from a file 39
date examination 30
date, Julian 31
dates 28
dates, concepts 28
dates, constructing 29
dates, day of the month 30
dates, day of the week 31
dates, day of the year 30
dates, day-month-year 29
dates, in days 30
dates, in hours 30
dates, in minutes 30
dates, in months 31
dates, in quarters 31

dates, in seconds 31
dates, in weekdays 31
dates, in weeks 31
dates, in years 31
dates, mathematical properties of 28
dates, month-year 30
dates, quarter-year 30
dates, time of day 31
dates, valid 28
dates, week-year 30
dates, year-day 30
day of the month 30
day of the week 31
day of the year 30
day-month-year 29
days 28, 29, 30, 31
description of command syntax 19
deviation, standard 26
dictionary 12
division 21
documentation, installing 112

E

embedding data in syntax files 39
Embedding data in syntax files 39
embedding fixed-format data 39
EQ 22
equality, testing 22
examination, of times 29
exponentiation 22
expression 20
expressions, mathematical 21
extraction, of dates 30
extraction, of time 29

F

false 22
FDL, GNU Free Documentation License 157
file definition commands 10
file mode 91
file, active 19
file, command 19
file, data 19
file, output 19
file, syntax file 19
files, PSPP 1
fixed-format data, reading 39
flow of control 77
Free Software Foundation 1
function, cross-case 31
functions 23
functions, miscellaneous 31
functions, missing-value 24
functions, statistical 25
functions, string 26
functions, time & date 28

G

gcc	112
GE	22
Ghostscript	1
GNU C compiler	112
GNU configure	112
graphics	1
greater than	22
greater than or equal to	22
grouping operators	21
GT	22

H

headers	96
hours	29, 30
hours-minutes-seconds	29

I

identifiers	8
identifiers, reserved	8
inequality, testing	22
input	39
input program commands	10
installation	112
installation, under UNIX	112
integer	20
integers	8
intersection, logical	22
introduction	1
inverse cosine	24
inverse sine	24
inverse tangent	24
inversion, logical	22
invocation	3

J

Julian date	31
-------------	----

K

keywords	20
----------	----

L

labels, value	13
labels, variable	13
language, command structure	9
language, lexical analysis	8
language, PSPP	1, 8
language, tokens	8
LE	22
length	96
less than	22
less than or equal to	22
lexical analysis	8

license	2
listing	96
logarithms	23
logical intersection	22
logical inversion	22
logical operators	22
logical union	22
loops	77
LT	22

M

makefile	112
'Makefile'	112
mathematical expressions	21
mathematics	23
mathematics, advanced	23
mathematics, applied to times & dates	28
mathematics, miscellaneous	23
maximum	25
mean	25
membership, of set	25
minimum	25
minimum valid number of arguments	25
minutes	29, 30
missing values	12, 13, 24
mode	91
modulus	23
modulus, by 10	23
month-year	30
months	31
more	96
multiplication	21

N

names, of functions	23
NE	22
negation	22
nonterminals	20
Normality, testing for	83
NOT	22
number	20
numbers	8
numbers, converting from strings	27
numbers, converting to strings	27

O

obligations, your	2
observations	39
operations, order of	37
operator precedence	37
operators	9, 20, 23
operators, arithmetic	21
operators, grouping	21
operators, logical	22
options, command-line	3

OR	22
order of commands	10
order of operations	37
output	39
output file	19
output, PSPP	1

P

padding strings	27
pager	96, 132
parentheses	21, 23
period	12
PostScript	1
precedence, operator	37
'pref.h'	112
print format	13
procedures	10
productions	20
PSPP language	1
PSPP, command structure	9
PSPP, configuring	113
PSPP, installing	112
PSPP, invoking	3
PSPP, language	8
punctuators	9, 20

Q

quarter-year	30
quarters	31

R

reading data from a file	39
reading fixed-format data	39
reals	8
reserved identifiers	8
restricted transformations	10
rights, your	2
rounding	23

S

searching strings	26
seconds	29, 31
self-tests, running	112
set membership	25
sine	24
square roots	23
standard deviation	26
start symbol	20
statistics	25
string	20
string functions	26
strings	9
strings, case of	26, 28
strings, concatenation of	26

strings, converting from numbers	27
strings, converting to numbers	27
strings, finding length of	26
strings, padding	26, 27
strings, searching backwards	27
strings, taking substrings of	27
strings, trimming	26, 27
substrings	27
subtraction	21
sum	26
symbol, start	20
syntax file	19
system variables	13
system-missing	22

T

tangent	24
terminals	20
terminals and nonterminals, differences	20
testing for equality	22
testing for inequality	22
time	31
time examination	29
time, concepts	28
time, in days	28, 29, 30
time, in hours	29, 30
time, in hours-minutes-seconds	29
time, in minutes	29, 30
time, in seconds	29, 31
time, instants of	28
time, intervals	28
time, lengths of	29
time, mathematical properties of	28
times	28
times, constructing	28
times, in days	31
TO convention	14
tokens	8
transformations	10, 64
trigonometry	24
true	22
truncation	23
type of variables	12

U

union, logical	22
UNIX, installing PSPP under	112
utility commands	10

V

value labels	13
values, Boolean	21
values, missing	12, 13, 24
values, system-missing	22
var-list	20

`var-name` 20
variable labels 13
variable names, ending with period 12
variables 12
variables, attributes of 12
variables, system 13
variables, type 12
variables, width 12
variance 26
variation, coefficient of 25

W

week 31

week-year 30
weekday 31
white space 9
white space, trimming 26, 27
width 96
width of variables 12
write format 13

Y

year-day 30
years 31
your rights and obligations 2

Appendix A Installing PSPP

PSPP conforms to the GNU Coding Standards. PSPP is written in, and requires for proper operation, ANSI/ISO C. You might want to additionally note the following points:

- The compiler and linker must allow for significance of several characters in external identifiers. The exact number is unknown but at least 31 is recommended.
- The `int` type must be 32 bits or wider.
- The recommended compiler is gcc 2.7.2.1 or later, but any ANSI compiler will do if it fits the above criteria.

Many UNIX variants should work out-of-the-box, as PSPP uses GNU autoconf to detect differences between environments. Please report any problems with compilation of PSPP under UNIX and UNIX-like operating systems—portability is a major concern of the author.

The pages below give specific instructions for installing PSPP on each type of system mentioned above.

A.1 UNIX installation

To install PSPP under a UNIX-like operating system, follow the steps below in order. Some of the text below was taken directly from various Free Software Foundation sources.

1. `cd` to the directory containing the PSPP source.
2. Type `./configure` to configure for your particular operating system and compiler. Running `configure` takes a while. While running, it displays some messages telling which features it is checking for.

You can optionally supply some options to `configure` to give it hints about how to do its job. Type `./configure --help` to see a list of options. One of the most useful options is `--with-checker`, which enables the use of the Checker memory debugger under supported operating systems. Checker must already be installed to use this option. Do not use `--with-checker` if you are not debugging PSPP itself.

3. (optional) Edit `Makefile`, `config.h`, and `pref.h`. These files are produced by `configure`. Note that most PSPP settings can be changed at runtime.

`'pref.h'` is only generated by `configure` if it does not already exist. (It's copied from `'prefh.orig'`.)

4. Type `'make'` to compile the package. If there are any errors during compilation, try to fix them. If modifications are necessary to compile correctly under your configuration, contact the author. See [Chapter 15 \[Submitting Bug Reports\]](#), page 101, for details.
5. Type `'make check'` to run self-tests on the compiled PSPP package.
6. Become the superuser and type `'make install'` to install the PSPP binaries, by default in `'/usr/local/bin/'`. The directory `'/usr/local/share/pspp/'` is created and populated with files needed by PSPP at runtime. This step will also cause the PSPP documentation to be installed in `'/usr/local/info/'`, but only if that directory already exists.
7. (optional) Type `'make clean'` to delete the PSPP binaries from the source tree.

Appendix B Configuring PSPP

PSPP has dozens of configuration possibilities and hundreds of settings. This is both a bane and a blessing. On one hand, it's possible to easily accommodate diverse ranges of setups. But, on the other, the multitude of possibilities can overwhelm the casual user. Fortunately, the configuration mechanisms are profusely described in the sections below. . .

B.1 Locating configuration files

PSPP uses the same method to find most of its configuration files:

1. The *base name* of the file being sought is determined.
2. The path to search is determined.
3. Each directory in the search path, from left to right, is searched for a file with the name of the base name. The first occurrence is read as the configuration file.

The first two steps are elaborated below for the sake of our pedantic friends.

1. A *base name* is a file name lacking an absolute directory reference. Some examples of base names are: 'ps-encodings', 'devices', 'devps/DESC' (under UNIX), 'devps\DESC' (under M\$ environments).

Determining the base name is a two-step process:

- a. If the appropriate environment variable is defined, the value of that variable is used (see [Section B.4 \[Environment variables\]](#), page 115). For instance, when searching for the output driver initialization file, the variable examined is `STAT_OUTPUT_INIT_FILE`.
- b. Otherwise, the compiled-in default is used. For example, when searching for the output driver initialization file, the default base name is 'devices'.

Please note: If a user-specified base name does contain an absolute directory reference, as in a file name like '/home/pfaff/fonts/TR', no path is searched—the file name is used exactly as given—and the algorithm terminates.

2. The path is the first of the following that is defined:
 - A variable definition for the path given in the user environment. This is a PSPP-specific environment variable name; for instance, `STAT_OUTPUT_INIT_PATH`.
 - In some cases, another, less-specific environment variable is checked. For instance, when searching for font files, the PostScript driver first checks for a variable with name `STAT_GROFF_FONT_PATH`, then for one with name `GROFF_FONT_PATH`. (However, font searching has its own list of esoteric search rules.)
 - The configuration file path, which is itself determined by the following rules:
 - a. If the command line contains an option of the form '`-B path`' or '`--config-dir=path`', then the value given on the rightmost occurrence of such an option is used.
 - b. Otherwise, if the environment variable `STAT_CONFIG_PATH` is defined, the value of that variable is used.
 - c. Otherwise, the compiled-in fallback default is used. On UNIX machines, the default fallback path is

1. `'~/pspp'`
2. `'/usr/local/lib/pspp'`
3. `'/usr/lib/pspp'`

On DOS machines, the default fallback path is:

1. All the paths from the DOS search path in the `'PATH'` environment variable, in left-to-right order.
2. `'C:\PSPP'`, as a last resort.

Note that the installer of PSPP can easily change this default fallback path; thus the above should not be taken as gospel.

As a final note: Under DOS, directories given in paths are delimited by semicolons (`;`); under UNIX, directories are delimited by colons (`:`). This corresponds with the standard path delimiter under these OSes.

B.2 Configuration techniques

There are many ways that PSPP can be configured. These are described in the list below. Values given by earlier items take precedence over those given by later items.

1. Syntax commands that modify settings, such as `SET`. See [Section 13.15 \[SET\]](#), page 91.
2. Command-line options. See [Chapter 3 \[Invocation\]](#), page 3.
3. PSPP-specific environment variable contents. See [Section B.4 \[Environment variables\]](#), page 115.
4. General environment variable contents. See [Section B.4 \[Environment variables\]](#), page 115.
5. Configuration file contents. See [Section B.3 \[Configuration files\]](#), page 114.
6. Fallback defaults.

Some of the above may not apply to a particular setting. For instance, the current pager (such as `'more'`, `'most'`, or `'less'`) cannot be determined by configuration file contents because there is no appropriate configuration file.

B.3 Configuration files

Most configuration files have a common form:

- Each line forms a separate command or directive. This means that lines cannot be broken up, unless they are spliced together with a trailing backslash, as described below.
- Before anything else is done, trailing white space is removed.
- When a line ends in a backslash (`'\'`), the backslash is removed, and the next line is read and appended to the current line.
 - White space preceding the backslash is retained.
 - This rule continues to be applied until the line read does not end in a backslash.
 - It is an error if the last line in the file ends in a backslash.
- Comments are introduced by an octothorpe (`'#'`), and continue until the end of the line.

- An octothorpe inside balanced pairs of double quotation marks ("") or single quotation marks ('') does not introduce a comment.
- The backslash character can be used inside balanced quotes of either type to escape the following character as a literal character.
(This is distinct from the use of a backslash as a line-splicing character.)
- Line splicing takes place before comment removal.
- Blank lines, and lines that contain only white space, are ignored.

B.4 Environment variables

You may think the concept of environment variables is a fairly simple one. However, the author of PSPP has found a way to complicate even something so simple. Environment variables are further described in the sections below:

B.4.1 Values of environment variables

Values for environment variables are obtained by the following means, which are arranged in order of decreasing precedence:

1. Command-line options. See [Chapter 3 \[Invocation\]](#), page 3.
2. The ‘environment’ configuration file—more on this below.
3. Actual environment variables (defined in the shell or other parent process).

The ‘environment’ configuration file is located through application of the usual algorithm for configuration files (see [Section B.1 \[File locations\]](#), page 113), except that its contents do not affect the search path used to find ‘environment’ itself. Use of ‘environment’ is discouraged on systems that allow an arbitrarily large environment; it is supported for use on systems like MS-DOS that limit environment size.

‘environment’ is composed of lines having the form ‘*key=value*’, where *key* and the equals sign (=) are required, and *value* is optional. If *value* is given, variable *key* is given that value; if *value* is absent, variable *key* is undefined (deleted). Variables may not be defined with a null value.

Environment substitutions are performed on each line in the file (see [Section B.4.2 \[Environment substitutions\]](#), page 115).

See [Section B.3 \[Configuration files\]](#), page 114, for more details on formatting of the environment configuration file.

Please note: Support for ‘environment’ is not yet implemented.

B.4.2 Environment substitutions

Much of the power of environment variables lies in the way that they may be substituted into configuration files. Variable substitutions are described below.

The line is scanned from left to right. In this scan, all characters other than dollar signs (\$) are retained unmolested. Dollar signs, however, introduce an environment variable reference. References take three forms:

\$var Replaced by the value of environment variable *var*, determined as specified in [Section B.4.1 \[Variable values\]](#), page 115. *var* must be one of the following:

- One or more letters.
 - Exactly one nonalphabetic character. This may not be a left brace ('{').
- `${var}` Same as above, but `var` may contain any character (except '}').
- `$$` Replaced by a single dollar sign.
- Undefined variables expand to a empty value.

B.4.3 Predefined environment variables

There are two environment variables predefined for use in environment substitutions:

- `'VER'` Defined as the version number of PSPP, as a string, in a format something like `'0.9.4'`.
- `'ARCH'` Defined as the host architecture of PSPP, as a string, in standard `cpu-manufacturer-OS` format. For instance, Debian GNU/Linux 1.1 on an Intel machine defines this as `'i586-unknown-linux'`. This is somewhat dependent on the system used to compile PSPP.

Nothing prevents these values from being overridden, although it's a good idea not to do so.

B.5 Output devices

Configuring output devices is the most complicated aspect of configuring PSPP. The output device configuration file is named `'devices'`. It is searched for using the usual algorithm for finding configuration files (see [Section B.1 \[File locations\]](#), page 113). Each line in the file is read in the usual manner for configuration files (see [Section B.3 \[Configuration files\]](#), page 114).

Lines in `'devices'` are divided into three categories, described briefly in the table below:

driver category definitions

Define a driver in terms of other drivers.

macro definitions

Define environment variables local to the the output driver configuration file.

device definitions

Describe the configuration of an output device.

The following sections further elaborate the contents of the `'devices'` file.

B.5.1 Driver categories

Drivers can be divided into categories. Drivers are specified by their names, or by the names of the categories that they are contained in. Only certain drivers are enabled each time PSPP is run; by default, these are the drivers in the category `'default'`. To enable a different set of drivers, use the `'-o device'` command-line option (see [Chapter 3 \[Invocation\]](#), page 3).

Categories are specified with a line of the form `'category=driver1 driver2 driver3 ... drivern'`. This line specifies that the category `category` is composed of drivers named `driver1`, `driver2`, and so on. There may be any number of drivers in the category, from zero on up.

Categories may also be specified on the command line (see [Chapter 3 \[Invocation\]](#), [page 3](#)).

This is all you need to know about categories. If you're still curious, read on.

First of all, the term 'categories' is a bit of a misnomer. In fact, the internal representation is nothing like the hierarchy that the term seems to imply: a linear list is used to keep track of the enabled drivers.

When PSPP first begins reading 'devices', this list contains the name of any drivers or categories specified on the command line, or the single item 'default' if none were specified.

Each time a category definition is specified, the list is searched for an item with the value of *category*. If a matching item is found, it is deleted. If there was a match, the list of drivers (*driver1* through *drivern*) is then appended to the list.

Each time a driver definition line is encountered, the list is searched. If the list contains an item with that driver's name, the driver is enabled and the item is deleted from the list. Otherwise, the driver is not enabled.

It is an error if the list is not empty when the end of 'devices' is reached.

B.5.2 Macro definitions

Macro definitions take the form '**define *macroname* *definition***'. In such a macro definition, the environment variable *macroname* is defined to expand to the value *definition*. Before the definition is made, however, any macros used in *definition* are expanded.

Please note the following nuances of macro usage:

- For the purposes of this section, *macro* and *environment variable* are synonyms.
- Macros may not take arguments.
- Macros may not recurse.
- Macros are just environment variable definitions like other environment variable definitions, with the exception that they are limited in scope to the 'devices' configuration file.
- Macros override other all environment variables of the same name (within the scope of 'devices').
- Earlier macro definitions for a particular *key* override later ones. In particular, macro definitions on the command line override those in the device definition file. See [Section 3.1 \[Non-option Arguments\]](#), [page 3](#).
- There are two predefined macros, whose values are determined at runtime:

'viewwidth'

Defined as the width of the console screen, in columns of text.

'viewlength'

Defined as the length of the console screen, in lines of text.

B.5.3 Driver definitions

Driver definitions are the ultimate purpose of the 'devices' configuration file. These are where the real action is. Driver definitions tell PSPP where it should send its output.

Each driver definition line is divided into four fields. These fields are delimited by colons (':'). Each line is subjected to environment variable interpolation before it is processed

further (see [Section B.4.2 \[Environment substitutions\]](#), page 115). From left to right, the four fields are, in brief:

driver name

A unique identifier, used to determine whether to enable the driver.

class name

One of the predefined driver classes supported by PSPP. The currently supported driver classes include ‘postscript’ and ‘ascii’.

device type(s)

Zero or more of the following keywords, delimited by spaces:

screen

Indicates that the device is a screen display. This may reduce the amount of buffering done by the driver, to make interactive use more convenient.

printer

Indicates that the device is a printer.

listing

Indicates that the device is a listing file.

These options are just hints to PSPP and do not cause the output to be directed to the screen, or to the printer, or to a listing file—those must be set elsewhere in the options. They are used primarily to decide which devices should be enabled at any given time. See [Section 13.15 \[SET\]](#), page 91, for more information.

options

An optional set of options to pass to the driver itself. The exact format for the options varies among drivers.

The driver is enabled if:

1. Its driver name is specified on the command line, or
2. It’s in a category specified on the command line, or
3. If no categories or driver names are specified on the command line, it is in category **default**.

For more information on driver names, see [Section B.5.1 \[Driver categories\]](#), page 116.

The class name must be one of those supported by PSPP. The classes supported depend on the options with which PSPP was compiled. See later sections in this chapter for descriptions of the available driver classes.

Options are dependent on the driver. See the driver descriptions for details.

B.5.4 Dimensions

Quite often in configuration it is necessary to specify a length or a size. PSPP uses a common syntax for all such, calling them collectively by the name *dimensions*.

- You can specify dimensions in decimal form (‘12.5’) or as fractions, either as mixed numbers (‘12-1/2’) or raw fractions (‘25/2’).

- A number of different units are available. These are suffixed to the numeric part of the dimension. There must be no spaces between the number and the unit. The available units are identical to those offered by the popular typesetting system \TeX :

<code>in</code>	inch (1 <code>in</code> = 2.54 <code>cm</code>)
<code>"</code>	inch (1 <code>in</code> = 2.54 <code>cm</code>)
<code>pt</code>	printer's point (1 <code>in</code> = 72.27 <code>pt</code>)
<code>pc</code>	pica (12 <code>pt</code> = 1 <code>pc</code>)
<code>bp</code>	PostScript point (1 <code>in</code> = 72 <code>bp</code>)
<code>cm</code>	centimeter
<code>mm</code>	millimeter (10 <code>mm</code> = 1 <code>cm</code>)
<code>dd</code>	didot point (1157 <code>dd</code> = 1238 <code>pt</code>)
<code>cc</code>	cicero (1 <code>cc</code> = 12 <code>dd</code>)
<code>sp</code>	scaled point (65536 <code>sp</code> = 1 <code>pt</code>)

- If no explicit unit is given, PSPP attempts to guess the best unit:
 - Numbers less than 50 are assumed to be in inches.
 - Numbers 50 or greater are assumed to be in millimeters.

B.5.5 Paper sizes

Output drivers usually deal with some sort of hardcopy media. This media is called *paper* by the drivers, though in reality it could be a transparency or film or thinly veiled sarcasm. To make it easier for you to deal with paper, PSPP allows you to have (of course!) a configuration file that gives symbolic names, like “letter” or “legal” or “a4”, to paper sizes, rather than forcing you to use cryptic numbers like “8-1/2 x 11” or “210 by 297”. Surprisingly enough, this configuration file is named ‘`papersize`’. See [Section B.3 \[Configuration files\]](#), [page 114](#).

When PSPP tries to connect a symbolic paper name to a paper size, it reads and parses each non-comment line in the file, in order. The first field on each line must be a symbolic paper name in double quotes. Paper names may not contain double quotes. Paper names are not case-sensitive: ‘`legal`’ and ‘`Legal`’ are equivalent.

If a match is found for the paper name, the rest of the line is parsed. If it is found to be a pair of dimensions (see [Section B.5.4 \[Dimensions\]](#), [page 118](#)) separated by either ‘`x`’ or ‘`by`’, then those are taken to be the paper size, in order of width followed by length. There *must* be at least one space on each side of ‘`x`’ or ‘`by`’.

Otherwise the line must be of the form “`paper-1`”=`paper-2`”. In this case the target of the search becomes paper name *paper-2* and the search through the file continues.

B.5.6 How lines are divided into types

The lines in ‘`devices`’ are distinguished in the following manner:

1. Leading white space is removed.
2. If the resulting line begins with the exact string `define`, followed by one or more white space characters, the line is processed as a macro definition.

3. Otherwise, the line is scanned for the first instance of a colon (':') or an equals sign ('=').
4. If a colon is encountered first, the line is processed as a driver definition.
5. Otherwise, if an equals sign is encountered, the line is processed as a macro definition.
6. Otherwise, the line is ill-formed.

B.5.7 How lines are divided into tokens

Each driver definition line is run through a simple tokenizer. This tokenizer recognizes two basic types of tokens.

The first type is an equals sign ('='). Equals signs are both delimiters between tokens and tokens in themselves.

The second type is an identifier or string token. Identifiers and strings are equivalent after tokenization, though they are written differently. An identifier is any string of characters other than white space or equals sign.

A string is introduced by a single- or double-quote character ('' or "") and, in general, continues until the next occurrence of that same character. The following standard C escapes can also be embedded within strings:

<code>\'</code>	A single-quote ('').
<code>\"</code>	A double-quote (").
<code>\?</code>	A question mark (?). Included for hysterical raisins.
<code>\\</code>	A backslash (\).
<code>\a</code>	Audio bell (ASCII 7).
<code>\b</code>	Backspace (ASCII 8).
<code>\f</code>	Formfeed (ASCII 12).
<code>\n</code>	New-line (ASCII 10).
<code>\r</code>	Carriage return (ASCII 13).
<code>\t</code>	Tab (ASCII 9).
<code>\v</code>	Vertical tab (ASCII 11).
<code>\ooo</code>	Each 'o' must be an octal digit. The character is the one having the octal value specified. Any number of octal digits is read and interpreted; only the lower 8 bits are used.
<code>\xhh</code>	Each 'h' must be a hex digit. The character is the one having the hexadecimal value specified. Any number of hex digits is read and interpreted; only the lower 8 bits are used.

Tokens, outside of quoted strings, are delimited by white space or equals signs.

B.6 The PostScript driver class

The `postscript` driver class is used to produce output that is acceptable to PostScript printers and to PC-based PostScript interpreters such as Ghostscript. Continuing a long tradition, PSPP's PostScript driver is configurable to the point of absurdity.

There are actually two PostScript drivers. The first one, `'postscript'`, produces ordinary DSC-compliant PostScript output. The second one `'epsf'`, produces an Encapsulated PostScript file. The two drivers are otherwise identical in configuration and in operation.

The PostScript driver is described in further detail below.

B.6.1 PostScript output options

These options deal with the form of the output and the output file itself:

`output-file=filename`

File to which output should be sent. This can be an ordinary filename (i.e., `"pspp.ps"`), a pipe filename (i.e., `"|lpr"`), or stdout (`"-"`). Default: `"pspp.ps"`.

`color=boolean`

Most of the time black-and-white PostScript devices are smart enough to map colors to shades themselves. However, you can cause the PSPP output driver to do an ugly simulation of this in its own driver by turning `color` off. Default: `on`.

This is a boolean setting, as are many settings in the PostScript driver. Valid positive boolean values are `'on'`, `'true'`, `'yes'`, and nonzero integers. Negative boolean values are `'off'`, `'false'`, `'no'`, and zero.

`data=data-type`

One of `clean7bit`, `clean8bit`, or `binary`. This controls what characters will be written to the output file. PostScript produced with `clean7bit` can be transmitted over 7-bit transmission channels that use ASCII control characters for line control. `clean8bit` is similar but allows characters above 127 to be written to the output file. `binary` allows any character in the output file. Default: `clean7bit`.

`line-ends=line-end-type`

One of `cr`, `lf`, or `crlf`. This controls what is used for new-line in the output file. Default: `cr`.

`optimize-line-size=level`

Either 0 or 1. If `level` is 1, then short line segments will be collected and merged into longer ones. This reduces output file size but requires more time and memory. A `level` of 0 has the advantage of being better for interactive environments. 1 is the default unless the `screen` flag is set; in that case, the default is 0.

`optimize-text-size=level`

One of 0, 1, or 2, each higher level representing correspondingly more aggressive space savings for text in the output file and requiring correspondingly more time and memory. Unfortunately the levels presently are all the same. 1 is the default unless the `screen` flag is set; in that case, the default is 0.

B.6.2 PostScript page options

These options affect page setup:

headers=boolean

Controls whether the standard headers showing the time and date and title and subtitle are printed at the top of each page. Default: **on**.

paper-size=paper-size

Paper size, either as a symbolic name (i.e., **letter** or **a4**) or specific measurements (i.e., 8-1/2x11 or "210 x 297". See [Section B.5.5 \[Paper sizes\]](#), page 119. Default: **letter**.

orientation=orientation

Either **portrait** or **landscape**. Default: **portrait**.

left-margin=dimension

right-margin=dimension

top-margin=dimension

bottom-margin=dimension

Sets the margins around the page. The headers, if enabled, are not included in the margins; they are in addition to the margins. For a description of dimensions, see [Section B.5.4 \[Dimensions\]](#), page 118. Default: **0.5in**.

B.6.3 PostScript file options

Oh, my. You don't really want to know about the way that the PostScript driver deals with files, do you? Well I suppose you're entitled, but I warn you right now: it's not pretty. Here goes...

First let's look at the options that are available:

font-dir=font-directory

Sets the font directory. Default: **devps**.

prologue-file=prologue-file-name

Sets the name of the PostScript prologue file. You can write your own prologue, though I have no idea why you'd want to: see [Section B.6.6 \[Prologue\]](#), page 124. Default: **ps-prologue**.

device-file=device-file-name

Sets the name of the Groff-format device description file. The PostScript driver reads this to know about the scaling of fonts and so on. The format of such files is described in the `groff-font` man page, included with Groff. Default: **DESC**.

encoding-file=encoding-file-name

Sets the name of the encoding file. This file contains a list of all font encodings that will be needed so that the driver can put all of them at the top of the prologue. See [Section B.6.7 \[Encodings\]](#), page 126. Default: **ps-encodings**.

If the specified encoding file cannot be found, this error will be silently ignored, since most people do not need any encodings besides the ones that can be found using **auto-encodings**, described below.

auto-encode=*boolean*

When enabled, the font encodings needed by the default proportional- and fixed-pitch fonts will automatically be dumped to the PostScript output. Otherwise, it is assumed that the user has an encoding file and knows how to use it (see [Section B.6.7 \[Encodings\], page 126](#)). There is probably no good reason to turn off this convenient feature. Default: **on**.

Next I suppose it's time to describe the search algorithm. When the PostScript driver needs a file, whether that file be a font, a PostScript prologue, or what you will, it searches in this manner:

1. Constructs a path by taking the first of the following that is defined:
 - a. Environment variable `STAT_GROFF_FONT_PATH`. See [Section B.4 \[Environment variables\], page 115](#).
 - b. Environment variable `GROFF_FONT_PATH`.
 - c. The compiled-in fallback default.
2. Constructs a base name from concatenating, in order, the font directory, a path separator ('/' or '\'), and the file to be found. A typical base name would be something like `devps/ps-encodings`.
3. Searches for the base name in the path constructed above. If the file is found, the algorithm terminates.
4. Searches for the base name in the standard configuration path. See [Section B.1 \[File locations\], page 113](#), for more details. If the file is found, the algorithm terminates.
5. At this point we remove the font directory and path separator from the base name. Now the base name is simply the file to be found, i.e., `ps-encodings`.
6. Searches for the base name in the path constructed in the first step. If the file is found, the algorithm terminates.
7. Searches for the base name in the standard configuration path. If the file is found, the algorithm terminates.
8. The algorithm terminates unsuccessfully.

So, as you see, there are several ways to configure the PostScript drivers. Careful selection of techniques can make the configuration very flexible indeed.

B.6.4 PostScript font options

The list of available font options is short and sweet:

prop-font=*font-name*

Sets the default proportional font. The name should be that of a PostScript font. Default: **"Helvetica"**.

fixed-font=*font-name*

Sets the default fixed-pitch font. The name should be that of a PostScript font. Default: **"Courier"**.

font-size=*font-size*

Sets the size of the default fonts, in thousandths of a point. Default: **10000**.

B.6.5 PostScript line options

Most tables contain lines, or rules, between cells. Some features of the way that lines are drawn in PostScript tables are user-definable:

line-style=style

Sets the style used for lines used to divide tables into sections. *style* must be either **thick**, in which case thick lines are used, or *double*, in which case double lines are used. Default: **thick**.

line-gutter=dimension

Sets the line gutter, which is the amount of white space on either side of lines that border text or graphics objects. See [Section B.5.4 \[Dimensions\]](#), page 118. Default: 0.5pt.

line-spacing=dimension

Sets the line spacing, which is the amount of white space that separates lines that are side by side, as in a double line. Default: 0.5pt.

line-width=dimension

Sets the width of a typical line used in tables. Default: 0.5pt.

line-width-thick=dimension

Sets the width of a thick line used in tables. Not used if **line-style** is set to **thick**. Default: 1.5pt.

B.6.6 The PostScript prologue

Most PostScript files that are generated mechanically by programs consist of two parts: a prologue and a body. The prologue is generally a collection of boilerplate. Only the body differs greatly between two outputs from the same program.

This is also the strategy used in the PSPP PostScript driver. In general, the prologue supplied with PSPP will be more than sufficient. In this case, you will not need to read the rest of this section. However, hackers might want to know more. Read on, if you fall into this category.

The prologue is dumped into the output stream essentially unmodified. However, two actions are performed on its lines. First, certain lines may be omitted as specified in the prologue file itself. Second, variables are substituted.

The following lines are omitted:

1. All lines that contain three bangs in a row (!!!).
2. Lines that contain **!eps**, if the PostScript driver is producing ordinary PostScript output. Otherwise an EPS file is being produced, and the line is included in the output, although everything following **!eps** is deleted.
3. Lines that contain **!ps**, if the PostScript driver is producing EPS output. Otherwise, ordinary PostScript is being produced, and the line is included in the output, although everything following **!ps** is deleted.

The following are the variables that are substituted. Only the variables listed are substituted; environment variables are not. See [Section B.4.2 \[Environment substitutions\]](#), page 115.

bounding-box

The page bounding box, in points, as four space-separated numbers. For U.S. letter size paper, this is '0 0 612 792'.

creator

PSPP version as a string: 'GNU PSPP 0.1b', for example.

date

Date the file was created. Example: 'Tue May 21 13:46:22 1991'.

data

Value of the **data** PostScript driver option, as one of the strings 'Clean7Bit', 'Clean8Bit', or 'Binary'.

orientation

Page orientation, as one of the strings **Portrait** or **Landscape**.

user

Under multiuser OSes, the user's login name, taken either from the environment variable **LOGNAME** or, if that fails, the result of the C library function **getlogin()**. Defaults to 'nobody'.

host

System hostname as reported by **gethostname()**. Defaults to 'nowhere'.

prop-font

Name of the default proportional font, prefixed by the word 'font' and a space. Example: 'font Times-Roman'.

fixed-font

Name of the default fixed-pitch font, prefixed by the word 'font' and a space.

scale-factor

The page scaling factor as a floating-point number. Example: 1.0. Note that this is also passed as an argument to the **BP** macro.

paper-length**paper-width**

The paper length and paper width, respectively, in thousandths of a point. Note that these are also passed as arguments to the **BP** macro.

left-margin**top-margin**

The left margin and top margin, respectively, in thousandths of a point. Note that these are also passed as arguments to the **BP** macro.

title

Document title as a string. This is not the title specified in the PSPP syntax file. A typical title is the word 'PSPP' followed by the syntax file name in parentheses. Example: 'PSPP (<stdin>)'.

source-file

PSPP syntax file name. Example: 'mary96/first.stat'.

Any other questions about the PostScript prologue can best be answered by examining the default prologue or the PSPP source.

B.6.7 PostScript encodings

PostScript fonts often contain many more than 256 characters, in order to accommodate foreign language characters and special symbols. PostScript uses *encodings* to map these onto single-byte symbol sets. Each font can have many different encodings applied to it.

PSPP's PostScript driver needs to know which encoding to apply to each font. It can determine this from the information encapsulated in the Groff font description that it reads. However, there is an additional problem—for efficiency, the PostScript driver needs to have a complete list of all encodings that will be used in the entire session *when it opens the output file*. For this reason, it can't use the information built into the fonts because it doesn't know which fonts will be used.

As a stopgap solution, there are two mechanisms for specifying which encodings will be used. The first mechanism is automatic and it is the only one that most PSPP users will ever need. The second mechanism is manual, but it is more flexible. Either mechanism or both may be used at one time.

The first mechanism is activated by the 'auto-encode' driver option (see [Section B.6.3 \[PS file options\]](#), [page 122](#)). When enabled, 'auto-encode' causes the PostScript driver to include the encodings used by the default proportional and fixed-pitch fonts (see [Section B.6.4 \[PS font options\]](#), [page 123](#)). Many PSPP output files will only need these encodings.

The second mechanism is the file specified by the 'encoding-file' option (see [Section B.6.3 \[PS file options\]](#), [page 122](#)). If it exists, this file must consist of lines in PSPP configuration-file format (see [Section B.3 \[Configuration files\]](#), [page 114](#)). Each line that is not a comment should name a PostScript encoding to include in the output.

It is not an error if an encoding is included more than once, by either mechanism. It will appear only once in the output. It is also not an error if an encoding is included in the output but never used. It *is* an error if an encoding is used but not included by one of these mechanisms. In this case, the built-in PostScript encoding 'ISOLatin1Encoding' is substituted.

B.7 The ASCII driver class

The ASCII driver class produces output that can be displayed on a terminal or output to printers. All of its options are highly configurable. The ASCII driver has class name 'ascii'.

The ASCII driver is described in further detail below.

B.7.1 ASCII output options

output-file=filename

File to which output should be sent. This can be an ordinary filename (e.g., "pspp.txt"), a pipe filename (e.g., "|lpr"), or stdout ("-"). Default: "pspp.list".

char-set=char-set-type

One of 'ascii' or 'latin1'. This has no effect on output at the present time.
Default: `ascii`.

form-feed-string=form-feed-value

The string written to the output to cause a formfeed. See also `paginate`, described below, for a related setting. Default: `"\f"`.

newline-string=new-line-value

The string written to the output to cause a new-line (carriage return plus linefeed). The default, which can be specified explicitly with `newline-string=default`, is to use the system-dependent new-line sequence by opening the output file in text mode. This is usually the right choice.

However, `newline-string` can be set to any string. When this is done, the output file is opened in binary mode.

paginate=boolean

If set, a formfeed (as set in `form-feed-string`, described above) will be written to the device after every page. Default: `on`.

tab-width=tab-width-value

The distance between tab stops for this device. If set to 0, tabs will not be used in the output. Default: `8`.

init=initialization-string.

String written to the device before anything else, at the beginning of the output.
Default: `""` (the empty string).

done=finalization-string.

String written to the device after everything else, at the end of the output.
Default: `""` (the empty string).

B.7.2 ASCII page options

These options affect page setup:

headers=boolean

If enabled, two lines of header information giving title and subtitle, page number, date and time, and PSPP version are printed at the top of every page. These two lines are in addition to any top margin requested. Default: `on`.

length=line-count

Physical length of a page, in lines. Headers and margins are subtracted from this value. Default: `66`.

width=character-count

Physical width of a page, in characters. Margins are subtracted from this value. Default: `130`.

lpi=lines-per-inch

Number of lines per vertical inch. Not currently used. Default: `6`.

cpi=characters-per-inch

Number of characters per horizontal inch. Not currently used. Default: `10`.

left-margin=left-margin-width

Width of the left margin, in characters. PSPP subtracts this value from the page width. Default: 0.

right-margin=right-margin-width

Width of the right margin, in characters. PSPP subtracts this value from the page width. Default: 0.

top-margin=top-margin-lines

Length of the top margin, in lines. PSPP subtracts this value from the page length. Default: 2.

bottom-margin=bottom-margin-lines

Length of the bottom margin, in lines. PSPP subtracts this value from the page length. Default: 2.

B.7.3 ASCII font options

These are the ASCII font options:

box[line-type]=box-chars

The characters used for lines in tables produced by the ASCII driver can be changed using this option. *line-type* is used to indicate which type of line to change; *box-chars* is the character or string of characters to use for this type of line.

line-type must be a 4-digit number in base 4. The digits are in the order ‘right’, ‘bottom’, ‘left’, ‘top’. The four possibilities for each digit are:

- | | |
|---|---|
| 0 | No line. |
| 1 | Single line. |
| 2 | Double line. |
| 3 | Special device-defined line, if one is available; otherwise, a double line. |

Examples:

box[0101]="|"

Sets ‘|’ as the character to use for a single-width line with bottom and top components.

box[2222]="#"

Sets ‘#’ as the character to use for the intersection of four double-width lines, one each from the top, bottom, left and right.

box[1100]="\xda"

Sets “\xda”, which under MS-DOS is a box character suitable for the top-left corner of a box, as the character for the intersection of two single-width lines, one each from the right and bottom.

Defaults:

- **box[0000]=" "**

- `box[1000]="-"`
`box[0010]="-"`
`box[1010]="-"`
- `box[0100]="|"`
`box[0001]="|"`
`box[0101]="|"`
- `box[2000]="="`
`box[0020]="="`
`box[2020]="="`
- `box[0200]="#"`
`box[0002]="#"`
`box[0202]="#"`
- `box[3000]="="`
`box[0030]="="`
`box[3030]="="`
- `box[0300]="#"`
`box[0003]="#"`
`box[0303]="#"`
- For all others, '+' is used unless there are double lines or special lines, in which case '#' is used.

`italic-on=italic-on-string`

Character sequence written to turn on italics or underline printing. If this is set to `overstrike`, then the driver will simulate underlining by overstriking with underscore characters ('_') in the manner described by `overstrike-style` and `carriage-return-style`. Default: `overstrike`.

`italic-off=italic-off-string`

Character sequence to turn off italics or underline printing. Default: "" (the empty string).

`bold-on=bold-on-string`

Character sequence written to turn on bold or emphasized printing. If set to `overstrike`, then the driver will simulated bold printing by overstriking characters in the manner described by `overstrike-style` and `carriage-return-style`. Default: `overstrike`.

`bold-off=bold-off-string`

Character sequence to turn off bold or emphasized printing. Default: "" (the empty string).

`bold-italic-on=bold-italic-on-string`

Character sequence written to turn on bold-italic printing. If set to `overstrike`, then the driver will simulate bold-italics by overstriking twice, once with the character, a second time with an underscore ('_') character, in the manner described by `overstrike-style` and `carriage-return-style`. Default: `overstrike`.

bold-italic-off=bold-italic-off-string

Character sequence to turn off bold-italic printing. Default: "" (the empty string).

overstrike-style=overstrike-option

Either **single** or **line**:

- If **single** is selected, then, to overstrike a line of text, the output driver will output a character, backspace, overstrike, output a character, backspace, overstrike, and so on along a line.
- If **line** is selected then the output driver will output an entire line, then backspace or emit a carriage return (as indicated by **carriage-return-style**), then overstrike the entire line at once.

single is recommended for use with ttys and programs that understand overstriking in text files, such as the pager **less**. **single** will also work with printer devices but results in rapid back-and-forth motions of the printhead that can cause the printer to physically overheat!

line is recommended for use with printer devices. Most programs that understand overstriking in text files will not properly deal with **line** mode.

Default: **single**.

carriage-return-style=carriage-return-type

Either **bs** or **cr**. This option applies only when one or more of the font commands is set to **overstrike** and, at the same time, **overstrike-style** is set to **line**.

- If **bs** is selected then the driver will return to the beginning of a line by emitting a sequence of backspace characters (ASCII 8).
- If **cr** is selected then the driver will return to the beginning of a line by emitting a single carriage-return character (ASCII 13).

Although **cr** is preferred as being more compact, **bs** is more general since some devices do not interpret carriage returns in the desired manner. Default: **bs**.

B.8 The HTML driver class

The **html** driver class is used to produce output for viewing in tables-capable web browsers such as Emacs' w3-mode. Its configuration is very simple. Currently, the output has a very plain format. In the future, further work may be done on improving the output appearance.

There are few options for use with the **html** driver class:

output-file=filename

File to which output should be sent. This can be an ordinary filename (i.e., "**pspp.ps**"), a pipe filename (i.e., "**|lpr**"), or stdout ("**-**"). Default: "**pspp.html**".

prologue-file=prologue-file-name

Sets the name of the PostScript prologue file. You can write your own prologue if you want to customize colors or other settings: see [Section B.8.1 \[HTML Prologue\]](#), page 131. Default: **html-prologue**.

B.8.1 The HTML prologue

HTML files that are generated by PSPP consist of two parts: a prologue and a body. The prologue is a collection of boilerplate. Only the body differs greatly between two outputs. You can tune the colors and other attributes of the output by editing the prologue.

The prologue is dumped into the output stream essentially unmodified. However, two actions are performed on its lines. First, certain lines may be omitted as specified in the prologue file itself. Second, variables are substituted.

The following lines are omitted:

1. All lines that contain three bangs in a row (!!!).
2. Lines that contain `!title`, if no title is set for the output. If a title is set, then the characters `!title` are removed before the line is output.
3. Lines that contain `!subtitle`, if no subtitle is set for the output. If a subtitle is set, then the characters `!subtitle` are removed before the line is output.

The following are the variables that are substituted. Only the variables listed are substituted; environment variables are not. See [Section B.4.2 \[Environment substitutions\]](#), [page 115](#).

generator

PSPP version as a string: ‘GNU PSPP 0.1b’, for example.

date

Date the file was created. Example: ‘Tue May 21 13:46:22 1991’.

user

Under multiuser OSes, the user’s login name, taken either from the environment variable `LOGNAME` or, if that fails, the result of the C library function `getlogin()`. Defaults to ‘nobody’.

host

System hostname as reported by `gethostname()`. Defaults to ‘nowhere’.

title

Document title as a string. This is the title specified in the PSPP syntax file.

subtitle

Document subtitle as a string.

source-file

PSPP syntax file name. Example: ‘mary96/first.stat’.

B.9 Miscellaneous configuration

The following environment variables can be used to further configure PSPP:

HOME

Used to determine the user’s home directory. No default value.

STAT_INCLUDE_PATH

Path used to find include files in PSPP syntax files. Defaults vary across operating systems:

UNIX

- `'.'`
- `'~/pspp/include'`
- `'/usr/local/lib/pspp/include'`
- `'/usr/lib/pspp/include'`
- `'/usr/local/share/pspp/include'`
- `'/usr/share/pspp/include'`

MS-DOS

- `'.'`
- `'C:\PSPP\INCLUDE'`
- `'$PATH'`

Other OSes

No default path.

STAT_PAGER

PAGER

When PSPP invokes an external pager, it uses the first of these that is defined. There is a default pager only if the person who compiled PSPP defined one.

TERM

The terminal type `termcap` or `ncurses` will use, if such support was compiled into PSPP.

STAT_OUTPUT_INIT_FILE

The basename used to search for the driver definition file. See [Section B.5 \[Output devices\]](#), page 116. See [Section B.1 \[File locations\]](#), page 113. Default: `devices`.

STAT_OUTPUT_PAPERSIZE_FILE

The basename used to search for the papersize file. See [Section B.5.5 \[papersize\]](#), page 119. See [Section B.1 \[File locations\]](#), page 113. Default: `papersize`.

STAT_OUTPUT_INIT_PATH

The path used to search for the driver definition file and the papersize file. See [Section B.1 \[File locations\]](#), page 113. Default: the standard configuration path.

TMPDIR

The directory in which PSPP stores its temporary files (used when sorting cases or concatenating large numbers of cases). Default: (UNIX) `'/tmp'`, (MS-DOS) `'\'`, (other OSes) empty string.

TEMP

TMP

Under MS-DOS only, these variables are consulted after TMPDIR, in this order.

B.10 Improving output quality

When its drivers are set up properly, PSPP can produce output that looks very good indeed. The PostScript driver, suitably configured, can produce presentation-quality output. Here are a few guidelines for producing better-looking output, regardless of output driver. Your mileage may vary, of course, and everyone has different esthetic preferences.

- Width is important in PSPP output. Greater output width leads to more readable output, to a point. Try the following to increase the output width:
 - If you're using the ASCII driver with a dot-matrix printer, figure out what you need to do to put the printer into compressed mode. Put that string into the `init-string` setting. Try to get 132 columns; 160 might be better, but you might find that print that tiny is difficult to read.
 - With the PostScript driver, try these ideas:
 - + Landscape mode.
 - + Legal-size (8.5" x 14") paper in landscape mode.
 - + Reducing font sizes. If you're using 12-point fonts, try 10 point; if you're using 10-point fonts, try 8 point. Some fonts are more readable than others at small sizes.

Try to strike a balance between character size and page width.

- Use high-quality fonts. Many public domain fonts are poor in quality. Recently, URW made some high-quality fonts available under the GPL. These are probably suitable.
- Be sure you're using the proper font metrics. The font metrics provided with PSPP may not correspond to the fonts actually being printed. This can cause bizarre-looking output.
- Make sure that you're using good ink/ribbon/toner. Darker print is easier to read.
- Use plain fonts with serifs, such as Times-Roman or Palatino. Avoid choosing italic or bold fonts as document base fonts.

Appendix C Portable File Format

These days, most computers use the same internal data formats for integer and floating-point data, if one ignores little differences like big- versus little-endian byte ordering. However, occasionally it is necessary to exchange data between systems with incompatible data formats. This is what portable files are designed to do.

Please note: Although all of the following information is correct, as far as the author has been able to ascertain, it is gleaned from examination of ASCII-formatted portable files only, so some of it may be incorrect in the general case.

C.1 Portable File Characters

Portable files are arranged as a series of lines of exactly 80 characters each. Each line is terminated by a carriage-return, line-feed sequence “new-lines”). New-lines are only used to avoid line length limits imposed by some OSes; they are not meaningful.

The file must be terminated with a ‘Z’ character. In addition, if the final line in the file does not have exactly 80 characters, then it is padded on the right with ‘Z’ characters. (The file contents may be in any character set; the file contains a description of its own character set, as explained in the next section. Therefore, the ‘Z’ character is not necessarily an ASCII ‘Z’.)

For the rest of the description of the portable file format, new-lines and the trailing ‘Z’s will be ignored, as if they did not exist, because they are not an important part of understanding the file contents.

C.2 Portable File Structure

Every portable file consists of the following records, in sequence:

- File header.
- Version and date info.
- Product identification.
- Author identification (optional).
- Subproduct identification (optional).
- Variable count.
- Case weight variable (optional).
- Variables. Each variable record may optionally be followed by a missing value record and a variable label record.
- Value labels (optional).
- Data.

Most records are identified by a single-character tag code. The file header and version info record do not have a tag.

Other than these single-character codes, there are three types of fields in a portable file: floating-point, integer, and string. Floating-point fields have the following format:

- Zero or more leading spaces.

- Optional asterisk ('*'), which indicates a missing value. The asterisk must be followed by a single character, generally a period ('.'), but it appears that other characters may also be possible. This completes the specification of a missing value.
- Optional minus sign ('-') to indicate a negative number.
- A whole number, consisting of one or more base-30 digits: '0' through '9' plus capital letters 'A' through 'T'.
- Optional fraction, consisting of a radix point ('.') followed by one or more base-30 digits.
- Optional exponent, consisting of a plus or minus sign ('+' or '-') followed by one or more base-30 digits.
- A forward slash ('/').

Integer fields take a form identical to floating-point fields, but they may not contain a fraction.

String fields take the form of an integer field having value *n*, followed by exactly *n* characters, which are the string content.

C.3 Portable File Header

Every portable file begins with a 464-byte header, consisting of a 200-byte collection of vanity splash strings, followed by a 256-byte character set translation table, followed by an 8-byte tag string.

The 200-byte segment is divided into five 40-byte sections, each of which represents the string *charset* SPSS PORT FILE in a different character set encoding, where *charset* is the name of the character set used in the file, e.g. ASCII or EBCDIC. Each string is padded on the right with spaces in its respective character set.

It appears that these strings exist only to inform those who might view the file on a screen, and that they are not parsed by SPSS products. Thus, they can be safely ignored. For those interested, the strings are supposed to be in the following character sets, in the specified order: EBCDIC, 7-bit ASCII, CDC 6-bit ASCII, 6-bit ASCII, Honeywell 6-bit ASCII.

The 256-byte segment describes a mapping from the character set used in the portable file to an arbitrary character set having characters at the following positions:

0–60

Control characters. Not important enough to describe in full here.

61–63

Reserved.

64–73

Digits '0' through '9'.

74–99

Capital letters 'A' through 'Z'.

100–125

Lowercase letters 'a' through 'z'.

126	Space.
127–130	Symbols . < (+
131	Solid vertical pipe.
132–142	Symbols & [! \$ *) ; ^ - /
143	Broken vertical pipe.
144–150	Symbols , % _ > ? ' :
151	British pound symbol.
152–155	Symbols @ ' = " .
156	Less than or equal symbol.
157	Empty box.
158	Plus or minus.
159	Filled box.
160	Degree symbol.
161	Dagger.
162	Symbol '~'.
163	En dash.
164	Lower left corner box draw.
165	Upper left corner box draw.

166	Greater than or equal symbol.
167–176	Superscript ‘0’ through ‘9’.
177	Lower right corner box draw.
178	Upper right corner box draw.
179	Not equal symbol.
180	Em dash.
181	Superscript ‘(’.
182	Superscript ‘)’.
183	Horizontal dagger (?).
184–186	Symbols ‘{ } \’.
187	Cents symbol.
188	Centered dot, or bullet.
189–255	Reserved.

Symbols that are not defined in a particular character set are set to the same value as symbol 64; i.e., to ‘0’.

The 8-byte tag string consists of the exact characters **SPSSPORT** in the portable file’s character set, which can be used to verify that the file is indeed a portable file.

C.4 Version and Date Info Record

This record does not have a tag code. It has the following structure:

- A single character identifying the file format version. The letter A represents version 0, and so on.
- An 8-character string field giving the file creation date in the format YYYYMMDD.
- A 6-character string field giving the file creation time in the format HHMMSS.

C.5 Identification Records

The product identification record has tag code ‘1’. It consists of a single string field giving the name of the product that wrote the portable file.

The author identification record has tag code ‘2’. It is optional. If present, it consists of a single string field giving the name of the person who caused the portable file to be written.

The subproduct identification record has tag code ‘3’. It is optional. If present, it consists of a single string field giving additional information on the product that wrote the portable file.

C.6 Variable Count Record

The variable count record has tag code ‘4’. It consists of two integer fields. The first contains the number of variables in the file dictionary. The purpose of the second is unknown; it contains the value 161 in all portable files examined so far.

C.7 Case Weight Variable Record

The case weight variable record is optional. If it is present, it indicates the variable used for weighting cases; if it is absent, cases are unweighted. It has tag code ‘6’. It consists of a single string field that names the weighting variable.

C.8 Variable Records

Each variable record represents a single variable. Variable records have tag code ‘7’. They have the following structure:

- Width (integer). This is 0 for a numeric variable, and a number between 1 and 255 for a string variable.
- Name (string). 1–8 characters long. Must be in all capitals.
- Print format. This is a set of three integer fields:
 - Format type (see [Section D.2 \[Variable Record\]](#), page 142).
 - Format width. 1–40.
 - Number of decimal places. 1–40.
- Write format. Same structure as the print format described above.

Each variable record can optionally be followed by a missing value record, which has tag code ‘8’. A missing value record has one field, the missing value itself (a floating-point or string, as appropriate). Up to three of these missing value records can be used.

There is also a record for missing value ranges, which has tag code ‘B’. It is followed by two fields representing the range, which are floating-point or string as appropriate. If a missing value range is present, it may be followed by a single missing value record.

Tag codes ‘9’ and ‘A’ represent `LO THRU x` and `x THRU HI` ranges, respectively. Each is followed by a single field representing `x`. If one of the ranges is present, it may be followed by a single missing value record.

In addition, each variable record can optionally be followed by a variable label record, which has tag code ‘C’. A variable label record has one field, the variable label itself (string).

C.9 Value Label Records

Value label records have tag code 'D'. They have the following format:

- Variable count (integer).
- List of variables (strings). The variable count specifies the number in the list. Variables are specified by their names. All variables must be of the same type (numeric or string).
- Label count (integer).
- List of (value, label) tuples. The label count specifies the number of tuples. Each tuple consists of a value, which is numeric or string as appropriate to the variables, followed by a label (string).

C.10 Portable File Data

The data record has tag code 'F'. There is only one tag for all the data; thus, all the data must follow the dictionary. The data is terminated by the end-of-file marker 'Z', which is not valid as the beginning of a data element.

Data elements are output in the same order as the variable records describing them. String variables are output as string fields, and numeric variables are output as floating-point fields.

Appendix D Data File Format

PSPP necessarily uses the same format for system files as do the products with which it is compatible. This chapter is a description of that format.

There are three data types used in system files: 32-bit integers, 64-bit floating points, and 1-byte characters. In this document these will simply be referred to as `int32`, `flt64`, and `char`, the names that are used in the PSPP source code. Every field of type `int32` or `flt64` is aligned on a 32-bit boundary.

The endianness of data in PSPP system files is not specified. System files output on a computer of a particular endianness will have the endianness of that computer. However, PSPP can read files of either endianness, regardless of its host computer's endianness. PSPP translates endianness for both integer and floating point numbers.

Floating point formats are also not specified. PSPP does not translate between floating point formats. This is unlikely to be a problem as all modern computer architectures use IEEE 754 format for floating point representation.

The PSPP system-missing value is represented by the largest possible negative number in the floating point format; in C, this is most likely `-DBL_MAX`. There are two other important values used in missing values: `HIGHEST` and `LOWEST`. These are represented by the largest possible positive number (probably `DBL_MAX`) and the second-largest negative number. The latter must be determined in a system-dependent manner; in IEEE 754 format it is represented by value `0xffeffffffffffffe`.

System files are divided into records. Each record begins with an `int32` giving a numeric record type. Individual record types are described below:

D.1 File Header Record

The file header is always the first record in the file.

```
struct sysfile_header
{
    char          rec_type[4];
    char          prod_name[60];
    int32         layout_code;
    int32         case_size;
    int32         compressed;
    int32         weight_index;
    int32         ncases;
    flt64         bias;
    char          creation_date[9];
    char          creation_time[8];
    char          file_label[64];
    char          padding[3];
};

char rec_type[4];
```

Record type code. Always set to `'$FL2'`. This is the only record for which the record type is not of type `int32`.

`char prod_name[60];`
Product identification string. This always begins with the characters '@(#) SPSS DATA FILE'. PSPP uses the remaining characters to give its version and the operating system name; for example, 'GNU pspp 0.1.4 - sparc-sun-solaris2.5.2'. The string is truncated if it would be longer than 60 characters; otherwise it is padded on the right with spaces.

`int32 layout_code;`
Always set to 2. PSPP reads this value to determine the file's endianness.

`int32 case_size;`
Number of data elements per case. This is the number of variables, except that long string variables add extra data elements (one for every 8 characters after the first 8). When reading system files, PSPP will use this value unless it is set to -1, in which case it will determine the number of data elements by context. When writing system files PSPP always uses this value.

`int32 compressed;`
Set to 1 if the data in the file is compressed, 0 otherwise.

`int32 weight_index;`
If one of the variables in the data set is used as a weighting variable, set to the index of that variable. Otherwise, set to 0.

`int32 ncases;`
Set to the number of cases in the file if it is known, or -1 otherwise.

In the general case it is not possible to determine the number of cases that will be output to a system file at the time that the header is written. The way that this is dealt with is by writing the entire system file, including the header, then seeking back to the beginning of the file and writing just the `ncases` field. For 'files' in which this is not valid, the seek operation fails. In this case, `ncases` remains -1.

`flt64 bias;`
Compression bias. Always set to 100. The significance of this value is that only numbers between $(1 - \text{bias})$ and $(251 - \text{bias})$ can be compressed.

`char creation_date[9];`
Set to the date of creation of the system file, in 'dd mmm yy' format, with the month as standard English abbreviations, using an initial capital letter and following with lowercase. If the date is not available then this field is arbitrarily set to '01 Jan 70'.

`char creation_time[8];`
Set to the time of creation of the system file, in 'hh:mm:ss' format and using 24-hour time. If the time is not available then this field is arbitrarily set to '00:00:00'.

`char file_label[64];`
Set the the file label declared by the user, if any. Padded on the right with spaces.

```
char padding[3];
```

Ignored padding bytes to make the structure a multiple of 32 bits in length.
Set to zeros.

D.2 Variable Record

Immediately following the header must come the variable records. There must be one variable record for every variable and every 8 characters in a long string beyond the first 8; i.e., there must be exactly as many variable records as the value specified for `case_size` in the file header record.

```
struct sysfile_variable
{
    int32          rec_type;
    int32          type;
    int32          has_var_label;
    int32          n_missing_values;
    int32          print;
    int32          write;
    char           name[8];

    /* The following two fields are present
       only if has_var_label is 1. */
    int32          label_len;
    char           label[/* variable length */];

    /* The following field is present only
       if n_missing_values is not 0. */
    float64        missing_values[/* variable length*/];
};
```

`int32 rec_type;`
Record type code. Always set to 2.

`int32 type;`
Variable type code. Set to 0 for a numeric variable. For a short string variable or the first part of a long string variable, this is set to the width of the string. For the second and subsequent parts of a long string variable, set to -1, and the remaining fields in the structure are ignored.

`int32 has_var_label;`
If this variable has a variable label, set to 1; otherwise, set to 0.

`int32 n_missing_values;`
If the variable has no missing values, set to 0. If the variable has one, two, or three discrete missing values, set to 1, 2, or 3, respectively. If the variable has a range for missing variables, set to -2; if the variable has a range for missing variables plus a single discrete value, set to -3.

`int32 print;`
Print format for this variable. See below.

int32 write;

Write format for this variable. See below.

char name[8];

Variable name. The variable name must begin with a capital letter or the at-sign ('@'). Subsequent characters may also be octothorpes ('#'), dollar signs ('\$'), underscores ('_'), or full stops ('.'). The variable name is padded on the right with spaces.

int32 label_len;

This field is present only if **has_var_label** is set to 1. It is set to the length, in characters, of the variable label, which must be a number between 0 and 120.

char label[/ * variable length * /];

This field is present only if **has_var_label** is set to 1. It has length **label_len**, rounded up to the nearest multiple of 32 bits. The first **label_len** characters are the variable's variable label.

flt64 missing_values[/ * variable length * /];

This field is present only if **n_missing_values** is not 0. It has the same number of elements as the absolute value of **n_missing_values**. For discrete missing values, each element represents one missing value. When a range is present, the first element denotes the minimum value in the range, and the second element denotes the maximum value in the range. When a range plus a value are present, the third element denotes the additional discrete missing value. HIGHEST and LOWEST are indicated as described in the chapter introduction.

The **print** and **write** members of **sysfile_variable** are output formats coded into **int32** types. The LSB (least-significant byte) of the **int32** represents the number of decimal places, and the next two bytes in order of increasing significance represent field width and format type, respectively. The MSB (most-significant byte) is not used and should be set to zero.

Format types are defined as follows:

0	Not used.
1	A
2	AHEX
3	COMMA
4	DOLLAR
5	F
6	IB
7	PIBHEX
8	P
9	PIB
10	PK

11	RB
12	RBHEX
13	Not used.
14	Not used.
15	Z
16	N
17	E
18	Not used.
19	Not used.
20	DATE
21	TIME
22	DATETIME
23	ADATE
24	JDATE
25	DTIME
26	WKDAY
27	MONTH
28	MOYR
29	QYR
30	WKYR
31	PCT
32	DOT
33	CCA
34	CCB
35	CCC
36	CCD
37	CCE
38	EDATE
39	SDATE

D.3 Value Label Record

Value label records must follow the variable records and must precede the header termination record. Other than this, they may appear anywhere in the system file. Every value label record must be immediately followed by a label variable record, described below.

Value label records begin with `rec_type`, an `int32` value set to the record type of 3. This is followed by `count`, an `int32` value set to the number of value labels present in this record.

These two fields are followed by a series of `count` tuples. Each tuple is divided into two fields, the value and the label. The first of these, the value, is composed of a 64-bit value, which is either a `flt64` value or up to 8 characters (padded on the right to 8 bytes) denoting a short string value. Whether the value is a `flt64` or a character string is not defined inside the value label record.

The second field in the tuple, the label, has variable length. The first `char` is a count of the number of characters in the value label. The remainder of the field is the label itself. The field is padded on the right to a multiple of 64 bits in length.

D.4 Value Label Variable Record

Every value label variable record must be immediately preceded by a value label record, described above.

```
struct sysfile_value_label_variable
{
    int32          rec_type;
    int32          count;
    int32          vars[/* variable length */];
};
```

`int32 rec_type;`
Record type. Always set to 4.

`int32 count;`
Number of variables that the associated value labels from the value label record are to be applied.

`int32 vars[/* variable length】;`
A list of variables to which to apply the value labels. There are `count` elements.

D.5 Document Record

There must be no more than one document record per system file. Document records must follow the variable records and precede the dictionary termination record.

```
struct sysfile_document
{
    int32          rec_type;
    int32          n_lines;
    char           lines[/* variable length */][80];
};
```

```
int32 rec_type;
    Record type. Always set to 6.

int32 n_lines;
    Number of lines of documents present.

char lines[/* variable length */ [80];
    Document lines. The number of elements is defined by n_lines. Lines shorter
    than 80 characters are padded on the right with spaces.
```

D.6 Machine int32 Info Record

There must be no more than one machine `int32` info record per system file. Machine `int32` info records must follow the variable records and precede the dictionary termination record.

```
struct sysfile_machine_int32_info
{
    /* Header. */
    int32      rec_type;
    int32      subtype;
    int32      size;
    int32      count;

    /* Data. */
    int32      version_major;
    int32      version_minor;
    int32      version_revision;
    int32      machine_code;
    int32      floating_point_rep;
    int32      compression_code;
    int32      endianness;
    int32      character_code;
};

int32 rec_type;
    Record type. Always set to 7.

int32 subtype;
    Record subtype. Always set to 3.

int32 size;
    Size of each piece of data in the data part, in bytes. Always set to 4.

int32 count;
    Number of pieces of data in the data part. Always set to 8.

int32 version_major;
    PSPP major version number. In version x.y.z, this is x.

int32 version_minor;
    PSPP minor version number. In version x.y.z, this is y.

int32 version_revision;
    PSPP version revision number. In version x.y.z, this is z.
```

```
int32 machine_code;
    Machine code. PSPP always set this field to value to -1, but other values may
    appear.

int32 floating_point_rep;
    Floating point representation code. For IEEE 754 systems this is 1. IBM 370
    sets this to 2, and DEC VAX E to 3.

int32 compression_code;
    Compression code. Always set to 1.

int32 endianness;
    Machine endianness. 1 indicates big-endian, 2 indicates little-endian.

int32 character_code;
    Character code. 1 indicates EBCDIC, 2 indicates 7-bit ASCII, 3 indicates 8-bit
    ASCII, 4 indicates DEC Kanji.
```

D.7 Machine flt64 Info Record

There must be no more than one machine `flt64` info record per system file. Machine `flt64` info records must follow the variable records and precede the dictionary termination record.

```
struct sysfile_machine_flt64_info
{
    /* Header. */
    int32      rec_type;
    int32      subtype;
    int32      size;
    int32      count;

    /* Data. */
    flt64      sysmis;
    flt64      highest;
    flt64      lowest;
};

int32 rec_type;
    Record type. Always set to 7.

int32 subtype;
    Record subtype. Always set to 4.

int32 size;
    Size of each piece of data in the data part, in bytes. Always set to 4.

int32 count;
    Number of pieces of data in the data part. Always set to 3.

flt64 sysmis;
    The system missing value.

flt64 highest;
    The value used for HIGHEST in missing values.
```

flt64 lowest;

The value used for LOWEST in missing values.

D.8 Auxilliary Variable Parameter Record

There must be no more than one auxilliary variable parameter record per system file. This record must follow the variable records and precede the dictionary termination record.

```
struct sysfile_aux_var_parameter
{
    /* Header. */
    int32      rec_type;
    int32      subtype;
    int32      size;
    int32      count;

    /* Data. */
    struct aux_params  aux_params[/* variable length */];
};
```

int32 rec_type;
Record type. Always set to 7.

int32 subtype;
Record subtype. Always set to 11.

int32 size;
The size int32. Always set to 4.

int32 count;
The total number of bytes in `aux_params` divided by 3.

struct aux_params aux_params[];
An array of `struct aux_params`. The order of the elements corresponds to the order of the variables in the Variable Records. The `struct aux_params` type is defined as follows:

```
struct aux_params
{
    int32 measure;
    int32 width;
    int32 alignment;
};
```

int32 measure
The measurement type of the variable:

0	Nominal Scale
1	Ordinal Scale
2	Continuous Scale

int32 width
The width of the display column for the variable in characters.

int32 alignment

The alignment of the variable for display purposes:

0	Left aligned
1	Right aligned
2	Centre aligned

D.9 Long Variable Names Record

There must be no more than one long variable names record per system file. This record must follow the variable records and precede the dictionary termination record.

```
struct sysfile_long_variable_names
{
    /* Header. */
    int32      rec_type;
    int32      subtype;
    int32      size;
    int32      count;

    /* Data. */
    char      var_name_pairs[/* variable length */];
};
```

int32 rec_type;
Record type. Always set to 7.

int32 subtype;
Record subtype. Always set to 13.

int32 size;
The size of each element in the `var_name_pairs` member. Always set to 1.

int32 count;
The total number of bytes in `var_name_pairs`.

char var_name_pairs[/* variable length】;
A list of *key-value* tuples, where *key* is the name of a variable, and *value* is its long variable name. The *key* field is at most 8 bytes long and must match the name of a variable which appears in the variable record See [Section D.2 \[Variable Record\]](#), page 142. The *value* field is at most 64 bytes long. The *key* and *value* fields are separated by a '=' byte. Each tuple is separated by a byte whose value is 09. There is no trailing separator following the last tuple. The total length is `count` bytes.

D.10 Miscellaneous Informational Records

Miscellaneous informational records must follow the variable records and precede the dictionary termination record.

Miscellaneous informational records are ignored by PSPP when reading system files. They are not written by PSPP when writing system files.

```

struct sysfile_misc_info
{
    /* Header. */
    int32      rec_type;
    int32      subtype;
    int32      size;
    int32      count;

    /* Data. */
    char      data[/* variable length */];
};

int32 rec_type;
    Record type. Always set to 7.

int32 subtype;
    Record subtype. May take any value. According to Aapi Hämäläinen, value 5
    indicates a set of grouped variables and 6 indicates date info (probably related
    to USE).

int32 size;
    Size of each piece of data in the data part. Should have the value 4 or 8, for
    int32 and flt64, respectively.

int32 count;
    Number of pieces of data in the data part.

char data[/* variable length */];
    Arbitrary data. There must be size times count bytes of data.

```

D.11 Dictionary Termination Record

The dictionary termination record must follow all other records, except for the actual cases, which it must precede. There must be exactly one dictionary termination record in every system file.

```

struct sysfile_dict_term
{
    int32      rec_type;
    int32      filler;
};

int32 rec_type;
    Record type. Always set to 999.

int32 filler;
    Ignored padding. Should be set to 0.

```

D.12 Data Record

Data records must follow all other records in the data file. There must be at least one data record in every system file.

The format of data records varies depending on whether the data is compressed. Regardless, the data is arranged in a series of 8-byte elements.

When data is not compressed, Every case is composed of `case_size` of these 8-byte elements, where `case_size` comes from the file header record (see [Section D.1 \[File Header Record\]](#), page 140). Each element corresponds to the variable declared in the respective variable record (see [Section D.2 \[Variable Record\]](#), page 142). Numeric values are given in `flt64` format; string values are literal characters string, padded on the right when necessary.

Compressed data is arranged in the following manner: the first 8-byte element in the data section is divided into a series of 1-byte command codes. These codes have meanings as described below:

- 0 Ignored. If the program writing the system file accumulates compressed data in blocks of fixed length, 0 bytes can be used to pad out extra bytes remaining at the end of a fixed-size block.

- 1 through 251 These values indicate that the corresponding numeric variable has the value (`code - bias`) for the case being read, where `code` is the value of the compression code and `bias` is the variable `compression_bias` from the file header. For example, code 105 with bias 100.0 (the normal value) indicates a numeric variable of value 5.

- 252 End of file. This code may or may not appear at the end of the data stream. PSPP always outputs this code but its use is not required.

- 253 This value indicates that the numeric or string value is not compressible. The value is stored in the 8-byte element following the current block of command bytes. If this value appears twice in a block of command bytes, then it indicates the second element following the command bytes, and so on.

- 254 Used to indicate a string value that is all spaces.

- 255 Used to indicate the system-missing value.

When the end of the first 8-byte element of command bytes is reached, any blocks of non-compressible values are skipped, and the next element of command bytes is read and interpreted, until the end of the file is reached.

Appendix E q2c Input Format

PSPP statistical procedures have a bizarre and somewhat irregular syntax. Despite this, a parser generator has been written that adequately addresses many of the possibilities and tries to provide hooks for the exceptional cases. This parser generator is named **q2c**.

E.1 Invoking q2c

`q2c input.q output.c`

q2c translates a `.q` file into a `.c` file. It takes exactly two command-line arguments, which are the input file name and output file name, respectively. **q2c** does not accept any command-line options.

E.2 q2c Input Structure

q2c input files are divided into two sections: the grammar rules and the supporting code. The *grammar rules*, which make up the first part of the input, are used to define the syntax of the statistical procedure to be parsed. The *supporting code*, following the grammar rules, are copied largely unchanged to the output file, except for certain escapes.

The most important lines in the grammar rules are used for defining procedure syntax. These lines can be prefixed with a dollar sign (`'$'`), which prevents Emacs' CC-mode from munging them. Besides this, a bang (`'!'`) at the beginning of a line causes the line, minus the bang, to be written verbatim to the output file (useful for comments). As a third special case, any line that begins with the exact characters `/* *INDENT` is ignored and not written to the output. This allows `.q` files to be processed through **indent** without being munged.

The syntax of the grammar rules themselves is given in the following sections.

The supporting code is passed into the output file largely unchanged. However, the following escapes are supported. Each escape must appear on a line by itself.

`/* (header) */`

Expands to a series of C `#include` directives which include the headers that are required for the parser generated by **q2c**.

`/* (decls scope) */`

Expands to C variable and data type declarations for the variables and **enums** input and output by the **q2c** parser. *scope* must be either **local** or **global**. **local** causes the declarations to be output as function locals. **global** causes them to be declared as **static** module variables; thus, **global** is a bit of a misnomer.

`/* (parser) */`

Expands to the entire parser. Must be enclosed within a C function.

`/* (free) */`

Expands to a set of calls to the **free** function for variables declared by the parser. Only needs to be invoked if subcommands of type **string** are used in the grammar rules.

E.3 Grammar Rules

The grammar rules describe the format of the syntax that the parser generated by **q2c** will understand. The way that the grammar rules are included in **q2c** input file are described above.

The grammar rules are divided into tokens of the following types:

Identifier (ID)

An identifier token is a sequence of letters, digits, and underscores ('_'). Identifiers are *not* case-sensitive.

String (STRING)

String tokens are initiated by a double-quote character ("") and consist of all the characters between that double quote and the next double quote, which must be on the same line as the first. Within a string, a backslash can be used as a "literal escape". The only reasons to use a literal escape are to include a double quote or a backslash within a string.

Special character

Other characters, other than white space, constitute tokens in themselves.

The syntax of the grammar rules is as follows:

```
grammar-rules ::= ID : subcommands .
subcommands ::= subcommand
               ::= subcommands ; subcommand
```

The syntax begins with an ID or STRING token that gives the name of the procedure to be parsed. The rest of the syntax consists of subcommands separated by semicolons (;) and terminated with a full stop (.).

```
subcommand ::= sbc-options ID sbc-defn
sbc-options ::=
               ::= sbc-option
               ::= sbc-options sbc-options
sbc-option  ::= *
               ::= +
               ::= ^
sbc-defn    ::= opt-prefix = specifiers
               ::= [ ID ] = array-sbc
               ::= opt-prefix = sbc-special-form
opt-prefix  ::=
               ::= ( ID )
```

Each subcommand can be prefixed with one or more option characters. An asterisk (*) is used to indicate the default subcommand; the keyword used for the default subcommand can be omitted in the PSPP syntax file. A plus sign (+) is used to indicate that a subcommand can appear more than once; if it is not present then that subcommand can appear no more than once. A carat sign (^) is used to indicate that a subcommand must appear at least once.

The subcommand name appears after the option characters.

There are three forms of subcommands. The first and most common form simply gives an equals sign (=) and a list of specifiers, which can each be set to a single setting. The

second form declares an array, which is a set of flags that can be individually turned on by the user. There are also several special forms that do not take a list of specifiers.

Arrays require an additional ID argument. This is used as a prefix, prepended to the variable names constructed from the specifiers. The other forms also allow an optional prefix to be specified.

```
array-sbc ::= alternatives
          ::= array-sbc , alternatives
alternatives ::= ID
             ::= alternatives | ID
```

An array subcommand is a set of Boolean values that can independently be turned on by the user, listed separated by commas (','),. If an value has more than one name then these names are separated by pipes ('|').

```
specifiers ::= specifier
            ::= specifiers , specifier
specifier ::= opt-id : settings
opt-id ::=
         ::= ID
```

Ordinary subcommands (other than arrays and special forms) require a list of specifiers. Each specifier has an optional name and a list of settings. If the name is given then a correspondingly named variable will be used to store the user's choice of setting. If no name is given then there is no way to tell which setting the user picked; in this case the settings should probably have values attached.

```
settings ::= setting
          ::= settings / setting
setting ::= setting-options ID setting-value
setting-options ::=
               ::= *
               ::= !
               ::= * !
```

Individual settings are separated by forward slashes ('/'). Each setting can be as little as an ID token, but options and values can optionally be included. The '*' option means that, for this setting, the ID can be omitted. The '!' option means that this option is the default for its specifier.

```
setting-value ::=
              ::= ( setting-value-2 )
              ::= setting-value-2
setting-value-2 ::= setting-value-options setting-value-type : ID
                  setting-value-restriction
setting-value-options ::=
                     ::= *
setting-value-type ::= N
                   ::= D
setting-value-restriction ::=
                          ::= , STRING
```

Settings may have values. If the value must be enclosed in parentheses, then enclose the value declaration in parentheses. Declare the setting type as ‘n’ or ‘d’ for integer or floating point type, respectively. The given ID is used to construct a variable name. If option ‘*’ is given, then the value is optional; otherwise it must be specified whenever the corresponding setting is specified. A “restriction” can also be specified which is a string giving a C expression limiting the valid range of the value. The special escape %s should be used within the restriction to refer to the setting’s value variable.

```

sbc-special-form ::= VAR
                  ::= VARLIST varlist-options
                  ::= INTEGER opt-list
                  ::= DOUBLE opt-list
                  ::= PINT
                  ::= STRING (the literal word STRING) string-options
                  ::= CUSTOM
varlist-options ::=
                ::= ( STRING )
opt-list ::=
              ::= LIST
string-options ::=
               ::= ( STRING STRING )

```

The special forms are of the following types:

VAR

A single variable name.

VARLIST

A list of variables. If given, the string can be used to provide PV_* options to the call to `parse_variables`.

INTEGER

A single integer value.

INTEGER LIST

A list of integers separated by spaces or commas.

DOUBLE

A single floating-point value.

DOUBLE LIST

A list of floating-point values.

PINT

A single positive integer value.

STRING

A string value. If the options are given then the first string is an expression giving a restriction on the value of the string; the second string is an error message to display when the restriction is violated.

CUSTOM

A custom function is used to parse this subcommand. The function must have prototype `int custom_name (void)`. It should return 0 on failure (when it has already issued an appropriate diagnostic), 1 on success, or 2 if it fails and the calling function should issue a syntax error on behalf of the custom handler.

Appendix F GNU Free Documentation License

Version 1.2, November 2002

Copyright © 2000,2001,2002 Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none. The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and

that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements.”

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called

an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

F.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts. A copy of the license is included in the section entitled ‘‘GNU
Free Documentation License’’.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with...Texts.” line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.