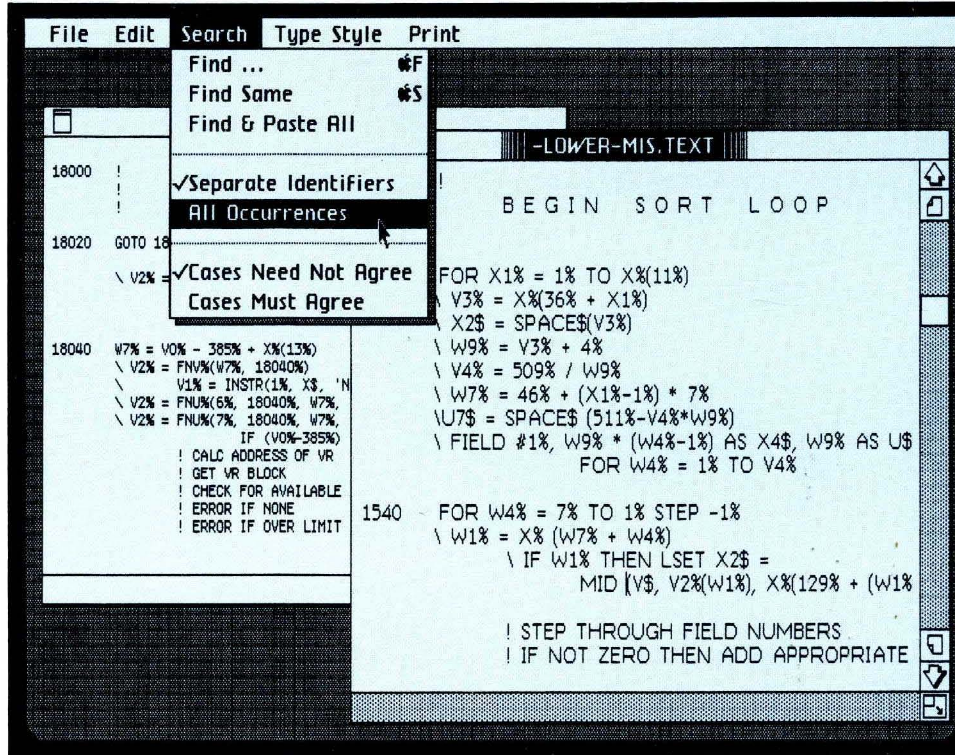
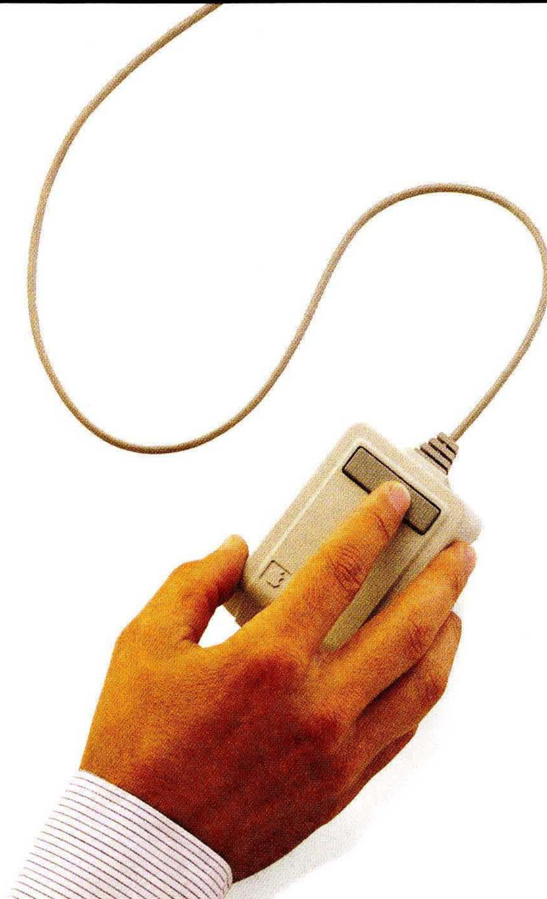
Lisa® BASIC-Plus 2.0 Language





BASIC-Plus for the Lisa

Release 2.0 Notes

What's in the BASIC-Plus Release Notes?

These notes describe situations that were brought to our attention after it was too late to document them in the BASIC-Plus manuals.

Insert these notes in the back of their respective manuals, so that you can refer to them as necessary. Included in these notes are revised versions of the *Workshop User's Guide* Appendix B and the *BASIC-Plus User's Guide* Appendix E to replace the copies bound in your manuals; take a moment now to make the substitutions.

If you have a question or a problem that you can't find the answer to, either in the manuals or in these notes, you should call the Lisa Telephone Support Line, (800) 553-4000.

Manual Chapter	Release Note
Workshop Chapter 1	To install the Pascal language and Workshop software from the set of micro diskettes packaged in your language manual binder, refer to Installing the Office System Software in Appendix G, Set Up Procedures, in the *Lisa 2 Owner's Guide*. If you plan to use the Office System, you must first install the Office System 2.0 micro diskettes. You do not need to install the Office System software if you intend to do only language development work. Before you insert the micro diskettes, make sure you can see the red tabs from the front of the micro diskettes. Start installing with steps 1, 2, and 3 on page G31. Then follow this sequence: 4> Turn the Lisa on by pressing the on-off button once. After a few seconds, you'll hear a click; immediately press the spacebar. 5> The Lisa goes through a self-test. When a menu of symbols appears in the upper left-hand corner of the screen, press and hold down the Apple key while you type a 2 -- on the main keyboard, not on the numeric keypad. 6> When the main menu shown on page G32 appears, click the mouse once on the Install box. 7> When the alert box with the message "The Lisa is installing startup software version 2.0" appears, click Don't Erase. When the first micro diskette is installed, it will eject. Continue installation by following the *Owner's Guide* instructions from step 6, inserting the remaining language diskettes in order.
Workshop Chapter 1	After successfully adding Pascal to a ProFile containing the Office System, if the system is merely allowed to reboot, the default of the Environments window will cause the Workshop shell to start up. To cause the initialization to pause at the Environments window in order to examine or change the default, press the space bar after the machine self-test, while the hourglass icon is showing.
Workshop Chapter 1	If you have just printed anything on a daisy wheel printer from the Office System, and you return to the Workshop using the Environments window, printing to logical device "-printer" will be garbled until the printer is switched off and then on again.

Manual Chapter	Release Note
Workshop Chapter 1	The print commands of the Editor always use the logical device "-printer" set in the System Manager. Choosing Daisy Wheel Printer or Dot Matrix Printer from the Print menu does not change the system's configuration, but only adjusts the Editor to the intended device.
Workshop Chapter 1	Any program intended to run as a background process (MakeBackgroundProcess) must include frequent and judicious calls to the Operating System procedure Yield_CPU. Hence, system utilities should never be run in the background. Also, a background process should not have any interaction with the console, and it cannot pull events from the hardware event queue.
Workshop Chapter 2	Designate user files with the pathname "SHELL." only if you want them to appear in the Environments window as an alternative shell.
Workshop Chapter 2	You cannot directly rename a file to a name that differs from the original only in the case of the characters, because the internal representation of the names is the same. Instead, rename the file to a temporary name, and then change that to the name you want.
Workshop Chapter 2	If you unmount the prefix volume by ejecting the diskette, Scavenging the volume, or using the Unmount command, the boot volume automatically becomes the prefix volume.
Workshop Chapter 2	Assume that a file FOO.TEXT has been damaged and no longer has the internal representation of a textfile. If the user enters the File Manager and tries to COPY the file to -PRINTER, the system generates a bus error and enters the Debugger.
Workshop Chapter 3	The Output Redirect function of the System Manager does not correctly handle screen output that uses GOTOXY, for example, screen output done by the File Manager when listing wildcard matches. This results in redirected output to the printer being overwritten on one line.
Workshop Chapter 3	Use "-printer" instead of "-RS232-B" when redirecting output to the printer.

Manual Chapter	Release Note
Workshop Chapter 3	If you change the name of a suspended file -- such as the Pascal compiler -- and attempt to manage the process from the System Manager, the new name appears in the pathname, but you must still use the *old* name to kill the process.
Workshop Chapter 4	The Editor changes the creation date of a text file to the current date each time the file is modified.
Workshop Chapter 4	If the initialization of the Editor fails due to lack of disk space (error 309), and space on the disk is then made free, the next attempt to start the Editor will also fail (error 304). You must enter the Process Manager of the System Manager, KILL the Editor process, and then retry.
Workshop Chapter 4	The language processors, Editor, and other utilities of the Workshop expect as input a standard .TEXT file. The internal structure of a text file in a block-structured device is described in the Lisa Pascal Reference Manual: • Each page (two 512-byte blocks) contains some number of complete lines of text and is padded with null characters (ASCII 0) after the last line as necessary to complete the page. • Two 512-byte header blocks are also present at the beginning of the file. These may or may not contain information. • A sequence of spaces (ASCII 32 decimal, $20 hexadecimal) can be compressed into a 2-byte code namely, a DLE character (ASCII 16 decimal, $10 hexadecimal), followed by a byte containing the value 32 decimal plus the number of spaces represented.
Workshop Chapter 4	The file name "PAPER.TEXT" is reserved for the default stationery template of the Editor and should not be used for other purposes.

Manual Chapter	Release Note
Workshop Chapter 4	Attempting to enter or paste more than about 1000 characters into one line causes a bus error. If you have a Debugger, type <g> to recover and exit the Workshop shell before running the Editor again, otherwise no menus appear and you must use NMI and OSQUIT.
Workshop Chapter 4	A triple-click will not select the last line in a file unless that line ends with a carriage return.
Workshop Chapter 4	If you are working on many files -- or a few large files -- and the Editor becomes sluggish, save and put away the files. Then either exit the Workshop shell and run the Workshop shell again, or use the DeleteResident command of the Manage Process subsystem of the System Manager to temporarily delete the Editor from the list of resident processes.
Workshop Chapter 4	When using the Tear Off Stationery command, type in the volume name if it differs from your boot volume.
Workshop Chapters 4 and 10	Cursor residue might be left on the screen in the Editor and the Transfer program, especially after an error message has appeared.
Workshop Chapters 4 and 10	The names of files created by the Editor and Transfer will be changed to be all upper case, regardless of how they are typed in.
Workshop Chapter 7	If multiple errors occur during a link, due an to attempt to link regular units with intrinsic units, the Linker will terminate after reporting only the first error.
Workshop Chapter 7	When the Linker detects the error of duplicate entry names -- for example after it reads the same file twice -- the error message may be difficult to interpret because it is formatted incorrectly.
Workshop Chapter 7	If an intrinsic unit is linked but not needed (i.e. no units in its library file are used), the Linker generates error 24: unexpected block type in IU file.

Manual Chapter	Release Note
Workshop Chapter 8	For the Debugger, >PR 2 is print to SLOT2CHAN2, not SLOT2CHAN1. Upper and lower are reversed in the manual.
Workshop Chapter 9	The exec file preprocessor does not have an easy way to input single spaces, even though these are required to respond to some Workshop messages: While waiting for a space input, the rest of the exec file is consumed without effect. Either set up your exec files so they don't require space inputs, or eliminate all spaces except the one you want and use the no-space option in the preprocessor.
Workshop Chapter 9	In an exec file, an attempt to pass a literal "%" to a program such as CODESIZE will not work.
Workshop Chapter 10	Display of error message 647 while you are using the Transfer utility probably indicates that after a timeout the program has failed to receive the appropriate handshake from the host.
Workshop Chapter 10	If you type any key during "Playback from what file" in the Transfer program, the playback will abort.
Workshop Chapter 10	If you use the Transfer program to make contact with a host computer, and you exit the program without logging off explicitly, the connection will not be automatically terminated. This is usually a convenience, but might not meet user expectations.
Workshop Chapter 10	When the Workshop shell is initialized, all serial ports are configured by default as if they were printers (e.g., 9600 baud, DTR handshake, automatic linefeed insertion), whether or not they are listed as such by Preferences. If you subsequently use and then exit the Transfer program, the printer configuration is restored automatically for ONLY those ports listed in Preferences as printers; others will retain the properties set by the Transfer program. The Editor will not reconfigure ports that have been changed by PortConfig.

Manual Chapter	Release Note
Workshop Chapter 10	To terminate recording to a file opened by the Transfer program during "Record to", open the Control menu and again select "Record to". This terminates recording and closes the file. Note that, unlike the Editor, Transfer does not automatically insert a carriage return at the end of the file. If you use this recording to capture text such as a source program, and the language processor (such as BASIC-Plus) expects to see a carriage return at the end of the file, attempting to run the raw recorded text might cause the system to hang.
Workshop Chapter 10	The manual states that the default handshake in the Transfer program is XOn/XOff. The correct default is None.
Workshop Chapter 11	Because most programs do not allow you to eject a disk in while they are running, plan ahead in large transactions, such as mass transfers, to allow a pause for changing disks.
Workshop Appendix B	ASCII characters in the range hex 20 through hex 7E are supported for screen display, for printing on a dot matrix printer, and for printing on a daisy wheel printer with the following print wheels: • Gothic, 15 pitch • Prestige Elite, 12 pitch • Courier, 10 pitch • Boldface/Executive, PS. Printing ASCII characters to a daisy wheel printer is not supported for the three print wheels with Modern type styles. The character set in the Appendix should show the full Lisa character set. All of the additional characters can be displayed on the screen. Selected subsets can be printed on dot matrix and daisy wheel printers. A new page B-1 is attached; take a moment now to make the substitution.

Manual Chapter	Release Note
Workshop Appendix C	If you wish to to position the cursor at coordinates (x,y) on the screen, use the two-character sequence <ESC>= (HEX 1B-3D, decimal 27-61) followed by the screen's y coordinate and then the screen's x coordinate -- note the order of the supplied arguments. The range for the screen's y-axis is from ASCII decimal 32 (<SPACE> on the keyboard), representing a screen coordinate of 0, through ASCII decimal 63 (? on the keyboard), representing a screen coordinate of 31. The range for the screen's x-axis is from ASCII decimal 32 (<SPACE> on the keyboard), representing a screen coordinate of 0, through ASCII decimal 119 (w on the keyboard), representing a screen coordinate of 87. If you supply coordinates outside these ranges, a bus error may result. Refer to the revised Appendix B, supplied with these release notes, for a complete chart of character equivalents. For example, in BASIC, either of the two statements below would place the cursor at position x=0, y=1. PRINT CHR$(27); "="; "!"; " "; or PRINT CHR$(27); CHR$(61); CHR$(33); CHR$(32);

Manual Chapter	Release Note
BASIC-Plus Chapter 3	The system error variable ERR is reset to zero after returning to the command line. If you want to preserve the value of this error return, use ON ERROR processing, and store the value in another variable.
BASIC-Plus Chapter 3	Renumbering sometimes results in a file beginning with BTEMP being left on the disk. Ignore or delete the BTEMP file.
BASIC-Plus Chapter 3	When you use Apple period to terminate your program, the command is sometimes included in the input stream of the next line, giving unpredictable results. Also, if you are redirecting your output to a dot-matrix printer, the character generated will turn on wide print. To restore normal print width, turn off the printer and turn it back on again.
BASIC-Plus Chapter 3	BASIC-Plus attempts to run a program even after detecting syntax error(s).
BASIC-Plus Chapter 3	If BASIC-Plus is processing a single operation with a long computation time (for example, an INV statement), the Lisa might not respond quickly to Apple period.
BASIC-Plus Chapter 3	While line number zero is a legal number, renumbering starts at the first line with a number greater than zero.
BASIC-Plus Chapter 4	The manual states that the smallest representable number is ± 4.9E-324. The smallest representable number is really ± 4.94066E-324. All other representable numbers are integral multiples of that number.
BASIC-Plus Chapter 5	A whole number can be printed out to a maximum of 12 places before going to scientific notation.
BASIC-Plus Chapter 5	You can use the following to terminate input, in addition to CR. Apple L — FF — (ASCII 12 decimal, $0C hex) CLEAR — ESC — (ASCII 27 decimal, $1B hex) Apple J — LF — (ASCII 10 decimal, $0A hex)
BASIC-Plus Chapter 5	BASIC-Plus doesn't append any further characters following a line terminator character at the end of a PRINT or INPUT statement.

Manual Chapter	Release Note
BASIC-Plus Chapter 5	If you input from a nonzero channel, the prompt will not be printed.
BASIC-Plus Chapter 5	The BASIC Interpreter doesn't differentiate between vertical and horizontal screen control characters. Refer to the Workshop manual, Appendix C, for information on vertical screen control.
BASIC-Plus Chapter 5	You can PRINT and INPUT only to text files (files that end in .text) and to the devices: -console, -printer, -keyboard.
BASIC-Plus Chapter 5	Spaces between words in DATA statements are thrown away. To preserve spaces, use quoted strings. The first character in an unquoted string variable in a DATA statement is not converted to uppercase, although the rest of the string is. Use CVT$$ with a value of 32% to convert lowercase to uppercase.
BASIC-Plus Chapter 5	BASIC-Plus supports six print zones.
BASIC-Plus Chapters 5 and 11	When using a GET or INPUT statement, -keyboard doesn't echo input. -console does echo input.
BASIC-Plus Chapter 7	To achieve the best performance in FOR loops and other constructs, use integer variables. For instance: 100 FOR I% = 1% To 5000% 110 NEXT I% executes approximately ten times faster than 100 FOR I = 1 To 5000 110 NEXT I
BASIC-Plus Chapter 8	The use of FOR modifiers in immediate mode will result in a fatal error.
BASIC-Plus Chapter 9	The system variables NUM and NUM2, which contain the size of a two dimensional array, were omitted from the index.

Manual Chapter	Release Note
BASIC-Plus Chapter 9	The syntax diagram for DIM does not provide for multiple array dimensioning in a single dimension statement. The written example is correct to show multiple array definitions.
BASIC-Plus Chapter 9	The statements MAT INPUT and MAT READ do not input into the zeroth row or column. Matrix redimensioning causes the contents of the zeroth row and column to be unpredictable.
BASIC-Plus Chapter 10	The "N" integer argument returns the DATE from the base of January 1, 1980. The formula used to translate between N and the date is (day of year) + [(number of years since 1980) * 1000]
BASIC-Plus Chapter 10	VAL ("") returns error code 69, "Illegal argument to VAL".
BASIC-Plus Chapter 10	VAL('2D4') does not return an error code. D is accepted as a specification in VAL but not in assignment or PRINT statements.
BASIC-Plus Chapter 10	TIME$(n), when n>0, gives n minutes before midnight -- n<0 time gives n minutes after midnight.
BASIC-Plus Chapter 10	The maximum length of a string that NUM1$ can return is 255. If you assign the maximum size number (for example, 5E300), you will get the error message "line too long".
BASIC-Plus Chapter 11	Block numbering starts at zero.
BASIC-Plus Chapter 11	On string assignment (LET A$ = B$ storage) a copy of B is made. Given the program LISTNH 400 B$ = "ABC" 410 A$ = B$ 420 LSET B4 = "XYZ" 430 PRINT A$ 440 END the result is the string "ABC".

Manual Chapter	Release Note
BASIC-Plus Chapter 11	The maximum recordsize of an argument is 32256.
BASIC-Plus Chapter 11	CLOSE with a negative channel number will not prevent the writing out of the buffer's last contents to the file.
BASIC-Plus Chapter 11	A PUT operation is not allowed for a file that has been OPENed FOR INPUT.
BASIC-Plus Chapter 13	Note that tokenization is optimized for constants. The statement A = 1E400 does not generate a run-time exception. If you want to raise the exception, write the statement as A = VAL("1E400").
BASIC-Plus Chapter 14	The statement CHAIN requires a full file name; for example, 'PROG2.TEXT'.
BASIC-Plus Chapter 14	You cannot write protect a file unless it already exists on the directory. You must CLOSE it first, and then OPEN and WRITEPROTECT it. You should also explicitly CLOSE any protected files before the end of your program.
BASIC-Plus Chapter 14	Writing to a write-protected file does not result in a run-time error. However, the file is not written to and remains unchanged.
BASIC-Plus Chapter 14	SLEEP(x) sleeps until a character is typed from the keyboard.
BASIC-Plus Appendix D	The following error messages have been added. 2 Nonzero mode values are not supported. 28 Apple period trap. 31 Maximum recordsize of 32256. 90 Error setting safety.

Appendix E
BASIC-Plus Workshop Files

This appendix lists the files on the BASIC-Plus 2.0 Sony micro-diskettes.

File Name	BASIC-Plus Diskette	Notes	Description
BASIC.obj	4	note 1	Workshop program- BASIC-Plus.
ByteDiff.obj	2		Utility program.
Diff.obj	2		Utility program.
DumpPatch.obj	2		Utility program.
EDIT.MENUS.TEXT	2		Editor support file.
Editor.obj	2	note 1	Workshop program- Mouse Editor.
Filediv.obj	2		Utility program.
Filejoin.obj	2		Utility program.
find.obj	2		Utility program.
font.heur	1		Short version for booting.
FONT.HEUR	2	note 3	Data needed to support SYS1Lib.
font.lib	1		Short version for booting.
font.lib	2	note 3	Data needed to support SYS1Lib.
Intrinsic.lib	1		Short version for booting.
Intrinsic.lib	2	note 3	Library directory- intrinsic units.
IOSFPlib.obj	2		Library unit w/interface.
IOSPaslib.obj	1		Short version for booting.
IOSPaslib.obj	2	note 3	Library unit w/interface.
LDSPREFERENCES.OBJ	3		Workshop program.
LDS_RES_PROCS.TEXT	3		Workshop data.
MASTERLIB.OBJ	1	note 2	Install program- library support.
MASTERPHRASE	1	note 2	Install program- alert messages.
Objiolib.obj	4		Library unit (no interface).
OSERRS.ERR	3	note 3	Workshop data- error messages.
PAPER.TEXT	3		Workshop data- Editor stationery.
Portconfig.obj	3		Utility program.

Note 1: These files are software-protected.
Note 2: These files are used for the installation procedure but are not installed.
Note 3: These files are the minimum necessary to run a user program in the Workshop environment. A user program may require other files as well.

File Name	BASIC-Plus Diskette	Notes	Description
Shell.WorkShop	3	note 3	Workshop main program.
Sulib.obj	3	note 3	Library unit w/interface.
Sxref.obj	3		Utility program.
SXREF.OMIT.TEXT	3		Data.
Sysllib.obj	3	note 3	Library units (no interface).
SYSTEM.BT_MICRO	3		System support- 400KB Sony.
SYSTEM.BT_PROF	1	note 3	System support- hard disk.
SYSTEM.BT_TWIG	3		System support- 860KB floppy.
SYSTEM.CD_PRIAM	3		System support- 70MB Priam.
SYSTEM.CD_RS232A	3		System support- RS232 port A.
SYSTEM.CD_RS232B	3		System support- RS232 port B.
SYSTEM.IUDIRECTORY	1		System data (dynamic).
SYSTEM.LLD	1	note 3	System program- low-level drivers.
SYSTEM.LOG	1		System data (dynamic).
SYSTEM.OS	1	note 3	System program- Operating System.
System.Shell	1	note 2	Installation program.
System.Shell	3	note 3	System program- Environment Window.
SYSTEM.UNPACK	1	note 3	System data.
term.menus.text	3		Data for transfer program.
transfer.obj	3		Workshop program- Transfer.
{T11}BUTTONS	4		Data- Preferences.
{T11}MENUS.TEXT	4		Data- Preferences.

Note 1: These files are software-protected.
Note 2: These files are used for the installation procedure but are not installed.
Note 3: These files are the minimum necessary to run a user program in the Workshop environment. A user program may require other files as well.

Appendix B
Workshop Character Set

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	SP	0	@	P	`	p	Ä	ê	†	∞	¿			
1	SOH	DC1	!	1	A	Q	a	q	Å	ë	°	±	¡			
2	STX	DC2	"	2	B	R	b	r	Ç	í	¢	≤	¬			
3	ETX	DC3	#	3	C	S	c	s	É	ì	£	≥	√			
4	EOT	DC4	$	4	D	T	d	t	Ñ	î	§	¥	ƒ			
5	ENQ	NAK	%	5	E	U	e	u	Ö	ï	•	µ	≈			
6	ACK	SYN	&	6	F	V	f	v	Ü	ñ	¶	∂	∆			
7	BEL	ETB	'	7	G	W	g	w	á	ó	ß	∑	«			
8	BS	CAN	(	8	H	X	h	x	à	ò	®	∏	»			
9	HT	EM	)	9	I	Y	i	y	â	ô	©	π	…			
A	LF	SUB	*	:	J	Z	j	z	ä	ö	™	∫	␣			
B	VT	ESC	+	;	K	[	k	{	ã	õ	´	ª				
C	FF	FS	,	<	L	\	l	|	å	ú	¨	º				
D	CR	GS	-	=	M	]	m	}	ç	ù	≠	Ω				
E	SO	RS	.	>	N	^	n	~	é	û	Æ	æ				
F	SI	US	/	?	O	_	o	DEL	è	ü	Ø	ø				

The first 32 characters and DEL are nonprinting control codes.

The shaded area is reserved for future use.

BASIC-Plus User's Guide
for the Lisa™

0370-A

Licensing Requirements for Software Developers

Apple has a low-cost licensing program, which permits developers of software for the Lisa to incorporate Apple-developed libraries and object code files into their products. Both in-house and external distribution require a license. Before distributing any products that incorporate Apple software, please contact Software Licensing at the address below for both licensing and technical information.

©1983 by Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010

Apple, Lisa, and the Apple logo are trademarks of Apple Computer, Inc.

Simultaneously published in the USA and Canada.

Reorder Apple Product #A6D0103 (Complete BASIC-Plus package)
#A6L0112 (Manuals only)

Customer Satisfaction

If you discover physical defects in the manuals distributed with a Lisa product or in the media on which a software product is distributed, Apple will replace the documentation or media at no charge to you during the 90-day period after you purchased the product.

Product Revisions

Unless you have purchased the product update service available through your authorized Lisa dealer, Apple cannot guarantee that you will receive notice of a revision to the software described in this manual, even if you have returned a registration card received with the product. You should check periodically with your authorized Lisa dealer.

Limitation on Warranties and Liability

All implied warranties concerning this manual and media, including implied warranties of merchantability and fitness for a particular purpose, are limited in duration to ninety (90) days from the date of original retail purchase of this product.

Even though Apple has tested the software described in this manual and reviewed its contents, neither Apple nor its software suppliers make any warranty or representation, either express or implied, with respect to this manual or to the software described in this manual, their quality, performance, merchantability, or fitness for any particular purpose. As a result, this software and manual are sold "as is," and you the purchaser are assuming the entire risk as to their quality and performance.

In no event will Apple or its software suppliers be liable for direct, indirect, special, incidental, or consequential damages resulting from any defect in the software or manual, even if they have been advised of the possibility of such damages. In particular, they shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering or reproducing these programs or data.

The warranty and remedies set forth above are exclusive and in lieu of all others, oral or written, express or implied. No Apple dealer, agent or employee is authorized to make any modification, extension or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights that vary from state to state.

License and Copyright

This manual and the software (computer programs) described in it are copy-righted by Apple or by Apple's software suppliers, with all rights reserved, and they are covered by the Lisa Software License Agreement signed by each Lisa owner. Under the copyright laws and the License Agreement, this manual or the programs may not be copied, in whole or in part, without the written consent of Apple, except in the normal use of the software or to make a backup copy. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or loaned to other persons if they agree to be bound by the provisions of the License Agreement. Copying includes translating into another language or format.

You may use the software on any computer owned by you, but extra copies cannot be made for this purpose. For some products, a multiuse license may be purchased to allow the software to be used on more than one computer owned by the purchaser, including a shared-disk system. (Contact your authorized Lisa dealer for more information on multiuse licenses.)

Contents

Chapter 1
Introduction

1.1 Hardware Requirements 1-1
1.2 Diskettes 1-1
1.3 Booting BASIC on the Lisa 1-1

Chapter 2
Language Fundamentals

2.1 Interpreted or Compiled? 2-1
2.2 BASIC Character Set 2-1
2.3 Spaces 2-1
2.4 Keywords and Reserved Words 2-2
2.5 Statements and Lines 2-2
2.6 Elements and Strings 2-3
2.7 Line Numbers 2-4
2.8 Immediate versus Program Execution 2-4
2.9 Entering Comments 2-4
2.10 Identifiers 2-5
2.11 Assignment 2-5

Chapter 3
BASIC Programming Environment

3.1 Using the BASIC Interpreter to Create BASIC Programs 3-1
3.2 Creating BASIC Programs Using the Workshop Editor 3-2
3.3 System Commands 3-2

Chapter 4
Data Types and Data Manipulation

4.1 Integer and Floating-Point Constants 4-1
4.2 String Constants 4-2
4.3 Variables and Variable Names 4-4
4.4 Expressions 4-6
4.5 Arithmetic Operators 4-8
4.6 Logical Operators 4-9
4.7 Relational Operators 4-12
4.8 Precedence of Operators 4-13

Chapter 5
Formatted ASCII Input and Output

5.1 Input and Output Channels 5-1
5.2 Read and Data 5-1
5.3 Restore 5-3
5.4 Input 5-4
5.5 Input Line 5-5
5.6 Print 5-5
5.7 Print Using 5-8

Chapter 6
Branching Statements

6.1 If Then Else 6-1
6.2 If Goto 6-2
6.3 On Goto 6-3
6.4 On Gosub 6-3
6.5 On Error Goto 6-4
6.6 Resume 6-5
6.7 Goto 6-6

Chapter 7
Looping Constructs

7.1 For Next 7-1
7.2 While Next 7-2
7.3 For While 7-4
7.4 For Until 7-5
7.5 Until Next 7-6
7.6 Nested Loops 7-7

Chapter 8
Statement Modifiers

8.1 The If Statement Modifier 8-1
8.2 The For Statement Modifier 8-2
8.3 The While Statement Modifier 8-3
8.4 The Until Statement Modifier 8-3
8.5 The Unless Statement Modifier 8-4
8.6 Multiple Modifiers 8-4

Chapter 9
Matrices

9.1 Dim 9-1
9.2 Mat 9-3
9.3 Mat Read 9-4
9.4 Mat Input 9-5
9.5 Mat Print 9-6
9.6 Matrix Calculations 9-7

Chapter 10
Subroutines and Functions

10.1 Gosub and Return 10-1
10.2 Nesting Subroutines 10-2
10.3 Arithmetic Functions 10-2
10.4 String Functions 10-6
10.5 Matrix Functions 10-11
10.6 Creating Your Own Functions 10-12
10.7 Change 10-14

Chapter 11
Block I/O, Open, and Close

11.1 Open 11-1
11.2 Close 11-3
11.3 Block I/O 11-4

Chapter 12
Virtual Arrays

12.1 Dim Statement for Virtual Arrays 12-1
12.2 Virtual Array Storage 12-2
12.3 Virtual Array Access 12-3
12.4 File Length 12-4

Chapter 13
Advanced Floating-Point Manipulation

13.1 Exceptions 13-1
13.2 Set Exception 13-2
13.3 Ask Exception 13-3
13.4 Set Halt 13-3
13.5 Ask Halt 13-3
13.6 Rounding Modes for Floating-Point Values 13-4
13.7 Set Rounding 13-5
13.8 Ask Rounding 13-5
13.9 Exception Handling and Rounding Examples 13-5

Chapter 14
System Statements

14.1 Wait 14-1
14.2 Sleep 14-1
14.3 Writeprotect 14-1
14.4 Writeallow 14-1
14.5 Unlock 14-2
14.6 Chain 14-2
14.7 Name As 14-3
14.8 Kill 14-3

Appendixes

A Language Summary A-1
B Floating-Point Arithmetic B-1
C Linear Algebra C-1
D Error Messages D-1
E BASIC Workshop Files E-1

Index

Tables

4-1 Ranges for Integer and Floating-Point Constants 4-1
4-2 Arithmetic Operators 4-8
4-3 Logical Operators 4-10
4-4 Truth Tables for Logical Operators 4-11
4-5 Relational Operators 4-12
5-1 Print Statement Punctuation 5-7
5-2 Print Using Statement Formatting Characters 5-9
12-1 Number of Elements in a Virtual Array Block 12-3
B-1 Results of Addition and Subtraction on Infinities B-2
B-2 Results of Multiplication and Division on Infinities B-2

Preface

The Audience of This Manual

This manual is written for experienced BASIC programmers. It describes completely the syntax of the language, and provides occasional examples to clarify points of syntax, but no examples of complete programs. Instructions for starting up and operating BASIC on the Lisa are in the *Workshop User's Guide for the Lisa*. We assume that you have read the *Workshop User's Guide* and the *Lisa Owner's Guide*, and are familiar with your Lisa system. For programming examples, we recommend *Instant BASIC* by Jerome Brown (Dilithium Press, 1975).

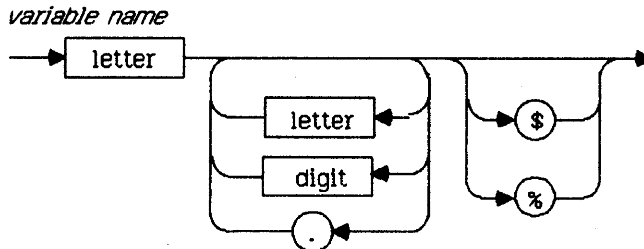
Type and Syntax Conventions

Boldface type is used throughout this manual to distinguish BASIC text from English text. For example, **for i = 1 to 10** is part of a BASIC program.

Italics are used when terms are introduced for the first time.

Lisa BASIC syntax is illustrated by syntax diagrams. The following diagram gives the syntax for constructing legal variable names.

variable name



Start at the upper left corner and follow the letters through the diagram. Various paths are possible. Every path that begins at the left and ends at the arrowhead on the right represents a legal construction of a variable name. Paths through the diagram that do not follow the arrows and curves of the line are not valid. Thus, the rules for constructing valid variable names are:

- A variable name must begin with a letter since the first arrow goes directly to a box containing the name **letter**.
- A variable name may consist of a single letter, since there is a path from this box to the arrowhead on the left that does not go through any more boxes.
- The required letter may be followed by various combinations of letters, digits, and periods (.). Loops within paths indicate repetition is legal.
- The symbol **%** or **$**, instead of a letter, digit, or period, may end a variable name.

Symbols are used in the syntax diagrams to distinguish between different types of input.

- A circle or oval indicates a BASIC keyword or another symbol such as an operator. Enter text in ovals and circles as shown (you can ignore capitalization).
- A word enclosed in a box may be a name for a statement element such as a keyword, or may be a name for some other syntactic construction, such as "variable", which is specified by another diagram. For example, **cost%** could be entered to replace **variable** in the example above.
- Arrows indicate the flow of the diagram.

NOTES

Chapter 1
Introduction

1.1 Hardware Requirements 1-1
1.2 Diskettes 1-1
1.3 Booting BASIC on the Lisa 1-1

Introduction

This manual describes the BASIC language for the Lisa. You should read the *Lisa Owner's Guide* and the *Workshop User's Guide for the Lisa* before using BASIC. We recommend that you have BASIC up and running while you read this manual, so you can try things as you read about them.

1.1 Hardware Requirements

BASIC runs on any standard Lisa system. The BASIC package includes:

- BASIC Interpreter (distributed on two diskettes).
- *BASIC-Plus User's Guide for the Lisa.*
- *Workshop User's Guide for the Lisa.*

1.2 Diskettes

The BASIC Interpreter is distributed on two diskettes. You should make at least one copy of the diskettes and use the copy instead of the master. Then, if anything happens to the copy, you can make new copies from the master. You can make as many copies of your master diskettes as you wish, and you can use the master diskettes on any Lisa system. However, any copy of the master runs only on the system you used to create the first copy. Refer to the *Workshop User's Guide* for information on how to install and copy diskettes.

1.3 Booting BASIC on the Lisa

From the Workshop command line type B (BASIC). A **Ready** prompt is displayed when BASIC is ready to be used.

NOTES

Chapter 2
Language Fundamentals

2.1 Interpreted or Compiled? 2-1

2.2 BASIC Character Set 2-1

2.2.1 Using Uppercase and Lowercase Letters 2-1

2.3 Spaces 2-1

2.4 Keywords and Reserved Words 2-2

2.5 Statements and Lines 2-2

2.6 Elements and Strings 2-3

2.7 Line Numbers 2-4

2.8 Immediate versus Program Execution 2-4

2.9 Entering Comments 2-4

2.10 Identifiers 2-5

2.11 Assignment 2-5

Language Fundamentals

This chapter summarizes the fundamentals of the BASIC language for the Lisa.

2.1 Interpreted or Compiled?

The BASIC that runs on the Lisa is a powerful interpreted BASIC. The Interpreter scans each line as it is input for syntax errors. If the Interpreter finds an error on a particular line, it displays an error message.

2.2 BASIC Character Set

The BASIC character set is comprised of letters, digits, and special characters. These are described in this section.

A *letter* is one of the following:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

A *digit* is one of the following:

0 1 2 3 4 5 6 7 8 9

Special characters include the following:

+ = # $, . < > ( ) \ : & ^ / * - ; " ' % !

2.2.1 Using Uppercase and Lowercase Letters

Uppercase and lowercase are not significant in BASIC commands or identifiers. In the following example, the commands are treated equivalently, but note that case *is* significant for the strings.

```
Ready
PRINT 'CasE sEnsitIve'
CasE sEnsitIve
Ready
A$='Identifiers'
Ready
pRint a$
Identifiers
```

2.3 Spaces

Spaces are significant in BASIC. Spaces may be used in the following ways:

- To separate elements of the BASIC language.
- Within character strings.

At least one space must separate a keyword from the next element in a statement. However, a space is not allowed anywhere within a keyword or identifier. In entering expressions, one or more spaces between an identifier and an operator are optional.

The following are examples of valid and invalid uses of spaces.

print "hi"	Valid.
pr int "hi"	Invalid because spaces are not allowed within keywords.
print "hi"	Valid.
printsin(x)	Invalid because a space is required after a keyword.

Spaces are optional between operators and identifiers in expressions. For example, all of the following are valid.

```
(a+b-d)/d+2               profit/sales-cost
(a + b - d ) / d + 2      profit / sales - cost
```

2.4 Keywords and Reserved Words

Reserved words are words that have a special meaning for the operating system or BASIC. Since they have special meaning, use of reserved words is restricted; they cannot be used, for example, as variable names or as part of variable names. A list of reserved words is provided in Appendix A, Language Summary.

Keywords are Englishlike words that are used in the BASIC language; in other words, keywords are elements of BASIC. Keywords may or may not be reserved. For example, **catalog** is a BASIC command, yet it can be used as a variable name.

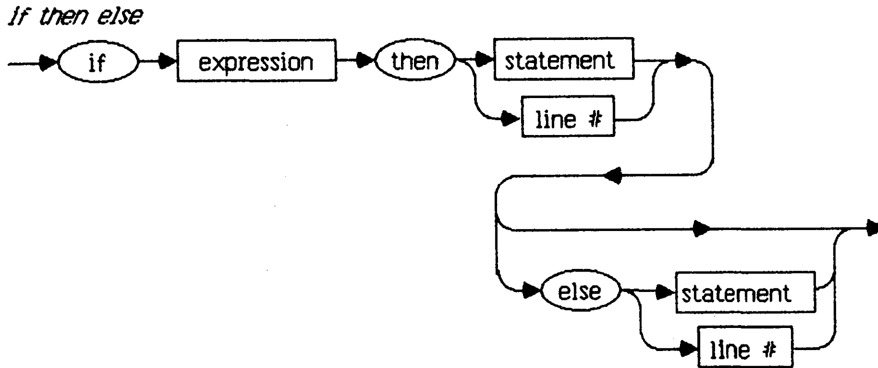
2.5 Statements and Lines

A *statement* is a single BASIC language instruction consisting of one or more keywords, mandatory clauses, and any optional clauses. For example, in the **if then else** statement, **if** is the keyword, **then** starts the mandatory clause, and **else** begins the optional clause.

```
200 if x=y then 250 else 300
```

The following is the syntax for the **if then else** statement:

If then else



A *program line* is comprised of one or more statements, ending with <RETURN>. When more than one statement exists on a program line, you must separate the statements with a backslash (\ ) or a colon (:), as shown below.

```
10 if x=y goto 20\if x=z then 25\goto 15
10 if x=y goto 20:if x=z then 25:goto 15
```

Most BASIC statements can appear anywhere within a multiple statement line; restrictions are given in the statement descriptions.

A program line, whether it contains one or more statements, can be longer than a screen line if an ampersand (&) and a <RETURN> are at the end of each screen line.

2.6 Elements and Strings

An *element* is any sequence of characters separated from other such sequences in a statement by one or more spaces. It is illegal to split an element between screen lines.

A *string* is a series of characters bounded by quotation marks. For example, the following is a legal statement:

```
20 if number1 < maximum then & <RETURN>
   print 'within range' & <RETURN>
   else & <RETURN>
   print 'overflow'
```

Note that strings are counted as elements when extending program lines so that it is illegal to split strings between screen lines.

The following is *not* a legal statement, because the string **"within range"** is split between two lines:

```
20 if number1 < maximum then & <RETURN>
   print 'within & <RETURN>
   range' else print 'overflow'
```

The & symbol at the end of the second screen line is interpreted as part of the string **'within &'**; it is not recognized as the continuation character. The program line is terminated, and the next line, **range' else print 'overflow'** causes a syntax error.

2.7 Line Numbers

A *line number* identifies the program line and defines the order of execution. The legal values for a line number are the positive integers from 1 to 32767 inclusive. BASIC stores and executes program lines in ascending order by line number, which is not necessarily the order you enter them.

Only one program line can have a given line number. If you enter a line with a duplicate line number, it erases the old line with that line number.

2.8 Immediate versus Program Execution

BASIC supports two modes: *program mode* and *immediate mode*. When in program mode, program lines are stored for later execution; when in immediate mode, instructions are executed as they are recognized by the interpreter. The presence or absence of a line number distinguishes between program and immediate modes. Program lines are stored when preceded by a line number. The following are examples of stored program lines:

```
10 rem  this is a stored program line
20 for number=1 to 10
```

Immediate mode allows you to execute a single line. Statements used in immediate mode can refer to established variables in the current program or to variables defined in previous immediate mode statements. Variables entered in immediate mode retain their values until the workspace is cleared. Refer to Chapter 3, BASIC Programming Environment, for an explanation of the workspace. The following are examples of immediate mode statements:

```
a=2
Let b=6
print a+b
```

2.9 Entering Comments

There are two ways to add comments to a program: the **rem** statement and the exclamation point (!). The **rem** statement is used when you wish only a comment to appear on a particular program line. The exclamation point is used to insert comments at the end of a program line.

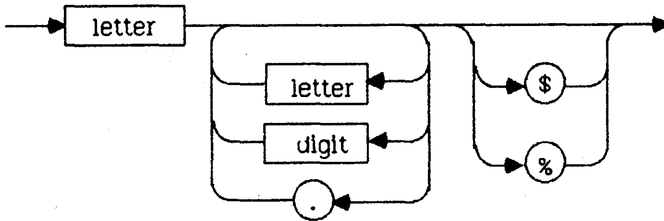
BASIC does not execute any program line that begins with the keyword **rem**. **Rem** statements must be numbered. If a comment requires more than one line, you can break it up into several **rem** statements, or you can use **&** **<RETURN>** and continue the comment on the next screen line.

Use an exclamation point when you want to add a comment after a BASIC program statement. You don't have to enter a colon before the exclamation point. The system ignores any text on that line following the exclamation point. You can also begin a line with an exclamation point instead of the **rem** keyword, and the system will not execute that line.

2.10 Identifiers

Identifiers identify user-defined functions, variable names, and array names. The following is the syntax for constructing identifier names.

identifier name



Identifier names should not exceed 30 characters. (Identifiers longer than 30 characters will generate an "Identifier too long" error and will be truncated to 30 characters.)

Identifiers are associated with functions, variables, and arrays. All identifiers that start with **fn** are function names. For example, the following are valid identifiers that are the names of functions:

fnaddem

fnscratch

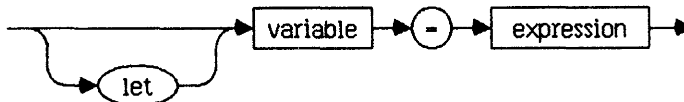
fncalc.int

For more information on variable names, refer to Chapter 4, Data Types and Data Manipulation.

2.11 Assignment

One of the most common operations in a program is the assignment of a value to a variable.The following is the syntax for the assignment statement.

assignment



The following examples are valid and assign values to variables.

```
10 let a%=10

20 let c$="current payment"

20 B=12.5

amount=123.98
```

Note that the keyword **let** is optional in the assignment statement, but the equal sign (=) is always required.

NOTES

Chapter 3
BASIC Programming Environment

3.1 Using the BASIC Interpreter to Create BASIC Programs 3-1

3.2 Creating BASIC Programs Using the Workshop Editor 3-2

3.3 System Commands 3-2

3.3.1 Controlling BASIC Programs in Program Space 3-3
3.3.1.1 Delete 3-4
3.3.1.2 New 3-4
3.3.1.3 Old 3-5
3.3.1.4 Replace 3-5
3.3.1.5 Save 3-6
3.3.1.6 Unsave 3-6
3.3.1.7 Append 3-7
3.3.1.8 Renumber 3-8
3.3.2 Informational Commands 3-8
3.3.2.1 List and Listnh 3-8
3.3.2.2 Length 3-9
3.3.2.3 Catalog or Cat 3-10
3.3.3 Controlling Execution of BASIC Programs 3-10
3.3.3.1 Run and Runnh 3-10
3.3.3.2 Cont 3-11
3.3.3.3 -Period 3-11
3.3.4 Debugging Commands 3-12
3.3.4.1 Trace 3-12
3.3.4.2 Variables 3-13
3.3.5 Leaving BASIC 3-13

BASIC Programming Environment

This chapter explains the BASIC system commands, touches on the Workshop programming environment, and points out those Workshop features most useful for BASIC programmers. The two ways of creating programs are discussed, and a discussion of the system commands follows. The BASIC system commands control the programming environment and are not a part of the BASIC language.

3.1 Using the BASIC Interpreter to Create BASIC Programs

You can create BASIC programs within the BASIC Interpreter, or within the Workshop editor. When you create a program within the Interpreter, each program line is scanned for syntax errors when <RETURN> is entered. If there is a syntax error in the line, an error message is displayed. For example, the Interpreter displays an error message when the following program line is entered.

```
220 far i=1 to 10

220 far i<<<<<

****** Unrecognizable statement or command
```

The BASIC editing capabilities consist of a destructive backspace and the **delete** command. You can use the destructive backspace on a particular line until you enter <RETURN>. After you enter <RETURN>, the only way to change the contents of a particular line is to retype the entire line using the same line number. For example, to correct the typographical error below, the entire line must be retyped.

```
220 print "The number of hours fothis pay period =  ";hours
                                -----------------
220 print "The number of hours for this pay period = ";hours
```

BASIC replaces the original line with the corrected version.

Lines and groups of lines can be removed using the **delete** command. Refer to Section 3.3.1.1, Delete, for an explanation of this command.

When you invoke BASIC, a *program space* is created. The BASIC Interpreter allows one program at a time to occupy program space. All changes and additions you make to programs in the Interpreter affect the program in program space. When you first invoke BASIC, and before you load any programs into this program space, it is empty and new programs can be input. When you wish to work on a different program, you must clear the program space, using the **old** or **new** commands.

You use these system commands to work with program space:

save	Saves the program file.
replace	Saves the new version of an existing file.
new	Clears the program space.
old	Clears the program space and loads the specified program.

These commands are described in Section 3.3.1, Controlling BASIC Programs in Program Space.

Whenever you start BASIC, the following occurs:

1. All the input and output channels are closed.
2. All **input** statements wait indefinitely for input.
3. All floating-point exceptions and halts are turned off. Refer to Chapter 13, Advanced Floating-Point Manipulation, for more information.

3.2 Creating BASIC Programs Using the Workshop Editor

We recommend that you use the Workshop editor to create and edit your BASIC programs. It provides more editing capabilities than the BASIC Interpreter. The fundamental editing operations are inserting characters, cutting a portion of the text, and pasting text to a new location. You can use the mouse to scroll the text in the window, move the insertion point, and select text to be cut or copied. For a detailed explanation, refer to the *Workshop User's Guide for the Lisa*.

When you create programs in the Workshop editor, you can't check the syntax as you type. However, the Interpreter checks the syntax of programs as they are loaded into program space (refer to Section 3.3.1.3, Old). Programs created using the Workshop editor can be run in BASIC using the **run** or **runnh** commands (refer to Section 3.3.3.1, Run and Runnh).

3.3 System Commands

System commands control the BASIC programming environment and are never part of a stored BASIC program. Rather, the system commands are used at command level (without line numbers). If you attempt to place a system command within a program, the following error message is displayed:

********Unrecognizable statement or command**

The BASIC system commands may be put into four functional groups:

- Commands that control the program in program space.
- Commands that provide information about programs.
- Commands that control program execution.
- Debugging commands.

The system commands that control the program in program space are:

save	Saves the program file.
unsave	Removes the program file from the specified or prefixed volume.
replace	Saves the new version of a program file.
new	Clears the program space.
old	Clears the program space and loads the specified program.
append	Consolidates two files.
delete	Deletes one or more lines.
renumber	Renumbers all lines in the resident program.

The informational commands provide information about the current program and about saved files. These are:

list	Lists all or part of the current program, including a header.
listnh	Lists all or part of the current program, excluding the header.
length	Prints the amount of memory occupied by the current program.
catalog	Prints a list of all .text files on the specified volume or the prefixed volume.
cat	Same as **catalog**.

The commands that control program execution are:

run	Prints a header and starts program execution.
runnh	Starts program execution, with no header.
cont	Restarts program execution (short for continue).
-period	Halts program execution.

The debugging commands are:

trace	Toggles between trace and nontrace modes.
variables	Lists all variables, their types and values.

3.3.1 Controlling BASIC Programs in Program Space

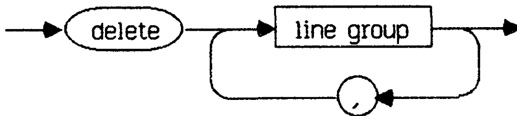
These system commands are used to control BASIC programs in program space. As was mentioned earlier, only one program may reside in program space at a time. The commands discussed in this section all control the current program and are used when BASIC programs are created in the Interpreter.

Note: Some computational run-time errors are not reported unless the set statement is used to require reporting. Refer to Chapter 13, Advanced Floating- Point Manipulation, for more information.

3.3.1.1 Delete

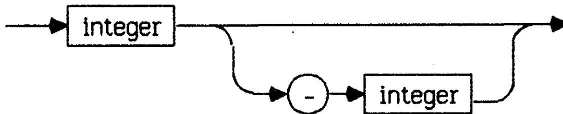
The **delete** command removes one or more lines from the current program. The syntax for **delete** is:

delete



where the syntax for **line group** is:

line group



Line group indicates which line, lines, and groups of lines are to be deleted. You may select any combination of lines and groups of lines in a single **delete** command if you separate each element as indicated in the **line group** syntax diagram.

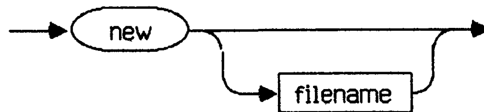
Examples:

delete 210	Deletes line 210.
delete 225-335	Deletes lines 225 to 335 inclusive.
delete 110,225-335,445	Deletes lines 110, 225 to 335 inclusive, and line 445.

3.3.1.2 New

The **new** command clears the program space in memory. Only one program, the current program, can reside in program space. The syntax is as follows:

new



The program space is cleared; you can begin entering a new program when the **Ready** prompt appears. If a **filename** is entered, the Interpreter uses that name for the new program. If a **filename** is not entered, the system prompts as follows:

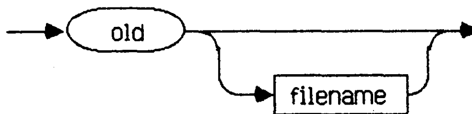
```
Filename: [.TEXT]
```

If a filename is now entered, it is used; otherwise NONAME is used. If a volume name is not included, **new** assumes the prefixed volume. The default file name extension is **.text**.

3.3.1.3 Old

The **old** command clears the program space in memory and loads the specified program. The syntax is as follows:

old



If a **filename** is not entered, the system prompts as follows:

```
Filename: [.TEXT]
```

If a volume name is not included, **old** assumes the prefixed volume. The default file name extension is **.text**.

Examples:

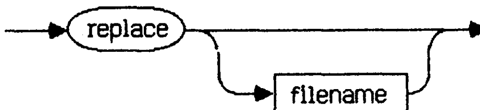
old

Filename: [.TEXT] -upper-x	The system prompts for the filename since it was not entered at the command level. You enter the volume name **upper** and the filename x. The extension **.text** is entered automatically.
old y	The **y.text** program is read into program space from the prefixed volume.

3.3.1.4 Replace

The **replace** command writes the current program to a specified file. The syntax is as follows:

replace



If a volume name is not included, **replace** assumes the prefixed volume. The default file name extension is .text; you need not enter it.

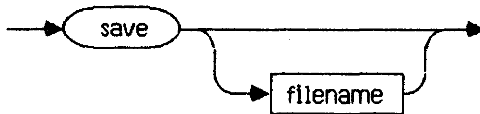
Example:

replace config	The program is written to **config.text**.

3.3.15 Save

The **save** command writes the current program to a volume. The .text extension, if not present, is added automatically. The syntax is as follows:

save



If a volume name is not included, **save** assumes the prefixed volume.

The **save** command assumes that no file already exists with the specified file name. When you want to replace an existing file, use **replace**. If you use **save** when a file with the specified filename already exists, the Interpreter prompts:

```
delete old filename?
```

If you respond y for yes (and <RETURN>), the new version of your file will be written to disk.

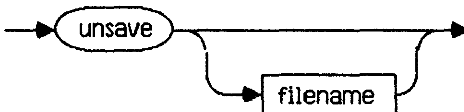
Example:

save progress	The program is written to **progress.text**.

3.3.16 Unsave

The **unsave** command removes the specifed filename from a volume. The syntax is as follows:

unsave



If you do not enter a **filename**, the system prompts for it as follows:

```
Filename: [.TEXT]
```

If you do not include a volume name, **unsave** assumes the prefixed volume.

The default file name extension is .text; you need not enter it.

Examples:

unsave config	Removes **config.text** from the prefixed volume.
unsave	
Filename: [.TEXT] config	Removes **config.text** from the prefixed volume.

3.3.1.7 Append

The **append** command consolidates two files by writing the contents of the specified file over the current file in the workspace. The syntax is:

append

→ (append) → [filename] →

where **filename** is the name of the program to be appended to the current program.

If you do not enter a **filename**, the system prompts for it as follows:

```
Filename: [.TEXT]
```

If you do not include a volume name, **append** assumes the prefixed volume. The default file name extension is .text; you need not enter it.

Append adds all unique lines to the current program. When duplicate line numbers are encountered, the line from the saved program overwrites the line in the current program.

Examples:

Assume the following is a saved program:

```
Program OLD
10 for i=1 to 3
20 print "write over line 20 in the current program?"
30 next i
40 end
```

Assume this is the current program:

```
05 for n=1 to 3
20 print "this is line 20 in the current program"
35 next n
40 end
```

To append **old.text** to the current program, you type **append old**

The following is the result:

```
05 for n=1 to 3
10 for i=1 to 3
20 print "write over line 20 in the current program?"
30 next i
35 next n
40 end
```

3.3.1.8 Renumber

The **renumber** command renumbers the program lines in the current program, from the specified starting line to the end of the program, with the specified increment. The following is the syntax:

renumber

When you type **renumber**, the system responds:

renumber starting at ?

Type in the first line to be renumbered. The system then asks:

increment ?

Type in the increment you want.

3.3.2 Informational Commands

These system commands provide information about the current program.

3.3.2.1 List and Listnh

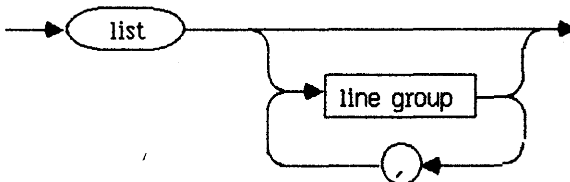
The **list** and **listnh** commands display all or part of the current program. The **list** command prints a header of the form:

Program filename

The **listnh** command (list no header) does not print a header.

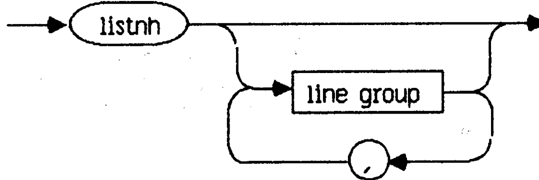
The syntax for **list** is:

list



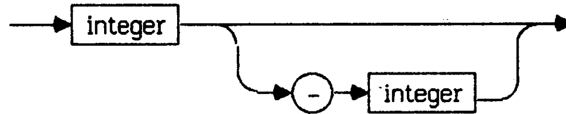
The syntax for **listnh** is:

listnh



The syntax for **line group** is the same for both **list** and **listnh**:

line group



Line group indicates which line, lines, and groups of lines are listed. You can request any combination of lines and groups of lines in a single **list** or **listnh** command if you separate each element with a comma as indicated in the **line group** syntax diagram.

Examples:

list 220	Prints a header and then lists line 220.
listnh 220	Lists line 220 (without a header).
list 220-335	Prints a header, lists lines 220 to 335 inclusive.
listnh 220-335	Lists lines 220 to 335 inclusive (without a header).
list 10,20,35-75,80-95	Prints a header, lists lines 10, 20, 35 to 75, inclusive, and 80 to 95, inclusive.
listnh 15,25-50,55,80-95	Lists lines 15, 25 to 50 inclusive, 55, and 80 to 95 inclusive (without a header).

3.3.2.2 Length

The **length** command prints the amount of memory occupied by the current program, and the maximum program size. The following is the syntax.

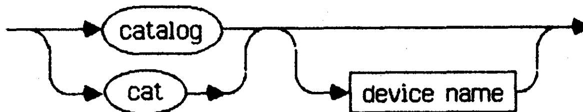
length

The **length** command returns x(y)K of memory used where y is the total amount of space you can ever have, x is how much space is used, and K stands for kilobytes. A kilobyte is 1024 bytes.

3.3.2.3 Catalog or Cat

The **catalog** (or **cat**) command prints a list of all .text files on the specified volume (or the prefixed volume, if it is different; refer to the *Workshop User's Guide for the Lisa* for an explanation of **prefixed** volumes). The following is the syntax for the **catalog** command.

catalog



Catalog prints a list of the following form:

```
catalog for -paraport
ldswdoc.text
documents.text
jrnlfiles.text
```

3.3.3 Controlling the Execution of BASIC Programs

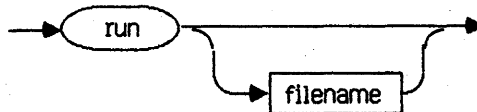
3.3.3.1 Run and Runnh

The **run** and **runnh** commands start program execution. Program execution always begins at the program line with the lowest number. The **run** command prints a header of the form:

```
Program filename
```

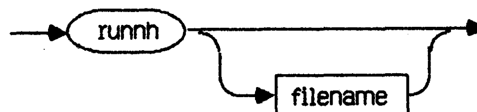
before starting program execution. The **runnh** command (run no header) omits the header. The syntax for **run** is as follows:

run



If a volume name is not specified, the prefixed volume is assumed. The syntax for **runnh** is as follows:

runnh



If a volume name is not specified, the prefixed volume is assumed.

If a file name is not specified, **run** and **runnh** execute the current program. If there is no program currently in memory and the command doesn't include a filename, the system displays the error message:

???Missing END statement in line -1.

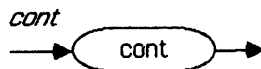
Examples:

run	Prints a header and starts execution of the current program.
runnh	Starts execution of current program (without a header).
run taxes	Loads **taxes.text** into memory, prints a header, and starts execution.
runnh taxes	Loads **taxes.text** into memory and starts execution (without a header).

3.3.3.2 Cont

The **cont** command restarts execution of programs halted by the **stop** command or by the -period interrupt. When a **cont** command is input, program execution is resumed at the statement immediately following **stop** (even if the **stop** statement is in the middle of a multiple statement line). The **cont** command is never part of a stored BASIC program. The syntax is as follows.

cont



Examples:

110 next n:stop:n=0	When a **cont** is input, program execution resumes with the statement **n=0**.
220 stop	
225 gosub 1101	When a **cont** is encountered, program execution resumes at the line after line 225.

3.3.3.3 -Period

-period is the soft interrupt character; it interrupts program execution at "safe" places, so that no data is lost. To use the -period interrupt, hold down the Apple key () while you type a period (.). When you type -period, BASIC will interrupt the program at the next safe place it encounters. The -period interrupt will *not* get you out of a request for input.

Note: Although you type two keys to use the -period interrupt, it is treated as only one character.

3.3.4 Debugging Commands

The debugging commands are used when a program does not work the way you want it to.

3.3.4.1 Trace

Trace is a debugging command that switches between trace and non-trace modes. When the system is in trace mode, program execution is followed and line numbers are printed as they are executed. The syntax is as follows.

trace

When trace mode is entered, the system prints

```
Trace flag set to TRUE
```

and when trace mode is left, it prints

```
Trace flag set to FALSE
```

Examples:

For the following very simple program:

```
20 for i=1 to 3
30 print "hello"
40 next i
50 end
```

The following is printed on the screen when the trace flag is set to true and the program is executed:

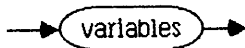
```
Line -1
Line 20
Line 30
hello
Line 40
Line 20
Line 30
hello
Line 40
Line 20
Line 30
hello
Line 40
Line 50
```

3.3.4.2 Variables

The **variables** command searches the current program for variables, printing a list by type and value. BASIC supports six types of variables: integer, floating point, string, floating-point arrays, string arrays, and integer arrays.

The syntax is:

variables

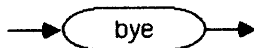


The **variables** command applies to the current program only.

3.3.5 Leaving BASIC

The **bye** command closes and saves open files and exits to the Workshop command line. The syntax is as follows.

bye



NOTES

Chapter 4
Data Types and Data Manipulation

4.1 Integer and Floating-Point Constants 4-1
4.1.1 Numeric Notation 4-1
4.1.2 Integer and Floating-Point Arithmetic 4-2
4.2 String Constants 4-2
4.3 Variables and Variable Names 4-4
4.4 Expressions 4-6
4.5 Arithmetic Operators 4-8
4.5.1 Results of Division by Zero 4-8
4.6 Logical Operators 4-9
4.7 Relational Operators 4-12
4.8 Precedence of Operators 4-13

Data Types and Data Manipulation

BASIC supports integer, floating-point, and string values. This chapter defines each of these data types and presents the legal BASIC data manipulation operations.

4.1 Integer and Floating-Point Constants

In BASIC an integer constant is specified as a series of digits ending with a percent sign (%). A floating-point constant is specified as a series of digits with an optional decimal point to separate the whole and fractional parts of the number. In other words, the absence of a percent sign makes a numeric value a floating-point value. A minus sign (-) preceding the first digit indicates negative integer and floating-point values. For example, the following are legal integer constants:

```
1%        -245%        8970%
```

and the following are legal floating-point constants:

```
1.275     -354.786     4
```

Table 4-1 lists the valid ranges for integer and floating-point constants.

Table 4-1

Ranges for Integer and Floating-Point Constants

Type of Constant	Range
Integer	-32768 through 32767
Floating-Point	±4.9*10^-324 through ±1.7*10^308

4.1.1 Numeric Notation

You can enter numeric constants in one of two ways. You can enter the value as a string of digits such as

```
105000
```

or you can use scientific notation and write the value as

```
105E+3
```

4.1.2 Integer and Floating-Point Arithmetic

Floating-point values occupy four 16-bit words of storage and use double precision arithmetic. Using double precision arithmetic you can represent values up to 15 decimal places accurately.

Integers occupy one 16-bit word of storage. The range of integer values supported by BASIC is continuous; the number following +32767 is -32768. Therefore when you add large positive integer values, negative numbers can result. For example, the following additions yield negative values:

```
print 32499% + 31223%

-1814

print 25678% + 31568%

-8290
```

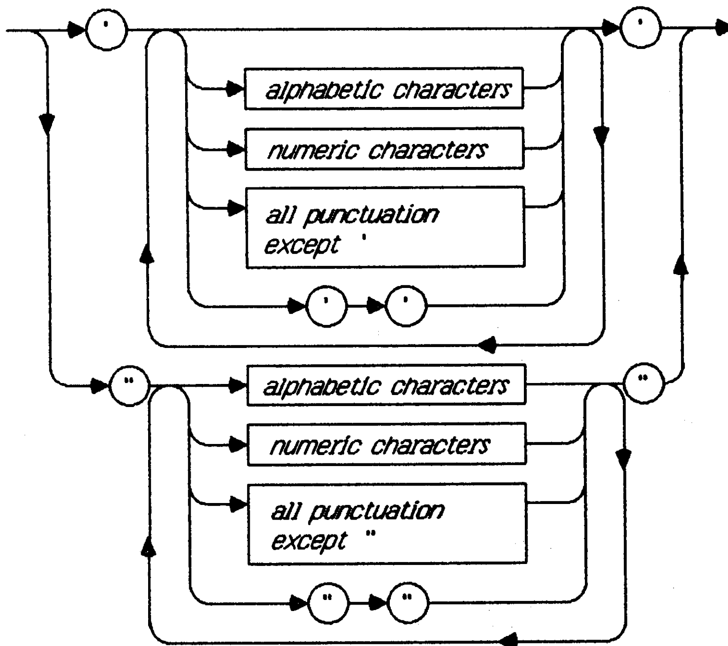
The values of integer variables or expressions can be used as logical variables. 0% corresponds to the logical value FALSE; any nonzero integer value corresponds to the logical value TRUE.

4.2 String Constants

A string constant is data made up of a series of characters, digits, and special characters. The value of a string constant does not change during program execution. A string constant can contain up to 32767 characters. However, to create a string longer than a screen line, you must use the string functions. Refer to Section 10.4, String Functions, for more information.

The following is the syntax for string constants:

string constant



When you enter a string constant, start and end the string with single or double quotation marks; the quotation marks distinguish a string constant from a string variable name. The quotation marks aren't part of the string and aren't included when you output the string. To print the phrase **Please enter your name**, you could type the following:

```
print 'Please enter your name'
```

The system displays

```
Please enter your name
```

in response. Note that the results would be the same if you began and ended the string with double quotation marks.

When you want a quotation mark to appear within a string you can:

1. Use the other type of quotation mark (e.g. the type of quotation mark that doesn't appear in the string) as the string delimiter. For example, if you type:

   ```
   print 'Use the "other" quotation mark'
   ```

   the system displays:

   ```
   Use the "other" quotation mark
   ```

   or if you type:

   ```
   print "use the 'other' quotation mark"
   ```

   the system displays:

   ```
   Use the 'other' quotation mark
   ```

2. Enter two quotation marks of the same type where you want a single quotation mark. This distinguishes a delimiter from part of the string. For example, if you type either of the **print** statements,

   ```
   print "Can't figure profits without more information."
   print 'Can''t figure profits without more information.'
   ```

   the system displays the line:

   ```
   Can't figure profits without more information.
   ```

   Typing:

   ```
   print "Use ""double"" quotation marks"
   ```

   prints:

   ```
   Use "double" quotation marks
   ```

   and typing:

   ```
   print 'Use ''double'' quotation marks'
   ```

   prints:

   ```
   Use 'double' quotation marks
   ```

4.3 Variables and Variable Names

A *variable* is a value that can change during program execution. The *variable name*, which does not change, is associated with the data. Variables can assume values of any of the three data types. The variable name determines the type of value that it represents.

The following is the syntax for creating variable names.

variable name

letter
letter
digit
.
$
%

***Note:* Variable names may not exceed thirty characters.**

A name ending in a percent sign (**%**) represents an integer value, a name ending with a dollar sign (**$**) represents a string value, and a name without either of the two distinguishing symbols represents a floating-point value.

The following are examples of legal variable names for each data type.

1. Integer

   loops%

   emp.num%

   crates%

2. Floating-point

   finaltotal

   totalcash

   bottom.line

3. String

   brand.name$

   supply.type$

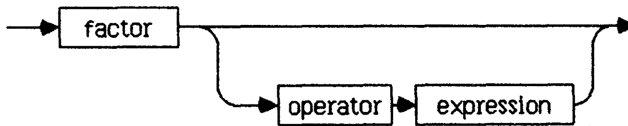
   personnel$

The system allocates space for a variable in BASIC the first time you use it in a statement. BASIC initializes numeric variables to zero and initializes string variables to the zero length string.

4.4 Expressions

An expression is a group of values (constants, variables, and functions) and operators that is used to compute a new value. The following is the syntax for expressions.

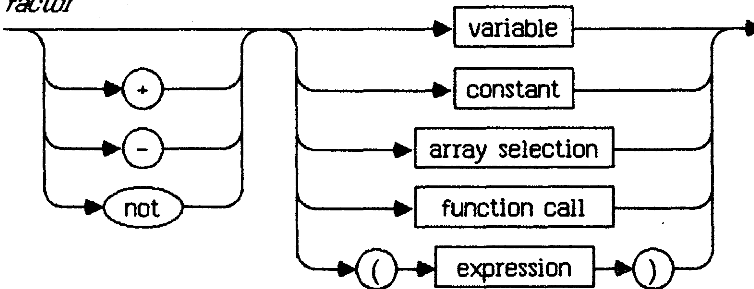
expression



Note: All implicit rounding of floating-point expressions to integers is to the nearest integer; values halfway between two integers are rounded to the even integer.

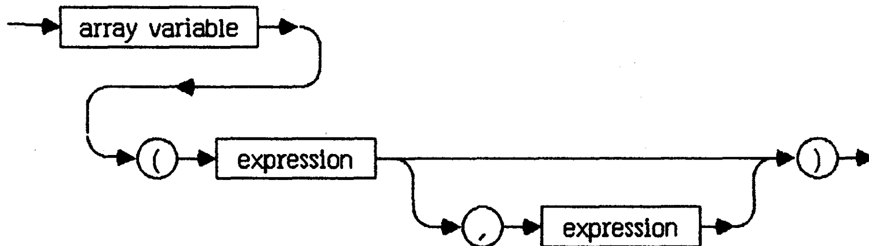
The following is the syntax for factor.

factor



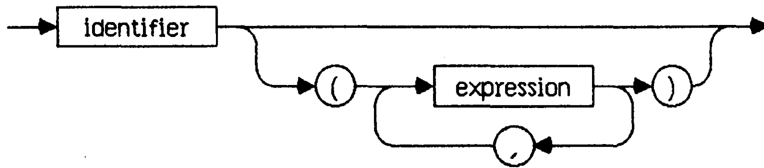
The syntax for **array selection** is as follows.

array selection



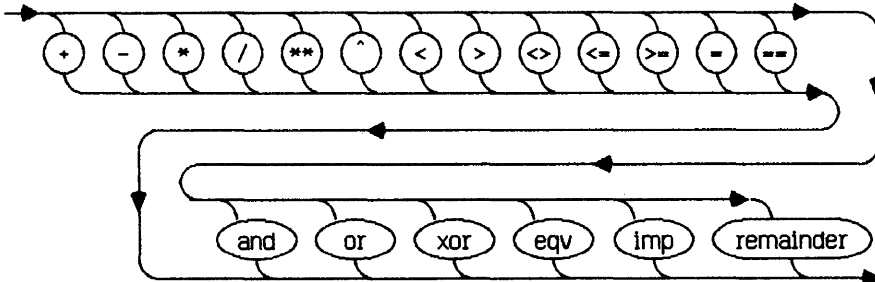
The syntax for **function call** is as follows.

function call



The syntax for **operator** is as follows.

operator



The type of operator used in expressions is dependent on the element type. If the elements that the operator separates are string constants or functions that return string values, the operator must be a relational operator or the string concatenation operator (+). (Refer to Section 4.7, Relational Operators, for an explanation and examples.) If the elements in the expression are numeric, the operator can be mathematical, logical, or relational. (Refer to Section 4.5, Arithmetic Operators; Section 4.6, Logical Operators; and Section 4.7, Relational Operators, for an explanation and examples.)

When evaluating an expression, the system always checks the data type of the result against the data type of the target variable. If the result of the expression is a string and the target variable is numeric, or vice versa, the

system returns an error message. If the result and the variable are both numeric but of different types, one of the following occurs:

- If the target variable is an integer and one or more values in the expression are floating-point values, the system evaluates the expression using the floating-point values, rounds any fractional portion of the result, and assigns that value to the integer variable.
- If the target variable is a floating-point value and one or more of the values in the expression are integer values, the system treats the integers as floating-point values.

4.5 Arithmetic Operators

BASIC recognizes the arithmetic operators defined in Table 4-2.

Table 4-2
Arithmetic Operators

Operator	Use	Explanation
+	X + Y	Adds X to Y
-	X - Y	Subtracts Y from X
*	X * Y	Multiplies X by Y
/	X/Y	Divides X by Y
^	X^Y	Raises X to the Y power
**	X**Y	Raises X to the Y power
remainder	X **remainder** Y	Computes **remainder**

The + and - signs can be used as unary operators. The + is ignored; the - changes the sign of the value which follows.

4.5.1 Results of Division by Zero

The results of a division by zero depend on the type of operand, that is, whether the value is floating-point or integer. If the division is between floating-point values, the result is usually positive or negative infinity.

For example:

5.5/0.0

results in positive infinity, while

-5.5/0.0

results in negative infinity. However,

0.0/0.0

results in a NaN ("Not a Number"). Refer to Appendix B, Floating-Point Arithmetic, for more information.

Division by zero when the values are integers results in a run-time error.

4.6 Logical Operators

A logical operator can separate two integer variables or constants, or two relational or arithmetic expressions. Floating point variables and constants are legal within a logical expression only as part of a relational or arithmetic expression.

When integer values are used, the value 0% is equivalent to false. All other values are true.

BASIC recognizes the logical operators listed in Table 4-3.

Table 4-3
Logical Operators

Operator	Rules of Evaluation
and	*X and Y* is true only if *X* and *Y* are both true.
or	*X or Y* is true when either *X* or *Y* is true. The expression is false only when both *X* and *Y* are false.
eqv	*X eqv Y* is true if *X* and *Y* are both true or both false.
not	If *X* is true, *not X* is false, and if *X* is false, *not X* is true.
imp	*X imp Y* is true unless *X* is true and *Y* is false.
xor	*X xor Y* is false when both *X* and *Y* are false, or when both *X* and *Y* are true. The expression is true when one value is false and the other is true.

The result of a logical operation is either true or false. BASIC considers 0% to be false and any other value to be true. The truth tables in Table 4-4 define the result of a logical expression for each possible pair of values.

Table 4-4

Truth Tables for Logical Operators

x y	x or y	*x y*	x xor y	*x y*	x eqv y
t t	true	t t	false	t t	true
t f	true	t f	true	t f	false
f t	true	f t	true	f t	false
f f	false	f f	false	f f	true

x	not x	*x y*	x and y	*x y*	x imp y
t	false	t t	true	t t	true
f	true	t f	false	t f	false
		f f	false	f t	true
		f t	false	f f	true

4.7 Relational Operators

Relational operators compare two numeric or string expressions that are composed of constants or variables, or both. The result of the comparison is either true or false. Table 4-5 lists the BASIC relational operators and the comparisons they perform.

Table 4-5
Relational Operators

Operator	Example	Explanation
=	X = Y	Determines whether the value of X is equal to the value of Y.
<	X < Y	Determines whether the value of X is less than the value of Y.
>	X > Y	Determines whether the value of X is greater than the value of Y.
<=	X <= Y	Determines whether the value of X is less than or equal to the value of Y.
>=	X >= Y	Determines whether the value of X is greater than or equal to the value of Y.
<>	X <> Y	Determines whether the value of X is not equal to the value of Y.
==	X == Y	Determines whether **print** X and **print** Y would agree.

The system compares strings with the ASCII sequence. When the two strings are of different length, the system compares the characters of the shorter string to the corresponding characters in the long string. If the system finds no differences between the two strings and the remaining characters of the longer string are blanks, the two strings are equal. If the remaining characters are not blanks, the longer string is greater than the shorter string.

For instance, suppose **String1$** contains 'Graphs', **String2$** contains 'Graphs ', and **String3$** contains 'Graphs and Charts'. The system considers **String1$** and **String2$** equivalent, although **String2$** has more characters than **String1$**, because the additional characters in **String2$** are blanks. However, the system treats **String3$** as greater than **String1$** because the characters in **String3$** following the common characters are not all blanks.

4.8 Precedence of Operators

When a calculation involves more than one operator, BASIC performs the operations in the following order:

1. ^ , ** (exponentiation)
2. **Unary+, unary-, not**
3. *, /, **remainder** (multiplication, division)
4. **Binary+ , binary-** (addition, subraction)
5. <, >, <=, >=, <>, ==
6. **and**
7. **or, xor, eqv, imp**

Within each level of hierarchy, operations are performed from left to right. However, parentheses change the order of evaluation. BASIC evaluates the expression within the innermost set of parentheses first, then the expression within the next higher set of parentheses, and so on. Within parentheses, BASIC follows the rules given above.

For example,

-2^2=-4

(-2)^2=4

If A=2, B=4, and C=5, then

A+B*C=22

(A+B)*C=30

When evaluating the expression, the system multiplies B by C and then adds A to the result, because multiplication is done before addition. But if you enclose the addition in parentheses, the system adds A and B first and then multiplies the result by C, because operations within parentheses are done before any others.

NOTES

Chapter 5
Formatted ASCII Input and Output

5.1 Input and Output Channels 5-1
5.2 Read and Data 5-1
5.3 Restore 5-3
5.4 Input 5-4
5.5 Input Line 5-5
5.6 Print 5-5
5.7 Print Using 5-8

Formatted ASCII Input and Output

Formatted ASCII input and output reads and writes characters in ASCII format to and from files and devices in the system. ASCII format is the format used for the keyboard and screen. ASCII input and output, although simple and flexible, require the system to convert values from internal forms to ASCII format and do not allow random file access during output.

5.1 Input and Output Channels

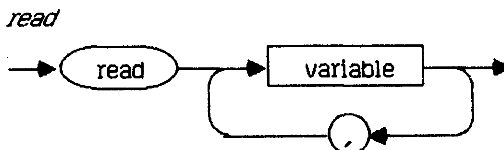
BASIC communicates with files through *channels*. The **open** statement assigns a logical input/output channel to a filename. Refer to Section 11.1, Open, for more information.

BASIC supports thirteen logical input and output channels. These channels are numbered 0-12. Channel 0 is always associated with the console. When you print from BASIC without specifying the channel number, channel 0 is assumed. Channels 1-12 are not associated with a file or device when you first start up the system. You associate the channels, as you need them, by using the **open** statement. The association imposed by the **open** statement lasts until you either close the channel with the **close** statement, clear the workspace using the **new** or **old** commands, or exit BASIC. Refer to Section 11.2, Close, for information about the **close** statement, and to Chapter 3, BASIC Programming Environment, for information about the **new** and **old** commands.

5.2 Read and Data

Data can be defined within a program with the **data** statement. Data defined within a program can be read by the **read** statement. Data are defined in the **data** statement in an ordered list. Likewise, variables are defined in the **read** statement in an ordered list. When Lisa BASIC executes a **read** statement, it assigns values from a **data** statement to each of the variables listed in the **read** statement. The following is the syntax for the **read** statement.

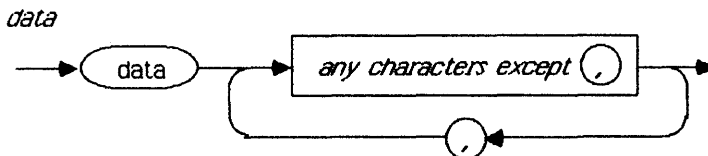
read



Variable is any valid variable name.

The following is the syntax for the **data** statement.

data



If a **data** statement is included in a multiple-statement line, it must be the last statement in the line.

BASIC maintains a list of values that it builds from all the **data** statements in a program. The first value in the list is the first value in the first **data** statement in the program; the last value in the list is the last value in the last **data** statement in the program.

When the system executes the first **read** statement in a program, it assigns the first value in the data list to the first variable in the **read** statement, the second value in the data list to the second variable in the **read** statement, and so on. For example,

```
20  read first$, middle$, last$
          .
          .
          .
90  data John, Henry, Madison
```

Line 20 is the first **read** statement in the program and line 90 is the first **data** statement in the program. BASIC assigns the string **John** to **first$**, the string **Henry** to **middle$**, and the string **Madison** to **last$** .

The number of **read** and **data** statements need not match. You can enter the data values for several **read** statements in a single **data** statement or enter the data values for one **read** statement in several **data** statements. There is, however, a one to one correspondence between the variable and value pairs. Note that the **read** statement determines whether the data objects are interpreted as integers, floating-point numbers, or strings.

Each value in the list of data can be used only once. If all values have been used when a **read** statement attempts to assign a value to a variable, the system returns an error message.

For example,

```
20  read product$, price, sales%
            .
            .
70  read store$, arearep$, shipped, onorder
            .
            .
            .
900  data pencils, 0.7, 10853
910  data nourney's, adams, 1000
```

The two **read** statements require a total of seven data values. However, the two data statements provide only six. When the read statement at line 70 attempts to assign a value to **onorder**, there are no data available and the system displays the following message:

```
???Out of data at line 70
```

When assigning **data** statement values to variables, the **read** statement checks to see if the type of the variable and the type of the data value match. The system will assign an integer in a **data** statement to either an integer or floating-point variable, but will not assign a floating-point value to an integer variable.

The **read** statement determines whether the data objects are interpreted as integers, floating-point numbers, or strings. This means that you don't have to use a **%** after integer values you enter in a **data** statment. In fact, if you do include the **%**, the system displays the following error message:

```
???Bad input format in <line #>
```

Within **data** statements, all strings that contain significant spaces or a comma must be delimited by single or double quotation marks. If a string is not delimited by quotation marks, and it contains a comma, the system interprets the comma as a delimiter between elements in the **data** statement.

5.3 Restore

The **restore** statement instructs the system to return to the top of the data list and assign values to subsequent **read** statement variables, starting with the first value in the data list. The syntax for **restore** is as follows.

restore

The following is an example of the **restore** statement.

```
135  read a, b, c, d
140  restore
145  read e
150  data 1, 2, 3, 4, 5
160  print a, b, c, d, e
run
1, 2, 3, 4, 1
```

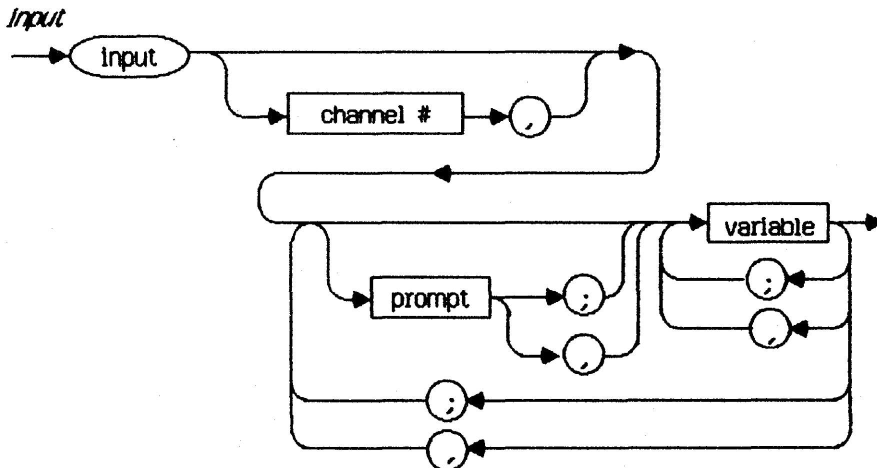
If line 140 is deleted above, the results are:

```
1, 2, 3, 4, 5
```

5.4 Input

The **input** statement assigns values to variables from a source other than a **data** statement. The statement retrieves the values from a designated source such as the console or a file. The default device is the console. The following is the syntax for the **input** statement.

Input



Channel # is a pound sign followed by an integer expression; **variable** is any legal variable name; **prompt** is a string constant that the system displays as a prompt when it executes the input command.

The following example shows how the **prompt** feature works.

```
10  input "Please enter your name and number. ";username$, usernumber
20  input "Now pick the topic that you want to research. "; topic$
 .
 .
 .
125  end
run
Please enter your name and number. Susan, 122
Now pick the topic that you want to research. stock market
```

Note that values entered by the user are separated by commas, as are values in the variable list.

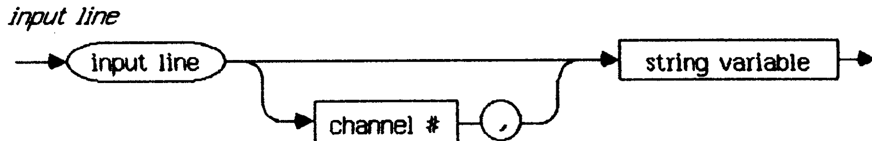
When you use the console (the default device for the **input** statement) the system displays a question mark (?) when it is ready to accept input. Values must be separated with commas, as above.

A response string cannot contain commas unless the string is enclosed in quotation marks. A way around this restriction is the **input line** statement.

5.5 Input Line

The **input line** statement requests input of one line from a specified device or file and assigns the line to the string variable in the statement. The following is the syntax for the **input line** statement.

input line



The default device is the keyboard. Characters are read into the string variable up to and including the first <RETURN>.

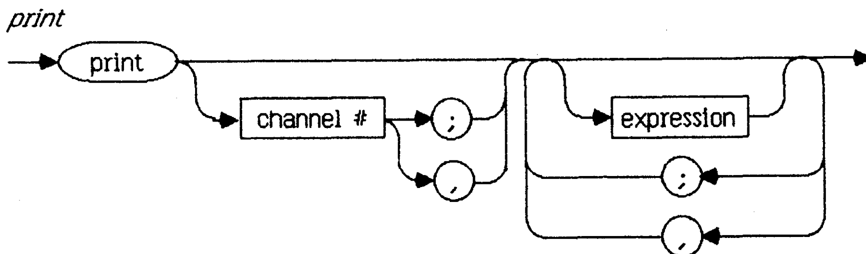
The following are examples of the input line statement.

100 input line address$	The keyboard is used as the source of input.
150 input line #5, county$	The file associated with data channel #5 is used as the source of input.

5.6 Print

The **print** statement is used to display data on the screen or to print to a file. The **print** statement outputs data to the console (the default file) or to another specified file. The following is the syntax.

print



The channel number must be one that is currently in use (associated with a file by an **open** statement). The punctuation (, ;) following the channel number is just a delimiter; it does not affect the print zones (described below).

The system divides each line on the screen into *print zones*. Each print zone is 14 characters wide.

The punctuation (, ;) that you use in a **print** statement determines the format of the output. Table 5-1 defines how the system formats output for each type of punctuation.

Table 5-1
Print Statement Punctuation

Punctu-ation	Example	Effect on Formatting
,	print A, B$ -12.50 credit	Each value begins in the next available print zone. When the current line is full, the next value is printed in the first zone of the next line.
;	print A; B$ -12.50 credit	Each value is printed immediately after the preceding value. Numeric values are formatted with spaces, as described below.

The system does not print a <RETURN> if the last character of a **print** statment is a comma (,) or a semicolon (;), and values from the next **print** statement are printed on the same line (if possible) .

Values from the next **print** statement are printed on the next line if you end a **print** statement without punctuation.

The list below gives other numeric formatting rules for the **print** statement.

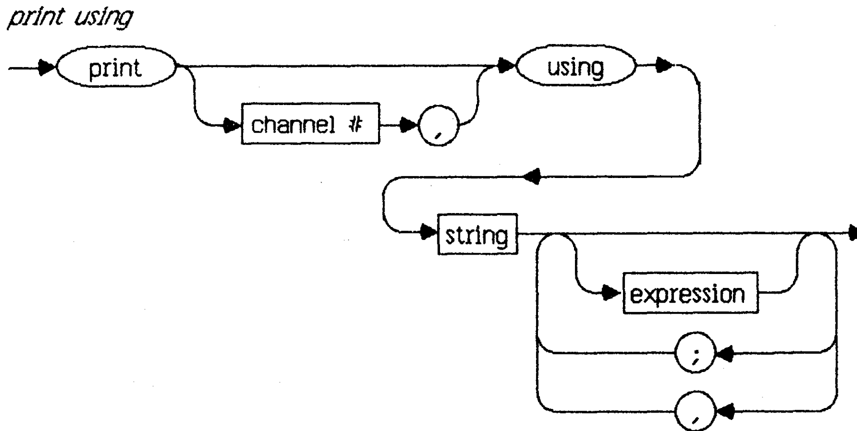
- Leading zeroes and nonsignificant trailing zeroes are suppressed. If a floating-point value has no fractional part, the decimal point is also suppressed.
- For integers and floating point numbers, the printed value has a trailing space. If positive, it has a leading space; if negative, it is preceded by a minus sign.
- Very large or small values are printed in scientific notation with a leading and a trailing space.

Note: Additional screen control is possible using special characters; see the Workshop User's Guide *for more information.*

5.7 Print Using

The **print using** statement, like the **print** statement, outputs data to a specified file. However, the **print using** statement uses a string (that you specify) as a guide for printing the information. In other words, the output is printed according to a specified format. The following is the syntax for the **print using** statement.

print using



The channel number must be one that is currently in use (associated with a file by an **open** statement). **String** is a string constant, variable, or expression that contains the format field for the statement. The system prints the list of data in the format specifed by the **string**.

Table 5-2 defines the **print using** format field characters. These characters dictate the format for the output.

Table 5-2
Print Using Statement Formatting Characters

Format Character	Effect on Output
!	Represents a one-character string.
\ \	Represents a string field of two or more characters: \\ is a two-character field, \<space><space>\ is a four-character field, and so on.
# .	Defines numeric formats. Each # represents a digit; the period (.) marks the decimal point. ###.## prints up to 999.99. Up to 15 formatting characters are permitted. Leading zeroes are replaced by spaces. A number that is too large for the format is printed unformatted, preceded by %.
-	When placed at the end of a numeric format, prints a trailing minus (-) for all negative values. For example, ###.##-
$$	When precedes a numeric format causes a dollar sign ($) to be printed before the first digit of the following numeric field.
,	If any commas appear within a numeric format to the left of the decimal point, commas will appear every three digits in the result. Such commas in the format also allocate space in the same way as #.
**	Precedes a numeric format; prints asterisks (*) instead of spaces within numeric output.
^ ^ ^	Follows a numeric field to indicate positions for scientific notation. At least three (^ ^ ^) are required, up to five (^ ^ ^ ^ ^) may be necessary.

NOTES

Chapter 6
Branching Statements

6.1 If Then Else 6-1
6.2 If Goto 6-2
6.3 On Goto 6-3
6.4 On Gosub 6-3
6.5 On Error Goto 6-4
6.6 Resume 6-5
6.7 Goto 6-6

Branching Statements

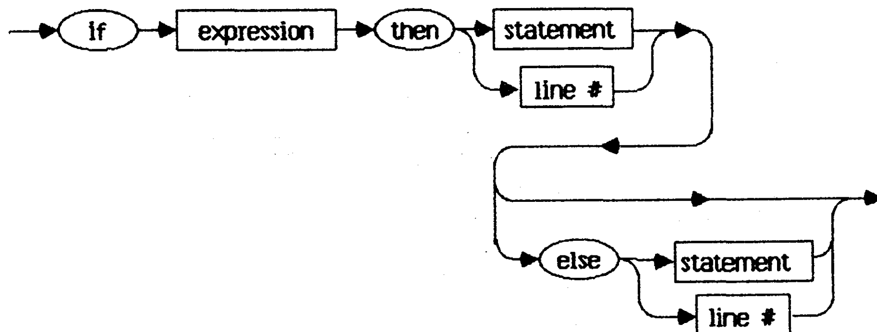
Branching statements modify the order of statement execution while a program is running. BASIC includes both conditional and unconditional branching statements. A conditional branching statement causes program execution to branch if a condition is met. Unconditional branching statements always branch.

6.1 If Then Else

The **if then else** statement is a conditional branching statement. If a condition is met (true), then whatever is in the **then** clause is executed. If a condition is not met (false), the **then** clause is not executed. If the expression is false and the **if then else** statement does not have an **else** clause, the system executes the first statement of the following program line, disregarding statements in a multiple statement line.

If you use the **else** clause, **if then** executes either the statement in the **then** clause, or the one in the **else** clause. If the condition is met, then something is executed, otherwise something **else** is executed. The following is the syntax for the **if then else** statement.

if then else



Expression must be an expression or integer variable, the value of which can be interpreted as either true or false; **statement** is a valid BASIC statement; **line #** is a number of a line within the program.

The following example compares sales to projected sales, setting a variable to true if sales fall below a certain point.

```
35 if sales < (projected * .9) then margintoolow% = 1  ! 1=true
```

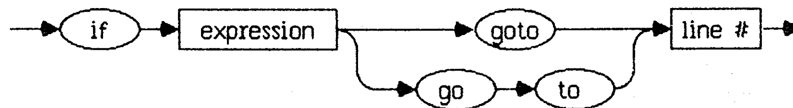
To set **margintoolow%** to false when sales *don't* fall below the critical point, you could use an **else** clause as follows:

```
35  if sales < (projected * .9) then margintoolow% = 1 &
    else margintoolow% = 0                  ! 0 = false
```

6.2 If Goto

The **if goto** statement skips to a different part of the program during execution if a condition is met. The following is the syntax for the **if goto** statement:

if goto



Expression is an expression whose value can be interpreted as true or false; **line #** is a valid line number within the program.

Goto can be broken into two words, **go to**, if you wish.

For example, assume you are writing a program that updates employee salaries. To recalculate salaries for those employees who have gotten a raise or been promoted since the last update, you could use an **if goto** statement as follows to direct the program to the salary recalculation and posting routine for all employees whose salaries need adjustment.

```
50 if datechanged <> dateupdated goto 1000
```

The statement checks the date the records were last updated with the date the employee's salary was last changed. If the employee's salary was changed since the last update, the program continues execution at line 1000, the beginning of the salary recalculation routine.

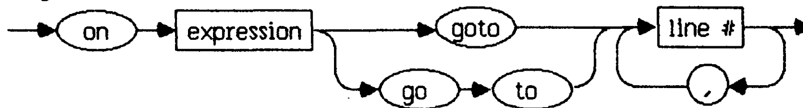
The example below is a program segment that checks data that the user enters at the keyboard and skips to line 1000 if the data are not within the defined limits.

```
10 input "Enter a number between 1 and 10"; number%
20 if not(number% >= 1 and number% <= 10) then &
     goto 1000
1000 rem Data input error routine starts here.
```

6.3 On Goto

The **on goto** statement transfers control to one of a list of line numbers. The following is the syntax for the **on goto** statement:

on goto



Goto can be broken into two words, **go to**, if you wish.

After evaluating **expression**, the **on goto** statement transfers control to the line with the position in the list that corresponds to the value of **expression**. For example, if the value of the expression is 1, the next line executed is the first line in the list. For example:

```
45 on howship% goto 100, 200, 300, 400
```

If the value of **howship%** is 1, control transfers to line 100; if the value of **howship%** is 2, control transfers to line 200, and so on. If, however, **howship%** is less than 1 or exceeds 4, the system displays the message:

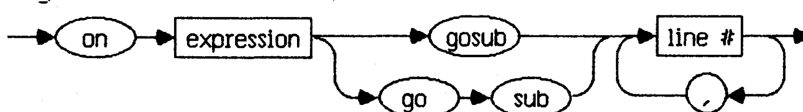
```
???on goto range error in line 45
```

The maximum number of line references in an **on goto** statement is 255.

6.4 On Gosub

The **on gosub** statement transfers control to one of a list of subroutines (subroutines are explained in detail in Chapter 10, Subroutines and Functions). The following is the syntax for the **on gosub** statement.

on gosub



Expression should be an expression which gives an integer result or an integer variable. The value of **expression** is an offset into the list of line numbers. The line numbers are the subroutine entry points.

Gosub can be broken into two words, **go sub**, if you wish.

The **on gosub** statement invokes the subroutine beginning at the line with the position in the list that corresponds to the value of **expression**. If the value is 4, for example, control transfers to the fourth line number in the list. Note that if the value of **expression** is less than 1 or greater than the number of lines listed, the system generates a run-time error.

Use the **on gosub** statement whenever you program multiple branches to subroutines. For example, you could use the following **on gosub** statement to direct execution of the subroutines:

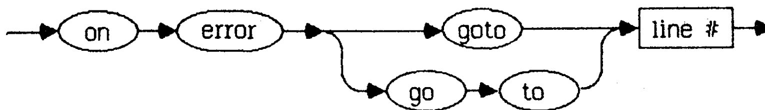
25 on choice gosub 30, 100, 300, 375, 500

Each **line #** is the entry point of a subroutine. The maximum number of line references in an **on gosub** statement is 255.

6.5 On Error Goto

The **on error goto** statement directs program execution to an error-handling routine when a recoverable run-time error is encountered. The following is the syntax for the **on error goto** statement.

on error goto



Line # represents the entry point into an error-handling routine. **Goto** can be broken into two words, **go to**, if you wish.

***Note:** Use the **resume** statement (described below) to exit the error-handling routine.*

BASIC provides two system integer variables for use in error handling; these may be printed or examined by the program in the error-handling routine:

- **err** contains the number of the error; a list of error numbers and their meaning is found in Appendix D, Error Messages.
- **erl** contains the line number of the statement that produced the error.

On error goto statements may appear anywhere in a program and must be executed before they take effect. If an **on error goto** statement has been executed, any recoverable error causes the program to branch to the specified line number. To disable a previously executed **on error goto** statement, execute **on error goto** with no line number or **on error goto 0**. To specify a new error-handling routine at line n, execute **on error goto n**.

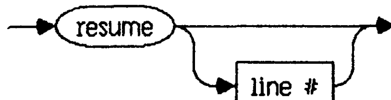
Here is an example of a program with an error-handling routine:

```
100 A$ = "current"
110 on error goto 500  ! prompt for valid filename
120 open A$ for input as file #3
130 rem File is open, so turn off error handling.
140 on error goto
...
500 rem Error handler checks for missing input file.
510 if err = 5% then goto 540  ! 5=file not found
520 on error goto  ! turn off error handler
530 resume  ! resume at open, line 120
540 print "FILE ";A$;" AT LINE ";erl;" NOT FOUND."
550 input "ENTER INPUT FILENAME..."; A$
560 resume  ! resume at open with new filename
```

6.6 Resume

The **resume** statement clears the current error and allows program execution to continue after an error has been handled. It should *always* be used to exit from an error-handling routine entered via **on error goto**. The following is the syntax for the **resume** statement.

resume



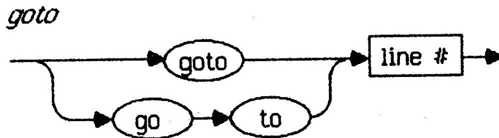
Resume 0 (which is the same as **resume** without a line number) causes the program to continue at the line that caused the error. If there are multiple statements on the line, **resume** resumes at the **dim, def, fnend, for,** or **next** statement immediately preceding the statement that caused the error.

Resume n resumes at line n.

6.7 Goto

The **goto** statement is an unconditional branching statement. In other words, the **goto** statement always transfers control to the specified program line. The following is syntax for the **goto** statement.

goto



Line # must be a valid line number that exists in the program. For example, the program line:

125 goto 335

transfers program exection to line 335.

Goto can be broken into two words, **go to**, if you wish.

NOTES

Chapter 7
Looping Constructs

7.1 For Next 7-1

7.2 While Next 7-2

7.3 For While 7-4

7.4 For Until 7-5

7.5 Until Next 7-6

7.6 Nested Loops 7-7

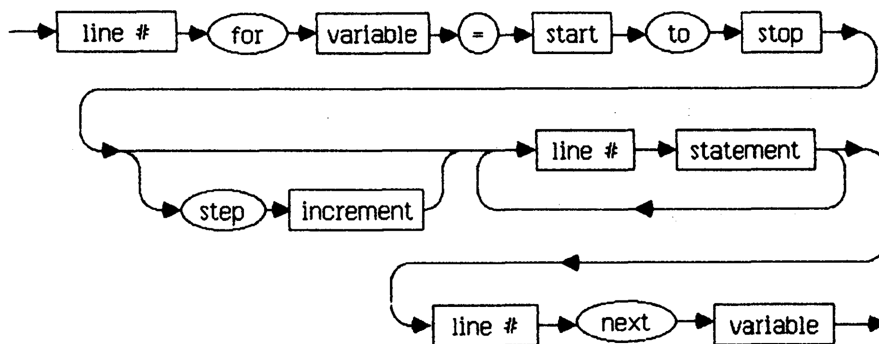
Looping Constructs

Looping constructs allow you to execute blocks of statements a specified number of times or until some condition is met.

7.1 For Next

The **for next** construct controls how many times the program executes the block of code between the two statements. The following is the syntax for the **for next** construct.

for next



Variable, the control variable for the loop, is used to determine the number of times the block of code bounded by the **for** and **next** statements is repeated. Note that the **variable** in the **for** clause must be the same as **variable** in the **next** statement.

Start, **stop**, and **increment** are numeric expressions. **Start** is the initial value of the control variable, **stop** is its final value, and **increment** is the quantity added to the value of the variable at the completion of each iteration of the loop. **Increment** can be either a positive or negative numeric value but cannot be 0; 1 is the default value.

The **for next** construct executes the statement or statements bounded by the **for** and **next** statements until the value of the control variable exceeds the limit specified in the **for** statement. Once the value of that variable exceeds the limit, program execution resumes at the statement following the **next** statement.

If the initial value of the control variable exceeds **stop** before the first iteration of the loop, the system ignores the statements bounded by the **for** and **next** statements and continues execution at the statement following the

next statement. (This situation exists if **increment** is positive and **start** is greater than **stop** or if **increment** is negative and **start** is less than **stop**.) For statements such as:

```
10 for iters = 4 to 1
40 for loops = 1 to 4 step -1
```

cause the system to skip to the statement following the associated **next** statement.

The **for** statement specifies the number of iterations of a loop. The number of iterations can be set prior to executing a program or can depend upon the run-time value of a variable. **For next** constructs such as:

```
70 for iterations = 1 to 4
        .
        .
        .
90 next iterations
```

execute the enclosed block of code a specific number of times. **For next** constructs such as:

```
45 for loops% = 1% to numberofrecords%
        .
        .
        .
80 next loops%
```

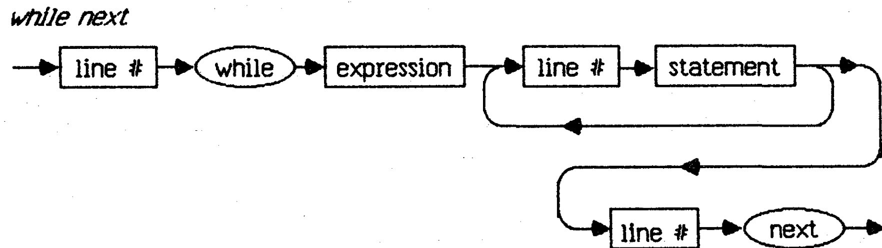
execute the bounded code the number of times that corresponds to the run time value of an integer variable (in this case, **numberofrecords%**).

Looping with integer indices is much faster than looping with floating-point indices.

7.2 While Next

The **while next** construct executes the enclosed block of code while the specified condition is true. The following is the syntax for the construct.

while next



The **while next** construct tests whether or not the **expression** is true before each iteration of the enclosed code. If the expression in the **while** statement is false before the first iteration of the loop, the system skips to the line following the **next** statement. For example, in the loop:

```
100 while 1 > 2
110     numberofproblems% = numberofproblems% + 1%
120 next
```

the system never executes line 110 because 1 is never greater than 2.

The **next** statement of a **while next** construct is only a delimiter for the loop; the **next** clause cannot include a variable name, and does not increment any variable in the expression in the **while** statement. Therefore, if you don't modify the value of the expression in the **while** statement within the bounds of the construct, the system cannot exit the loop.

For example,

```
150 while ordercount% < max%
160     input 'Please enter the next order.', ordernumber%
170 next
```

Once the system enters this loop, it never exits, because line 160 doesn't change the values of **ordercount%** or **max%**, the variables that determine the truth of the expression in the **while** statement. The only way to interrupt the loop is to use the -period interrupt.

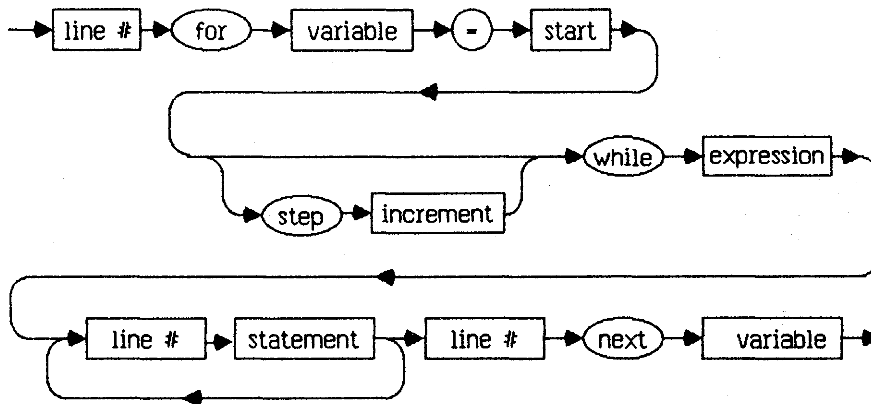
The following is a correct example of a **while next** construct that inputs 10 values into the array **a**.

```
100 i = 10
110 while i > 0
120     input "next value", a(i)
130     i = i - 1
140 next i
150 print "done"
```

7.3 For While

The **for while** construct executes a loop **while** a condition remains true. Looping ceases when the condition in the **while** clause becomes false. The following is the syntax for the **for while** construct.

for while



The **for while** construct tests whether or not the expression is true before each iteration of the enclosed code. If the initial value of the expression is false, the system skips the enclosed code and executes the statement following the **next** statement.

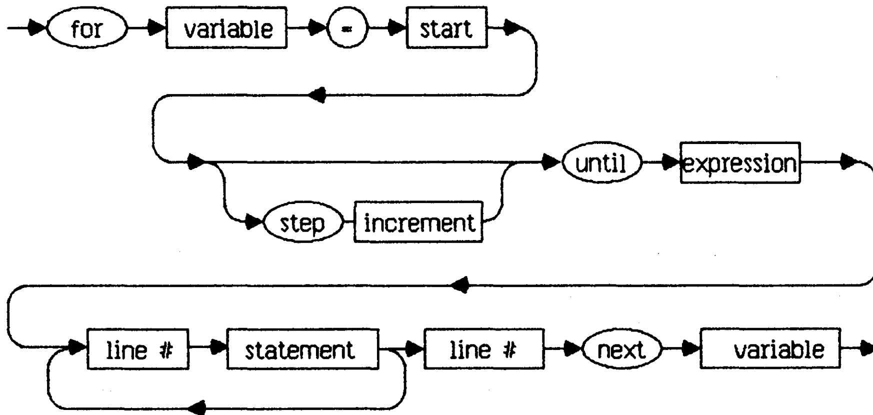
The following example of a **for while** construct concatenates the elements of the string arrays **A$** and **B$** as long as the elements of **A$** are not null.

```
150 for i = 1 while A$(i) > ""
160     A$(i) = A$(i) + B$(i)
         .
         .
         .
200 next i
```

7.4 For Until

The **for until** construct executes a loop **until** a condition becomes true. Looping ceases when the condition in the **until** clause becomes true. The following is the syntax for the **for until** construct.

for until



The **for until** construct tests whether or not the expression is true before each iteration of the enclosed code. If the initial value of the expression is true, the system skips the enclosed code and executes the statement following the **next** statement.

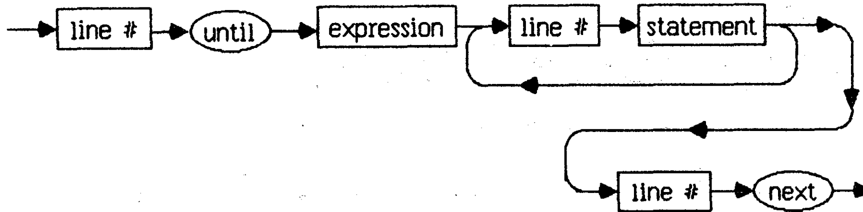
The following is an example of a **for until** construct.

```
150 for i = 10 until i >= 50 or A(i) >= 100
160     A(i) = A(i) + A(i-1)
         .
         .
         .
200 next i
```

7.5 Until Next

The **until next** construct executes the bounded code until the condition in the **until** clause is true. The following is the syntax for the construct.

until next



The **until next** construct tests whether or not the expression is true before each iteration of the enclosed code. If the initial value of the expression is true, the system skips the enclosed code and executes the statement following the **next** statement.

As with the **while next** construct, the **next** statement is only a delimiter of the loop; the **next** statement cannot contain a variable name, and it does not affect any variable in the **until** clause expression.

If you don't modify the value of one or more of the variables in the **until** statement expression within the bounds of the construct, the system cannot exit the loop.

The following **until next** construct will continue requesting values until **0.0** is input. Line 190 guarantees that at least one pass through the loop will be made.

```
190 v = -1.0
200 until v = 0.0
210     input "next value?", v
        .
        .
        .
300 next
```

7.6 Nested Loops

Any of the looping constructs can appear within the code bounded by another looping construct. But the two statements of the nested construct must occur between the beginning and ending statements of the outer construct, as below:

```
begin outer construct
        .
        .
    begin inner construct
        .
        .
    end inner construct
        .
        .
end outer construct
```

The following is an example of nesting.

```
10 for cycle = 1 to 4
20    print 'Cycle', cycle
25    print
30    for subcycle = 1 to 10
40       print 'Subcycle'; subcycle,
50    next subcycle
55    print
60 next cycle
70 end
```

The following is an incorrect example of nesting.

```
10  for cycle1 = 1 to 3
20       print 'cycle1'; cycle1
30       for cycle2 = 1 to 4
40       print 'cycle2'; cycle 2
50  next cycle1
60  next cycle 2
70  end
```

The example above is incorrect because the inner construct is not contained within the outer construct. To make it correct, lines 50 and 60 should be reversed.

Chapter 8
Statement Modifiers

8.1 The If Statement Modifier 8-1
8.2 The For Statement Modifier 8-2
8.3 The While Statement Modifier 8-3
8.4 The Until Statement Modifier 8-3
8.5 The Unless Statement Modifier 8-4
8.6 Multiple Modifiers 8-4

Statement Modifiers

Statement modifiers modify statements. You can use the statement modifiers on many of the statements discussed in Chapters 6 and 7.

When a statement modifier is used in a multiple-statement line, the statement modifier qualifies only the statement it follows. For example:

```
235  print "hello" : print "goodbye" if greeting$ = 'g': print &
          "Choose an activity"
```

The **if** modifier affects only **print "goodbye"**; none of the other statements is modified.

8.1 The If Statement Modifier

The **if** statement modifier qualifies the execution of the preceding statement. In other words, the modified statement is executed **if** a condition is met (true). The following is the syntax for the **if** statement modifier.

if statement modifier

Statement is any valid BASIC statement; **expression** is any logical or relational expression. If the result of the **expression** is true, the **statement** is executed. Otherwise, execution is resumed at the next program line.

The **if** statement modifier is functionally equivalent to the **if then** construct. For example,

```
25  total% = total% + 1 if total% < 100
```

and

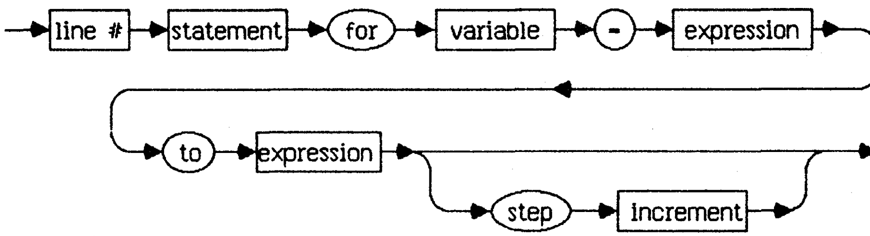
```
25  if total% < 100 then total% = total% + 1
```

do the same thing; the value of **total%** is incremented if its value is less than 100.

8.2 The For Statement Modifier

The **for** statement modifier executes the preceding statement the number of times specified in the **for** clause. The following is the syntax for the **for** statement modifier.

for statement modifier



The **for** statement modifier generates a loop that executes **statement** until **variable** reaches or surpasses the defined limit.

Functionally the **for** statement modifier is equivalent to a **for next** construct which affects one statement. The **for** statement modifier can be used with only one statement, unlike the **for next** construct. For example,

```
10  boxes% = boxes% + 10 for crates% = 1 to lastcrate%
```

It is equivalent to

```
10  for crates% = 1 to lastcrate%
20     boxes% = boxes% + 10
30  next crates%
```

In the following example, program execution is transferred to a profit calculation routine if the status of an item in inventory equals **chosen**.

```
50  for month% = 1 to 12
60     if status$(month%) = "chosen" then gosub 350
70  next month%
```

You can use a **for** statement modifier and do the same thing in one line.

```
60  gosub 350 if status$(month%) = 'chosen' for month% = 1 to 12
```

8.3 The While Statement Modifier

The **while** statement modifier executes the preceding statement until the condition specified in the **while** statement is false. The following is the syntax for the **while** statement modifier.

while statement modifier

The **while** statement modifier can control only one statement, which must modify the control expression, otherwise the loop will never terminate. The following is an example of a **while** statement modifier.

```
100 if profits$ = "UP" then bonus = bonus * 1.1 &
    while bonus < amount
```

8.4 The Until Statement Modifier

The **until** statement modifier executes the preceding statement until the value of the **until** expression is true. In other words, it executes the preceding statement as long as the **until** expression is false. The following is the syntax for the **until** statement modifier.

until statement modifier

The **until** statement modifier can control only one statement. Once the condition in the expression is true, execution passes to the next statement in the program. For example, the statement:

```
20 loops = loops + 1 until loops >= limit
```

increments the variable **loops** by one until the value of **loops** is greater than or equal to the value of the variable **limit**.

The **until** statement modifier creates an endless loop unless the statement affects the value of the **until** expression. For example,

```
30 loops = loops + 1 until revenues > anticipated
```

endlessly increments **loops**, unless **revenues** is initially greater than **anticipated**.

8.5 The Unless Statement Modifier

The **unless** statement modifier executes the preceding statement unless the expression in the **unless** statement is true. The following is the syntax for the **unless** statement modifier.

unless statement modifier

This statement modifier is especially useful when a decision to perform a task depends upon two conditions, as in the following example.

```
100 if balance <> onhand then &
        print 'OUT OF BALANCE' unless flag$ = 'errorok'
```

8.6 Multiple Modifiers

You can append more than one statement modifier to a single statement. For example, the following are legal BASIC statements.

```
10 length = length + 1 for iters = 1 to limit unless flag$ = 'stop'
20 share = share + .01 while share <= MAX unless flag$ = 'notelig'
30 print 'true' if i <> 3 for i = 1 to 10
```

NOTES

Chapter 9
Matrices

9.1 Dim 9-1

9.2 Mat 9-3

9.3 Mat Read 9-4

9.4 Mat Input 9-5

9.5 Mat Print 9-6

9.6 Matrix Calculations 9-7

9.6.1 Addition and Subtraction 9-8

9.6.2 Multiplication 9-9

Matrices

A matrix is an ordered collection of variables of the same type. Matrices are also called arrays. Valid variable names are used as matrix names. The last character of the name determines the type of all the data in the array.

Matrices can have one or two dimensions. A one-dimensional matrix is a single list of variables. The individual variables (or elements) within a matrix are numbered, starting with 0. To refer to an individual element within a one-dimensional matrix, you specify the name of the matrix, followed by the number of the element enclosed in parentheses. For example,

```
print projectedcost(3)
```

prints the contents of element number 3 in the matrix named **projectedcost**. The number of the element enclosed in parentheses, for example **(3)**, is called the *matrix subscript*. Since a one-dimensional matrix is a single list of variables, only one matrix subscript is needed to identify an element.

In a two-dimensional matrix, two subscripts are needed to specify an individual element. For example, the following elements are part of a matrix named **cleanup%**.

```
cleanup%(0,0)  cleanup%(0,1)  cleanup%(0,2)
cleanup%(1,0)  cleanup%(1,1)  cleanup%(1,2)
cleanup%(2,0)  cleanup%(2,1)  cleanup%(2,2)
```

Cleanup%(0,2) specifies the element in the first row and third column.

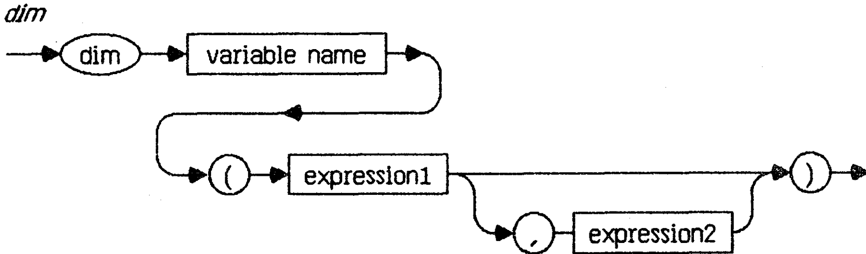
This chapter presents the statements that you use to define, fill, and access matrix elements, and explains the matrix arithmetic operations.

9.1 Dim

Whenever you create a matrix, you must tell BASIC the maximum number of elements and dimensions you want. The **dim** statement defines, or **dimensions**, the matrix; with it you can name one or more matrices and define the data type and maximum size for each matrix. The **dim** statement reserves a certain amount of space for the matrix. You don't necessarily have to use all the space you reserve, but you can't use more than was specified. The default dimension for all matrices is 10.

The following is the syntax for the **dim** statement.

dim



Variable can be an integer, floating-point, or string variable name. Each matrix can store only one type of value. The variable name determines the type of value the matrix accommodates. **Expression1** and **expression2** should have non-negative integer values.

A single **dim** statement can define more than one matrix, and these matrices can be of different types.

The number of rows and columns in a matrix are its dimensions. The values of **expression1** and **expression2** are the upper limits of a matrix's dimensions. **Expression1** is the highest row number and **expression2** is the highest column number.

Remember that the list of dimensions for all matrices begins with 0, so the number of elements in a matrix is always the largest subscript value plus 1. For example, the **dim** statement:

```
50 dim junk% (12, 12)
```

creates a matrix that has 13 rows and 13 columns. However, unless you specifically access the zero'th row and column, they are ignored.

Examples:

```
20  dim shoes(2)

66  dim shoes(2), shoes%(2,2), shoes$(15,14)
```

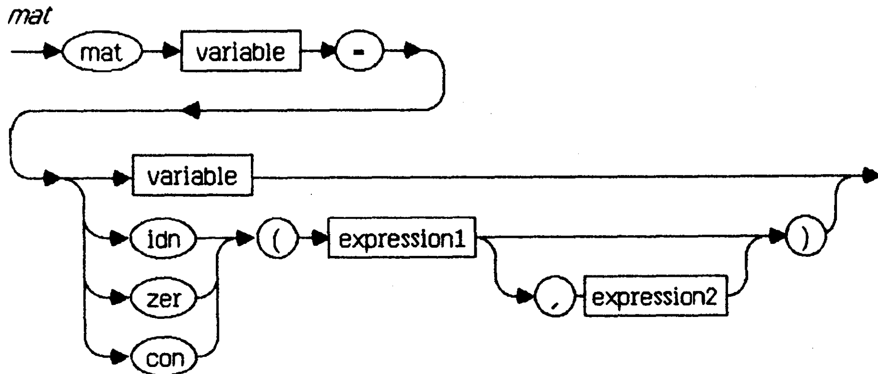
Several of the matrix operations allow you to redimension a matrix after defining its size in a **dim** statement. However, you cannot make a matrix larger than its original size or change between one-dimensional and two-dimensional matrices.

Note: *The maximum size for a non-virtual array is 32K bytes. The maximum number of array elements (including the zero'th element) is 2730 for real arrays, 16383 for integer arrays.*

9.2 Mat

The **mat** statement is the matrix initialization statement. The following is the syntax for the **mat** statement.

mat



The first **variable** must be the name of an already dimensioned matrix; **expression1** and **expression2** are its dimensions. You can use this **variable** to redimension the matrix, but remember that you cannot make the matrix larger than its original size. Zer sets all elements of the matrix to zero (the default value for elements in a newly created matrix); **con** results in a matrix of all ones; **idn** sets the matrix elements to one on the diagonal where row number equals column number, and all other elements in the array to zero.

Examples:

```
35 mat junk% = idn (20, 20)
99 mat dentist = zer (12,1)
22 mat pagoda = con
```

You can also use the **mat** statement to assign the value of one matrix to another matrix. For example,

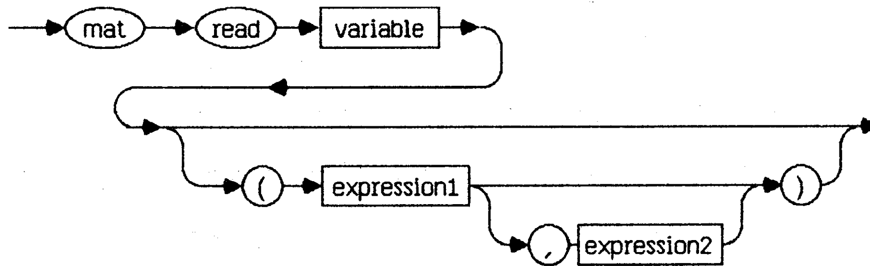
```
35 mat comp = jade
```

assigns the value of **comp** to **jade**, redimensioning **jade** if necessary.

9.3 Mat Read

The **mat read** statement loads values from one or more **data** statements into one or more matrices (see Section 5.2, Read and Data). The following is the syntax for the **mat read** statement.

mat read



Variable is the matrix name; **expression1** and **expression2** are the matrix dimensions. If you don't specify dimensions, the current dimensions of the matrix are assumed; if you do include them, the statement redimensions the matrix to conform. However, the system cannot increase the number of elements in the matrix or change between one-dimensional and two-dimensional matrices. If no dimensions follow the name of a matrix, Lisa BASIC fills the entire matrix with values from the **data** statement beginning with row 1 and proceeding to the next row as each row is filled.

In a program that loads the values of a two-by-three matrix from within the program, the **dim**, **mat read**, and **data** statements could be:

```
20 dim stock (2,3)
30 mat read stock
          .
          .
          .
100 data 25.8, 18.75, 17.25, 56.7, 98.6, 125.9
```

The **dim** statement creates a six-value matrix and the **mat read** statement instructs the program to read these values from the **data** statement. The program fills the matrix row by row (i.e., for each row, the row stays fixed, the column varies). The following table shows **stock** after the **mat read** statement assigns values to each element.

		Column # (row, column)		
		1	2	3
Row #	1	25.8	18.75	17.25
	2	56.7	98.6	125.9

If the **data** statement doesn't contain enough values to fill the matrix, the system displays the error message:

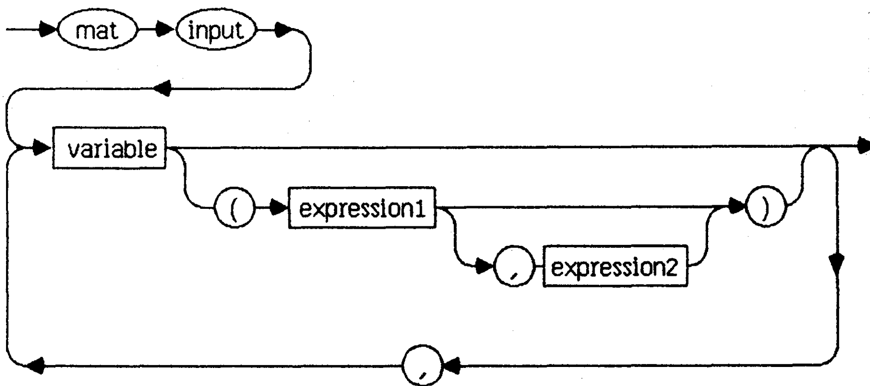
??? Out of data in line X

where X is the line number of the **mat read** statement.

9.4 Mat Input

The **mat input** statement loads values into one or more matrices from the keyboard or from a file. The following is the syntax for the **mat input** statement.

mat input



Variable is the matrix name; **expression1** and **expression2** are the matrix dimensions. If you don't specify dimensions, the current matrix dimensions are assumed; if you do include them, the statement redimensions the matrix to conform. However, the system cannot increase the number of elements in the matrix, or change between one-dimensional and two-dimensional matrices. When you input matrix values from the keyboard, the **mat input** statement displays a question mark when the program is ready for the matrix values. The values you enter must be of the same type as the matrix.

There are two system variables, **num** and **num2**, that are set during execution of the **mat input** statement to describe the size of the entered array. **Num**

contains the number of rows for a two-dimensional matrix, or the number of elements for a one-dimensional matrix. **Num2** contains the number of columns in a two-dimensional matrix.

Unlike the **input** statement, the **mat input** statement displays the prompt once and accepts the values only until the user types <RETURN>. Therefore, be careful not to press the <RETURN> key before entering the last matrix value.

The **mat input** statement requires a comma between values. For example, if you enter the following seven values after the question mark prompt, the **mat input** statement interprets them as one value.

```
? 10 20 30 40 50 60 70
```

To enter the values as separate elements in a matrix, you must enter a comma as a delimiter between values.

```
? 10, 20, 30, 40, 50, 60, 70
```

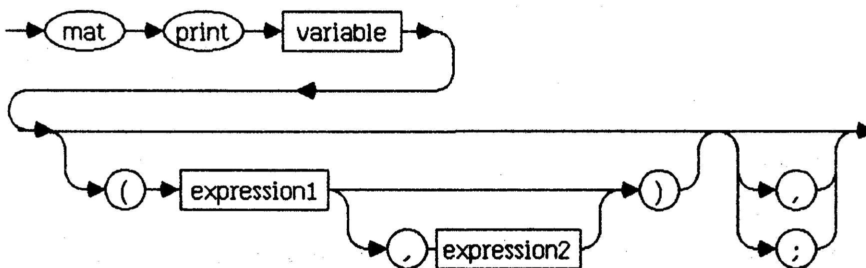
Example:

```
10 dim a(10)
20 mat input a(10)
```

9.5 Mat Print

The **mat print** statement prints all or a portion of the named matrix. However, the zero'th row and column of a matrix are never printed by the **mat print** statement. The following is the syntax.

mat print



Variable is the variable name associated with a matrix; **expression1** and **expression2** are the matrix dimensions. If you don't specify dimensions, the current dimensions of the matrix are assumed. If you include dimensions, the statement prints only the portion of the matrix that you specify; it does not redimension the matrix.

The comma and the semicolon determine the print format of a matrix. The punctuation is the same as for the **print** statement. Refer to Section 5.6, Print, for more information.

Example:

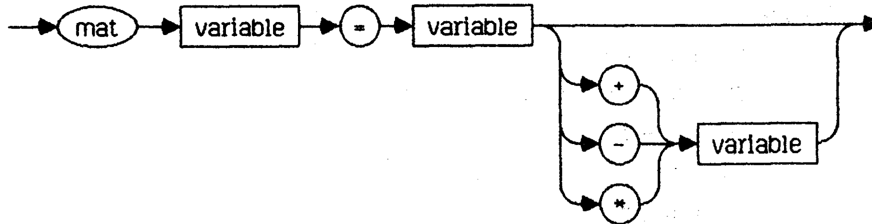
```
10  dim a(10)
20  for x = 1 to 10
30     a(x) = x
40  next x
50  mat print a;
60  print
70  mat print a
80  end
run
1 2 3 4 5 6 7 8 9 10
1
2
3
4
5
6
7
8
9
10
```

9.6 Matrix Calculations

You can add, subtract, and multiply matrices. There are also five built-in matrix functions: **trn**, **inv**, **det**, **linsys**, and **cond**; refer to Section 10.5, Matrix Functions, for more information.

The following is the general syntax for matrix arithmetic operations:

general matrix arithmetic



The matrix on the left of the equal sign is redimensioned to conform to the dimensions of the resulting matrix. Only one matrix arithmetic operation can be performed per statement. For example,

```
30 mat result = effect - cost
```

is legal, while

```
30 mat result = effect - cost + deterioration
```

is illegal.

9.6.1 Addition and Subtraction

You can add or subtract matrices of the same dimensions (having the same number of rows and columns). However, the target matrix only needs to be large enough to accommodate the results. If the target matrix is larger than necessary, the system redimensions it to conform to the dimensions of the input matrices.

For example,

```
10 dim totals ( 31 ), store1 ( 7 ), store2 ( 7 )
        .
        .
        .
500 mat totals = store1 + store2
```

Store1 and **store2** have seven elements each. When the system adds the two together and stores the result in **totals**, it also redimensions **totals** to seven elements.

When adding or subtracting matrices, the system adds the values in corresponding positions and stores the result in the same position in the target matrix.

9.6.2 Multiplication

There are two types of matrix multiplication: *scalar multiplication* and multiplication of *conforming* matrices.

When you multiply a matrix by a scalar value, the system multiplies the value of each matrix element by that value. For example, the following line multiplies each element in **scaled.total%** by 10 and stores the result in the matrix **final.total%**.

```
100 mat final.total% = (10) * scaled.total%
```

Note that the keyword **mat** is necessary to identify this statement as a matrix calculation. The parentheses around the scalar value are also required.

Matrices **x** and **y** are conforming matrices if the number of columns in **x** is equal to the number of rows in **y**. For instance, the following **dim** statements define pairs of conforming matrices.

```
120 dim jan.graph ( 10, 30 ), feb.graph ( 30, 12 )
10 dim ratios ( 4, 12 ), inverses ( 12, 10 )
```

When you multiply conforming matrices, the matrix that receives the calculated values must have dimensions that can accommodate the number of rows of the first matrix and the number of columns of the second. The target matrix is redimensioned if necessary, but it cannot be dimensioned to a larger size. For example, if you multiplied **ratios** by **inverses** (above), the resulting matrix would be four by ten.

NOTES

Chapter 10
Subroutines and Functions

10.1 Gosub and Return 10-1

10.2 Nesting Subroutines 10-2

10.3 Arithmetic Functions 10-2

- 10.3.1 Abs 10-2
- 10.3.2 Sqr 10-2
- 10.3.3 Pi 10-2
- 10.3.4 Sin 10-2
- 10.3.5 Cos 10-2
- 10.3.6 Tan 10-3
- 10.3.7 Exp 10-3
- 10.3.8 Atn 10-3
- 10.3.9 Log 10-3
- 10.3.10 Log10 10-3
- 10.3.11 Int and Fix 10-3
- 10.3.12 Rnd and Randomize 10-4
- 10.3.13 Sgn 10-4
- 10.3.14 Intpart 10-4
- 10.3.15 Intpart% 10-5
- 10.3.16 Compound 10-5
- 10.3.17 Annuity 10-5
- 10.3.18 Time 10-5
- 10.3.19 Ccpos or pos 10-6
- 10.3.20 Tab 10-6
- 10.3.21 Swap% 10-6

10.4 String Functions 10-6

- 10.4.1 Len 10-6
- 10.4.2 Left 10-6
- 10.4.3 Right 10-7
- 10.4.4 Mid 10-7
- 10.4.5 Instr 10-7
- 10.4.6 + ... 10-7
- 10.4.7 Space$ 10-7
- 10.4.8 Chr$ 10-8
- 10.4.9 String$ 10-8
- 10.4.10 Xlate 10-8
- 10.4.11 Cvt Functions 10-8
  - 10.4.11.1 Cvt$$ 10-9

10.4.12 Sum$ 10-9
10.4.13 Dif$ 10-9
10.4.14 Prod$ 10-9
10.4.15 Quo$ 10-10
10.4.16 Place$ 10-10
10.4.17 Comp% 10-10
10.4.18 Val 10-10
10.4.19 Num$ 10-10
10.4.20 Num1$ 10-10
10.4.21 Ascii 10-11
10.4.22 Rad$ 10-11
10.4.23 Date$ 10-11
10.4.24 Time$ 10-11

10.5 Matrix Functions 10-11

10.5.1 Trn 10-11
10.5.2 Inv 10-12
10.5.3 Det 10-12
10.5.4 Linsys 10-12
10.5.5 Cond 10-12

10.6 Creating Your Own Functions 10-12

10.6.1 Def* 10-13
10.6.2 Fnend 10-13

10.7 Change 10-14

Subroutines and Functions

Subroutines and functions are blocks of code that perform specific tasks. Subroutines and functions serve different purposes and are used differently. This chapter explains the differences between the two and presents the related statements.

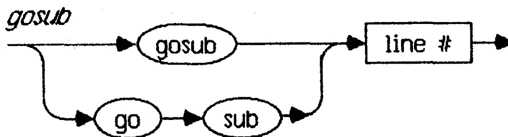
A subroutine is a separate block of code within a program that performs certain actions and then returns control to the main program. To invoke a subroutine, you use the **gosub** statement; to return to the main program, you use the **return** statement.

A function, however, is a block of code that returns a value. A function name can appear in a program anywhere a constant or variable of the same type as the function result can appear. BASIC provides arithmetic, matrix, and string functions and also allows you to define your own. The functions provided by BASIC are not part of your programs. Functions you create are a part of the programs you use them in. This chapter defines each of the functions provided and explains how to create your own.

10.1 Gosub and Return

The **gosub** statement requests execution of a subroutine. A subroutine is a block of code within the program which performs a specific task. The **return** statement is placed at the last line in the subroutine to return program execution to the program line after the one which contains the **gosub** statement. The following is the syntax for the **gosub** statement.

gosub



Line number is the entry point to a subroutine within the program. **Gosub** can be broken into two words, **go sub**, if you wish.

The following is the syntax for the **return** statement.

return

When the system executes the **return** statement of a subroutine, control passes to the statement immediately after the **gosub** statement.

10.2 Nesting Subroutines

A subroutine can call another subroutine which in turn can call a third subroutine, and so on. The **return** statement of each subroutine returns control to the statement following the **gosub** statement that initiated execution of that subroutine. Therefore, a subroutine can call itself. The maximum level of nesting depends upon the size of the program and the amount of available memory.

A subroutine can have more than one entry point; in fact, you can use any line number within a subroutine for the line number in the **gosub** statement.

10.3 Arithmetic Functions

The functions **sqr, pi, sin, cos, tan, exp, atn, log, log10, compound,** and **annuity** return approximate values only.

10.3.1 Abs

The **abs** function returns the absolute value of the argument. The format of the **abs** function is as follows.

abs (a)

The argument a is a numeric value.

10.3.2 Sqr

The **sqr** function returns the square root of the argument. The format of the **sqr** function is as follows.

sqr (a)

The argument a is a numeric value.

10.3.3 Pi

The **pi** function returns the constant which approximates the value of π (3.14159...). The value of π is the ratio of a circle's circumference to its diameter. The format of the **pi** function is as follows.

pi

The **pi** function requires no arguments.

10.3.4 Sin

The **sin** function returns the sine of the argument. The format of the **sin** function is as follows.

sin (a)

The argument a is a numeric value, in radians.

10.3.5 Cos

The **cos** function returns the cosine of the argument. The format of the **cos** function is as follows.

cos (a)

The argument a is a numeric value, in radians.

10.3.6 Tan

The **tan** function returns the tangent of the argument. The format of the **tan** function is as follows.

tan (a)

The argument **a** is a numeric value, in radians.

10.3.7 Exp

The **exp** function returns the exponential value of the argument, e^a, where e=2.71828.... The format of the **exp** function is as follows.

exp (a)

The argument **a** is a numeric value.

10.3.8 Atn

The **atn** function returns the arctangent of the argument. The format of the **atn** function is as follows.

atn (a)

The argument **a** is a numeric value, in radians.

10.3.9 Log

The **log** function returns the natural logarithm ($\log_e x$) of the argument. The format of the **log** function is as follows.

log (a)

where $e^{\log(a)}=a$ or **exp(log(a))=a**

10.3.10 Log10

The **log10** function returns the base 10 logarithm ($\log_{10} x$) of the argument. The format of the **log10** function is as follows.

log10 (a)

where $10^{\log10(a)}=a$, or10^**log10(a)=a**

10.3.11 Int and Fix

Both the **int** and **fix** functions return the integer part of x as a floating-point value. The formats of the two functions are:

int (a)

fix (a)

If the floating-point value is already an integer value, either function returns that value. Otherwise, for positive values of a, the functions both return the largest integer that is not greater than a. For example, **int (2.5)** and **fix (2.5)** both return the value 2.

Int and **fix** handle negative values of a differently. **Int** rounds towards negative infinity, while **fix** rounds towards zero. For example, **int (-32.355)** yields -33, while **fix** (-32.55) yields -32.

10.3.12 Rnd and Randomize

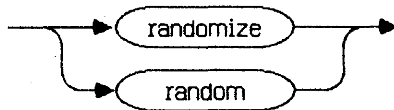
The **rnd** function generates the next number in a sequence of values greater than 0 but less than 1. Each time you execute a program **rnd** generates the same sequence of values, until you execute the **randomize** statement.

The **randomize** (or **random**) statement changes the starting point in the sequence to a random one. The following example generates twenty integer values between one and ten. A different set is generated each time you run it:

```
30 dim r(20)
40 randomize
50 for i=1 to 20
60     r(i) = int(1+10*rnd)
70 next i
```

The following is the syntax for the **randomize** statement.

randomize



10.3.13 Sgn

The **sgn** function determines whether the argument is positive, negative, or zero. The format is :

sgn (a)

The **sgn** function returns the following values.

0 if **a** = **0**

1 if **a** has a positive sign

-1 if **a** has a negative sign

10.3.14 Intpart

The **intpart** function returns the integer part of x as a floating-point value. The format is:

intpart (x)

Unlike **int** or **fix** (described in Section 10.13.11), the current rounding mode is observed; see Chapter 13, Advanced Floating-Point Manipulation, for more information on rounding modes.

10.3.15 Intpart%

The **intpart%** function returns the integer part of x, as an integer value. The format is:

intpart% (x)

Like **intpart**, the current rounding mode is observed.

10.3.16 Compound

The format of the **compound** function is:

compound (i,n)

where **compound (i,n)** = $(1+i)^n$. This function is used to determine the effect of compound interest. For example, given present value of principal **pv**, and periodic interest rate i, to compute future value of principal **fv** after **n** periods:

```
100  fv = pv * compound (i,n)
```

10.3.17 Annuity

The format of the **annuity** function is:

annuity (i,n)

where **annuity (i,n)** = $(1 - (1+i)^{-n})/i$

This function is used to determine the present value of **n** equal payments at interest rate **i**. For example, given amount of loan **p**, and periodic interest rate **i**, to compute the amount of **n** equal periodic payments, **pp**:

```
100 pp=p/annuity(i,n)
```

10.3.18 Time

The **time** function returns a number, in seconds. **Time(1)** tells you how long BASIC has been running. **Time(2)** tells you how long the program has been running in the workspace. **Time(-1)** gives you the time BASIC began, in seconds since midnight. **Time(-2)** gives you the time the current program began, in seconds since midnight. The format of the function is:

time(n)

where **n** is an integer argument. Therefore, if **n=1** then **time(1)** tells how long BASIC has been running. If BASIC has been running 30 minutes and 45 seconds, then **time(1)** returns 1845.

If **n=0** or **n>2** or **n<-2**, then **time(n)** returns the current time, in seconds since midnight.

10.3.19 Ccpos or pos

The **ccpos** or **pos** function returns the current position of the print head for a specified input/output channel. The format of the function is:

ccpos (i%)

pos (i%)

where **i%** is the channel number.

10.3.20 Tab

The **tab** function, when used with a **print** statement, moves the printing position to a specified column. The format is:

tab (i%)

where **i%** is an integer expression that results in the column number where you want the print position.

10.3.21 Swap%

The **swap%** function swaps the upper and lower bytes of an integer. The format is:

swap% (i%)

where **i%** is an integer expression.

10.4 String Functions

The string functions make handling alphanumeric strings easier. Character strings are sequences of characters bounded by quotation marks. Numeric strings are sequences of digits bounded by quotation marks. A numeric string can also include a plus (+) or a minus (–) sign, or a decimal point (.). Functions intended to apply to numeric strings produce undefined results if applied to other strings.

10.4.1 Len

The **len** function returns the number of characters, including trailing blanks, in the specified string. The format for **len** is:

len (s$)

The argument **s$** is a string variable.

10.4.2 Left

The **left** function returns a specified number of characters of a string, starting at the first character in the string. The format for **left** is:

left (s$, n)

s$ is a string variable; **n** is the number of characters to be extracted.

If **n** is equal to 0, the result is a null string. If you specify a value larger than the number of characters in the string, **left** returns the contents of the entire string.

10.4.3 Right

The **right** function returns a subset of the string, beginning with the character in the specified position of the string and ending with the last character in the string. The format for **right** is:

right (s$, n)

s$ is a string variable; **n** is the position of the character where the extraction begins.

If n is less than 1, the result is all of the string. If you specify a value larger than the number of characters in the string, **right** returns a null string.

10.4.4 Mid

The **mid** function extracts a substring of the string, beginning with the character in the specified position of the string continuing for a specified number of characters. The format for **mid** is:

mid (s$, m, n)

s$ is a string variable, **m** is the position of the first character, and **n** is the number of characters extracted. (This means that the position of the character where the extraction ends is **m+n+1**.) For example,

mid ('abcdefgh',3,5)

results in the string **'cdefg'**.

10.4.5 Instr

The **instr** function searches for a specified substring within a string. The format for **instr** is:

instr (n, s$, a$)

s$** is a string variable; **a$ is the substring; **n** is the position in the string where the search is started. If **a$** is found, its character position is returned. If **a$** is not in **s$**, 0 is returned. If **a$** is null, 1 is returned.

10.4.6 +

The + sign concatenates two strings. The format is:

s$ + a$

The arguments **s$** and **a$** can be string variables or string constants.

10.4.7 Space$

The **space$** function creates a string of spaces of the specified size. The format of **space$** is:

space$ (x%)

The argument **x%** is an integer value.

10.4.8 Chr$

The **chr$** function returns a single character that is the ASCII equivalent of the specified numeric value. The format is:

chr$ (n)

The argument **n** is a number from 0 to 127.

10.4.9 String$

The **string$** function creates a string of the specified length, all elements of which are the specified value. The format is:

string$ (x,y)

x is size of the string; **y** is the ASCII value.

For example, to create a string of ten A's (ASCII value 65):

```
print string$(10,65)
```

10.4.10 Xlate

Xlate is used to translate the characters in a string. You give two strings: one to be translated, and one to be used as a "table" for the translation. The ASCII value of each character in the first string is used to pick out the new character from the second.

For example, the first character in the second string is picked if the ASCII value of the character from the first string is 0. If the ASCII value of the first string character is 1, it is translated into the second character in the second string. If the first string character's value is 3, the fourth character from the second string is used. The format for the **xlate** function is:

xlate (s$,t$)

s$** is the string to be translated; **t$ is the "table".

10.4.11 Cvt Functions

The cvt functions map values between numeric and string data. Note that "mapping" means copying the bit pattern, *not* converting the value. Five cvt functions are provided by BASIC:

s$ = cvt%$ (i%)	Maps the value of **i%** into **s$** (a two-character string). The result is a two character string.
i% = cvt$% (s$)	Maps the first two characters of **s$** into **i%**. The result is an integer.
s$ = cvtf$ (b)	Maps a floating-point expression **b** into **s$** (an eight-character string). The result is an eight-character string.
x = cvt$f (s$)	Maps **s$** into a floating-point value **x**.
x = cvt$$ (s$, n)	Edits the string **s$** (see below).

10.4.11.1 Cvt$$

The **cvt$$** function provides string editing. The editing is performed under the control of the specified argument. The format is:

cvt$$ (s$, n)

s$ is a character string; **n** is a control argument, which must be an integer value. **n** is a "bit mask". The control values have the following meanings:

1	Trim the parity bit from each character in the string.
2	Remove all spaces and tabs from the string.
4	Remove all carriage returns, line feeds, form feeds, rubouts, and **null** characters from the string.
8	Remove the leading spaces and tabs from the string.
16	Reduce groups of multiple spaces or tabs to a single space.
32	Convert lowercase letters to uppercase letters.
64	Convert [to (and] to).
128	Remove the trailing tabs and spaces from the string.
256	Prevent alteration of character within single (') or double (") quotation marks.

To obtain one control function, set **n** to that value. To obtain more than one control function, set **n** to the sum of the individual functions.

10.4.12 Sum$

The **sum$** function adds two numeric strings together and returns the result as a string. The format of the function is:

sum$ (s$, a$)

The arguments **s$** and **a$** are both numeric strings.

10.4.13 Dif$

The **dif$** function subtracts a numeric string from another and returns the result as a string. The format of the function is:

dif$ (s$, t$)

The arguments **s$** and **t$** are both numeric strings; **t$** is subtracted from **s$**.

10.4.14 Prod$

The **prod$** function multiplies two numeric strings, rounding the product to the specified number of spaces. The format of the function is:

prod$ (s$, t$, p)

s$** and **t$ are both numeric strings; **p** is the number of decimal places the result is rounded to.

10.4.15 Quo$

The **quo$** function divides two numeric strings, rounding the product to the specified number of spaces. The format of the function is:

quo$ (s$, t$, p)

s$** and **t$ are both numeric strings; **s$** is divided by **t$**. **p** is the number of decimal places the result is rounded to.

10.4.16 Place$

The **place$** function rounds the value of the specified numeric string to the specified number of spaces. The format of the function is:

place$ (s$, p)

s$ is a numeric string; **p** is the number of decimal places the result is rounded to.

10.4.17 Comp%

The **comp%** function compares two strings, returning a truth value based on the result. The format of the function is:

comp% (s$, t$)

The arguments **s$** and **t$** are numeric strings. The truth values, and the conditions under which they are returned, are:

-1 if **s$** < **t$**

0 if **s$** = **t$**

1 if **s$** > **t$**

10.4.18 Val

The **val** function returns the numeric value of a numeric string. The format of the function is:

val (s$)

The argument **s$** is a numeric string.

10.4.19 Num$

The **num$** function returns the string of characters representing the numeric value x exactly as it would be output by the statement **print** x, including spaces, using E-format where necessary. The following is the format:

num$ (x)

10.4.20 Num1$

The **num1$** function returns the string of characters representing the numeric value x in non-E-format, without spaces, to the maximum decimal precision. The result may be used as a string function operand. The following is the format:

num1$ (x)

10.4.21 Ascii

The **ascii** function returns the ASCII value of the first character of the specified string. The format is:

ascii (s$)

The argument **s$** is a string variable or constant.

10.4.22 Rad$

The **rad$** function converts an integer in Radix-50 format to a string. This function is provided to maintain compatibility with DEC BASIC-PLUS. The format is as follows:

rad$ (i%)

10.4.23 Date$

The **date$** function returns the date **n** days from the current date. The format is as follows:

date$(n)

The argument **n** is an integer.

For example, if today is March 16, 1983, then:

date$(0) returns today's date, **'March 16, 1983'**

date$(-1) returns yesterday's date, **'March 15, 1983'**

date$(7) returns the date a week from today, **'March 23, 1983'**

10.4.24 Time$

The **time$** function returns the current time. The format is as follows:

time$(n)

The argument **n** is an integer.

If the time is 10:46, **time$(0)= '10:46:00'. Time$(n) when n<>0** gives the time **n** minutes after midnight.

10.5 Matrix Functions

BASIC provides five matrix-related functions defined in this section. A call to **trn, inv,** or **linsys** must begin with the keyword **mat**. Except for **trn**, these functions are approximations.

10.5.1 Trn

The **trn** function transposes a matrix, placing the value in a specified target matrix. When a matrix is transposed, the rows and columns are interchanged. If the target matrix is not the correct dimension, **trn** redimensions it. The format of the function is:

mat a = trn(b)

where **a** is the target matrix and **b** is the matrix to be transposed.

10.5.2 Inv

The **inv** function finds the pseudo-inverse of any matrix. The format is:

mat y = inv(a)

Use of **inv** is not recommended. See Appendix C, Linear Algebra, for more information.

10.5.3 Det

The **det** function returns the determinant of the matrix, the name of which appeared in the most recently executed **inv** function. The format is:

d = det

Use of **det** is not recommended. See Appendix C, Linear Algebra, for more information.

10.5.4 Linsys

The **linsys** function finds a matrix x such that

$x = a^{+}b$

where the relevant dimensions are:

a(n, p), x(p, m), b(n, m)

a^{+} is a pseudo-inverse, which equals the inverse when a is square and non-singular. In that case, **ax=b**.

The format is as follows:

```
mat x = linsys (a, b)
```

10.5.5 Cond

The **cond** function computes an estimate of the inverse of the condition number of the last matrix to be an argument to **inv** or **linsys**. If 1 + **cond** = 1 then the result is completely unreliable, and you may need to reformulate your problem. See Appendix C, Linear Algebra, for more information. The format is as follows:

c = **cond**

10.6 Creating Your Own Functions

You can add functions of your own creation to the ones that BASIC provides. The **def*** statement defines the name of the function and the tasks to be performed by the function.

Function names must always begin with **fn**, so that a function name is:

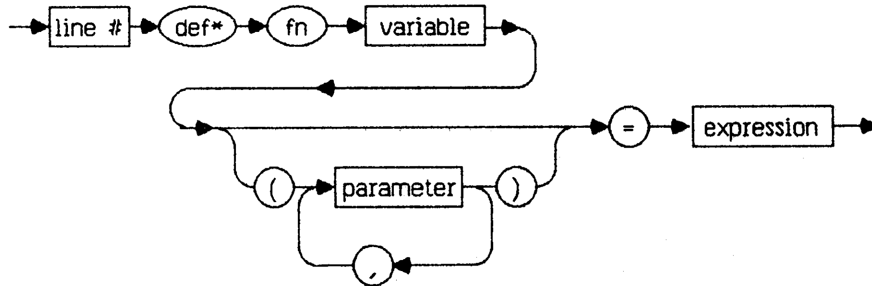
fn *variablename*

Variablename is a floating-point, integer, or string variable.

10.6.1 Def*

The **def*** statement is used to define functions. Functions may be defined as single-line functions. The syntax for a single-line function definition is:

*def**



Note: You may specify as many as five parameters.

The **expression** following the equal sign (=) specifies the operations to be performed and returned as the value of the function.

The statements up to the **fnend** statement comprise the operations to be performed and returned as the value of the function. During execution of multiple-line functions, the function is assigned expressions that comprise the operations to be performed. The value of the function is the result of the execution of the last expression.

It is illegal to nest function definitions.

10.6.2 Fnend

The **fnend** statement is used in a multiple-line function definition to signify the end of the function definition. The syntax for the **fnend** statement **is:**

fnend

The following are the definitions of two simple functions.

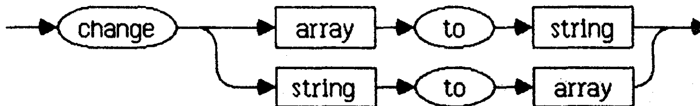
```
10      def* fntenx (a) = 10 * a
100     print fntenx (10)
```

```
3000   def* fnfactorial (n)
3010      if n<= 1 then fnfactorial = 1 else &
          fnfactorial=fnfactorial (n-1) * n
3020   fnend
```

10.7 Change

The **change** statement allows you to change a string into an array of numeric values, or change an array of numeric values into a string. In other words, you can change each character in a string to its ASCII value, and you can change an ASCII value to its corresponding character. The following is the syntax for the **change** statement.

change



The following example converts the characters in the string **B$** to their ASCII values (in the array **A%**), then converts them back to a string (**C$**).

```
10  dim A% (5)
20  read B$
30  data 'abcde'
40  change B$ to A%      !  convert string to array
40  for i=0 to A%(0)     !  A%(0) holds length of string
50     print A%(i)
60  next i
70  change A% to C$      !  convert back to a string
80  print C$
90  end
```

NOTES

Chapter 11
Block I/O, Open, and Close

11.1 Open 11-1

11.1.1 How the Open Statement Works 11-2
11.1.2 Open for ASCII I/O and Virtual Arrays 11-2
11.1.3 Open for Block 11-3

11.2 Close 11-3

11.3 Block I/O 11-4

11.3.1 Get and Put 11-4
11.3.2 Buffer Management 11-5
11.3.2.1 Field 11-5
11.3.2.2 Lset and Rset 11-6
11.3.3 Block I/O Sample Program 11-7

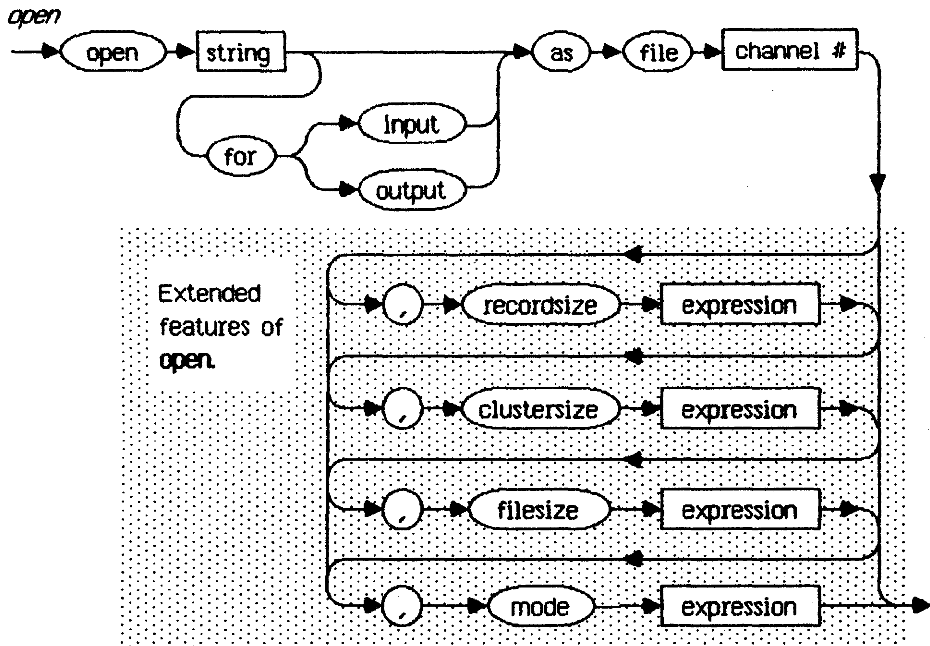
Block I/O, Open, and Close

Before you can read or write to a file or device, the file or device must be open. The console, which you have accessed using **input** and **print** statements, is always open; other files and devices must be opened and closed explicitly. The **open** and **close** statements are described below. This chapter also describes *block I/O*, an advanced programming technique for reading and writing files.

11.1 Open

The **open** statement assigns a logical input and output channel to a file or device. The device can be a *block-structured device* such as a disk or a *character device* such as the console or a printer. The syntax of the open statement is shown below; the extended options apply only to block I/O.

open



11.1.1 How the Open Statement Works

You open a file or device by giving it a name and a logical channel number in the **open** statement. In the following example, a file named "payroll" is associated with channel 4:

```
25 open 'payroll' as file #4%
```

The valid channel numbers are 1 through 12; channel 0 is reserved for the console and is always open. Up to 12 files can be open at any one time.

Whether you open a file for input or for output, you can perform both read and write operations in it. However, there are some important differences between these options; there is also a third option in which you do not specify input or output. The options work as follows:

- *Open for input* looks for an existing file with the name you specified in the **open** statement. If the file is not found, an error message is displayed on the console when the program is run; for example,

  ```
  50 open 'oldfile' for input as file #1
  ???Can't find file OLDFILE in line 50
  ```

  Here are some examples of **open for input** statements for a character device, a formatted ASCII file, and a virtual array:

  ```
  70 open '-keyboard' for input as file #1
  80 open '-upper-ledger.text' for input as file #2%
  90 open 'tax.array' for input as file #filnum%
  ```

- *Open for output* creates a new file with the name you specified in the **open** statement. If a file with that name already exists, it is deleted before the new file is created. Here are some examples of **open for output** statements:

  ```
  40 open '-printer' for output as file #3
  50 open '-upper-memo.text' for output as file #5%
  60 open 'employment.data' for output as file #x%
  ```

- *Unspecified open* tries to do an **open for input**. If the file is not found, a new file is created. Here are some examples of unspecified **open** statements:

  ```
  70 open '-upper-memo.text' as file #5%
  80 open 'tax.array' as file #x%
  ```

11.1.2 Open for ASCII I/O and Virtual Arrays

If you are doing formatted ASCII I/O, as described in Chapter 5, or if you are using virtual arrays, as described in Chapter 12, use the simple forms of the **open** statement shown above; the extended features apply only to block I/O.

11.1.3 Open for Block I/O

There are four extended options shown in the syntax diagram for the **open** statement: **recordsize, clustersize, filesize,** and **mode.** The **clustersize** and **mode** options are nonoperative; they have been kept to maintain compatibility with DEC BASIC-PLUS.

Recordsize determines the size of the buffer the system reserves when the file is opened. **Recordsize** must be a multiple of 512; if not, the value you specify is rounded up to the next multiple of 512. All files and devices available for BASIC on the Lisa have a default and minimum record size of 512 bytes.

Clustersize specifies the number of contiguous blocks to be allocated. In Lisa BASIC, **clustersize** is always 1. If you specify another value, it is ignored.

Filesize is an integer value that pre-extends the file to a designated number of blocks. The default **filesize** is 0. The system automatically extends the file block by block as you write records to it. You can also extend a file by several blocks at once, as in the following example.

```
50 open 'newfile' for output as file #2
60 put #2, block 200
```

Mode sets device-dependent properties. In Lisa BASIC, mode must be zero; if not, a run-time error is generated.

Bufsiz and **status** return information about an open file at run time. **Bufsiz** is an integer function that returns the buffer size of an open input/output channel. For instance,

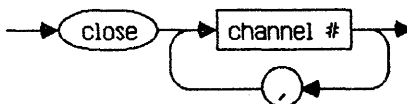
```
95 print bufsize(4%)
```

prints the buffer size for the file associated with channel 4. **Status** is a variable that contains information about the last channel that was opened. In Lisa BASIC, **status** is always 0; it exists to maintain compatibility with DEC BASIC-PLUS.

11.2 Close

The **close** statement closes the specified file, ends the association between a file or device and its input/output channel, and returns the buffer to the system. The following is the syntax for the **close** statement.

close



Channel # is any expression that results in the channel number (integer) associated with the file you want to close.

If your program leaves a file open, BASIC will close it when the workspace is cleared or when BASIC is exited. However, it is good programming practice to close all files in the program with a *positive channel number*. The system transfers the final information in the buffer to the output file only if you close it with a positive channel number.

When a *negative channel number* is used to close a file, the **close** statement returns the assigned buffer space but doesn't write the last buffer of information to the output file. The option of using a negative channel number is provided as an advanced technique for use with block I/O.

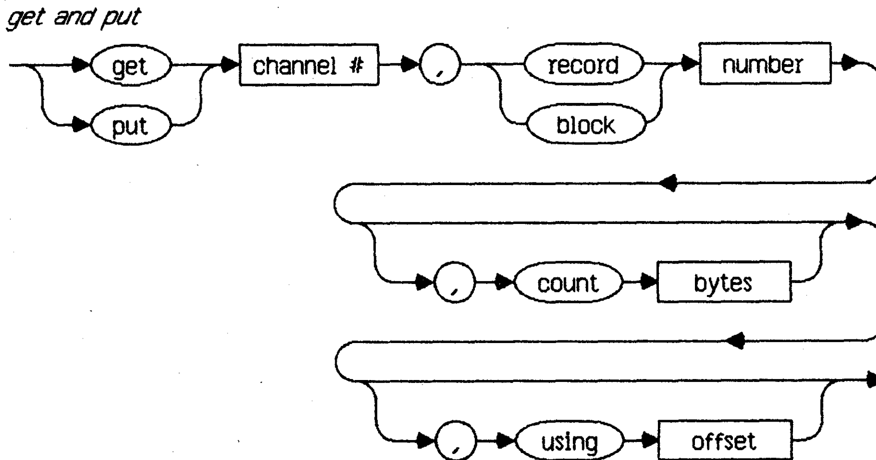
11.3 Block I/O

Block input and output is the reading and writing of file records, or blocks. Each record of a particular file is stored in a contiguous space on a disk. The space allotted to each record is the same length (usually 512 bytes). Block input and output permits both sequential and random file access.

11.3.1 Get and Put

The **get** statement reads blocks from a file into a buffer; the **put** statement writes blocks to a file from a buffer. The following is the syntax for both **get** and **put**:

get and put



An example of a **put** statement that writes the second half of a 512-byte buffer onto the first half of a 512-byte block on disk is

```
70 put #1, record 5, count 256, using 256
```

Before you use either the **get** or the **put** statement, the file or device you refer to in the statement must be open. The channel number you assign to the device in the **open** statement must be the same as the channel number you use in the **get** or **put** statement.

The **record** and **block** options specify a particular logical block of the file. **Number** is a numeric expression. **Block** doesn't restrict the value of **number**; **record** restricts the value to 32767. If the record or block options are not used, the next sequential block is written to or read from. If you attempt to access a block past the end of the file, a run-time error results.

The **count** option allows you to specify the number of bytes read from or written to the file or device. **Bytes** is an expression that represents the number of bytes read or written; it must have a positive integer value that does not exceed the size of the buffer. In a **put**, **count** specifies how many bytes are written; if **count** is omitted, **put** writes the entire buffer. In a **get**, **count** specifies the maximum number of characters to be read, ignoring the buffer size. If **count** is omitted, **get** fills the entire buffer. A **get** from a block device quits after reading the specified count. This means that succeeding data in the block, if any, will be lost; the next **get** will read from the next block. A **get** from a character device, however, terminates when the first <RETURN> is encountered, even if the count has not been exhausted. Use the **recount** variable with a character device; it tells how much of the buffer was read by the device.

Offset in the **using** clause is a numeric expression that specifies where the input and output operation starts. If **offset** is 10 in a **get** statement, for example, the system skips the first nine bytes of the buffer and begins reading into the buffer at the tenth byte. If **offset** is 10 in a **put** statement, the system skips the first nine bytes of the buffer and begins writing data to the file beginning with the tenth byte of the buffer.

11.3.2 Buffer Management

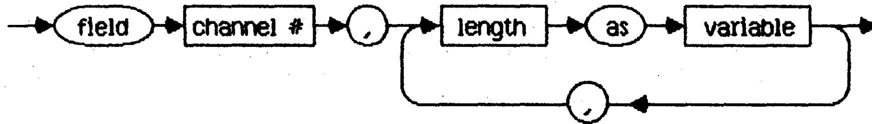
The **field**, **lset**, and **rset** statements manage the buffer that the system creates when a file is opened. The **field** statement associates string variables with specific bytes in the input/output buffer. The **lset** and **rset** statements assign values to these variables without moving them from the buffer.

11.3.2.1 Field

The **field** statement associates a section of an input and output buffer with a string variable.

The following is the syntax for the **field** statement:

field



Channel # must be the number of a channel already associated with a file. **Length** is a numeric expression specifying the number of bytes necessary for the **variable** in the **as** clause. The **field** statement allocates the first **length** bytes of the buffer to the first variable named, the second **length** bytes of the buffer to the second variable named, and so on.

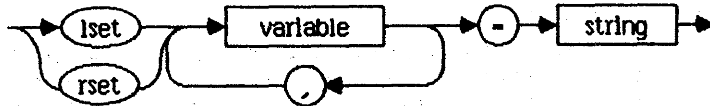
11.3.2.2 Lset and Rset

The **lset** and **rset** statements assign values to the string variables associated with a buffer in a **field** statement.

*Note: Always use the **lset** and **rset** statements to assign values to buffer variables when using block I/O. If the **let** statement is used to assign a value to a buffer variable, the variable is no longer associated with the buffer.*

The following is the syntax for both **lset** and **rset**.

lset and rset



For **lset** and **rset**, **variable** is any string variable assigned to a buffer in a **field** statement; **string** is any legal string value.

If the string value you put into the buffer is longer than the number of bytes allocated to the variable, **lset** and **rset** truncate the value--the length of the variable within the buffer is not enlarged. If the string value is shorter than the number of bytes allocated, the value in the buffer is padded with spaces. The **lset** statement left-justifies the string within the buffer; the **rset** statement right-justifies the string within the buffer.

11.3.3 Block I/O Sample Program

The following program demonstrates the use of block I/O.

```
100 open 'strange.data' for output as file 10
120 field #10, 350 as A$, 150 as B$, 12 as C$
130 lset A$ = 'left-justified string in the first 350' + &
     ' characters of the buffer'
140 rset B$ = 'right-justified string in the next 150' + &
     ' characters of the buffer'
150 lset C$ = '12characters'  !at end of buffer
160 put #10
170 close 10
180 end
```

NOTES

Chapter 12
Virtual Arrays

12.1 Dim Statement for Virtual Arrays 12-1
12.2 Virtual Array Storage 12-2
12.3 Virtual Array Access 12-3
12.4 File Length 12-4

Virtual Arrays

Virtual arrays allow you to store one or more matrices in a disk file and retrieve any element of any matrix in the file at random. You can define virtual arrays for floating-point, integer, or string variables. It is legal to have more than one type of array in a virtual array file. As with regular matrices, virtual array matrices allocate space for a zero'th element.

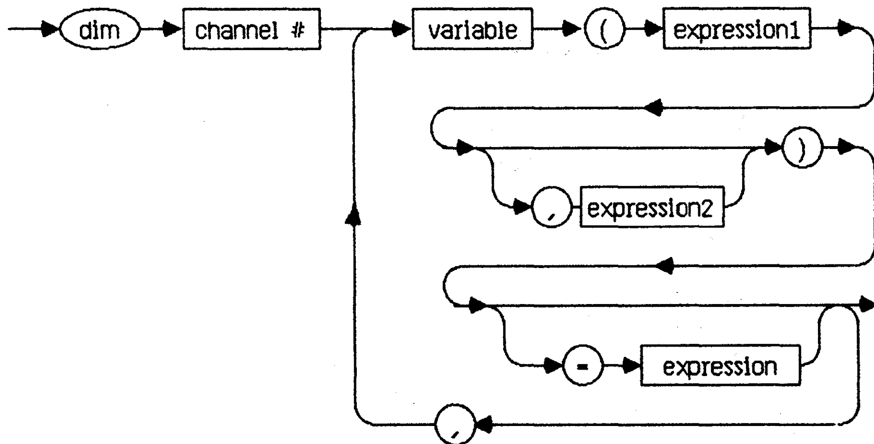
An array in a virtual array file can be larger than system memory. The part of the array that is not in use and does not fit into memory is kept on disk.

Opening and closing virtual array files is like opening and closing formatted ASCII files. If you don't explicitly close a virtual array file or if you close it with a negative channel number, any records remaining in the buffer are lost. Only when you close a virtual array file with a positive channel number does the **close** statement transfer any data remaining in the buffer to the file (see Section 11.2, Close).

12.1 Dim Statement for Virtual Arrays

A **dim** statement names the array or arrays contained in one virtual array file and, optionally, defines the size of each array element. The following is the syntax for the **dim** statment when it is used with virtual arrays.

dim (for virtual arrays)



The **dim** statement for a virtual array includes a mandatory input/output **channel #** and an optional record size specification for string arrays. Otherwise, it is the same as the **dim** statement for matrices that exist in

memory only during the execution of a program. The **dim** statement associates the channel number; then identifies the file in any subsequent input and output statements.

The optional clause '**= expression**' in the **dim** statement applies only to virtual string arrays. Unlike memory string array elements, there are restrictions on the size of string virtual array elements. All elements in a string virtual array have the same maximum length. This length, set by the program, must be a power of 2 in the range 2 to 512; the default is 16 characters. If you specify a number of characters that is not an acceptable power of 2, the system allocates the next highest power of 2 as the maximum size of the array elements.

Note that the **dim** statement allocates space in the file equal to the maximum length for each element in a string array even though any element can be shorter. For example, the statement

```
75 dim #1, prices (1000), items$ (1000) = 32%
```

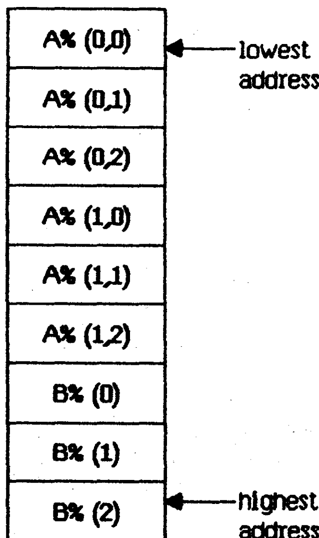
allocates space for an array of 1000 floating-point values and an array of 1000 string values that are each no more than 32 characters long.

12.2 Virtual Array Storage

Lisa BASIC stores arrays in a virtual file in the order named in the **dim** statement. The lowest address in the file corresponds to the first element in the first array. For example, the arrays defined in the **dim** statement below are stored as shown.

```
25 dim A% (1,2), B% (2)
```

Virtual Array Storage Chart



In the illustration on the previous page, the numbers in parentheses are the dimensions for a particular element, not its value.

There can be a gap in a virtual file that contains more than one array because no virtual array element can cross a disk block boundary. If one virtual array takes up more than one block, the blocks are not necessarily contiguous. The number of elements in one block of a virtual array file depends on the type of the array. Table 12-1 defines the number of array elements per block for each of the three data types.

Table 12-1

Number of Elements in a Virtual Array Block

Type of Element	Number
Integer	256
Floating-Point	64
String *(where x is the maximum element length)*	512/x

No virtual array element can be longer than one block or extend over a block boundary. This imposes no further restrictions on numeric values. However, this does limit virtual strings to 512 bytes.

As stated above, the system assigns each element in an array of virtual strings the same amount of storage space. The amount of storage is equal to some power of two bytes that results in an integer from 2 to 512 inclusive; 16 bytes is the default.

When the system assigns a value to a virtual string element, it is stored left-justified in its position within the file block. If the string is shorter than the maximum length, the system pads the rest of the space with null characters. When a program retrieves a padded string, the system strips the added null characters before returning the string.

12.3 Virtual Array Access

When a program asks for a virtual array element, the system first checks the buffer to see if the needed element is there. If it's there, the system passes it to the program. If it isn't there, the system updates records currently in the buffer and reads in successive pieces of the array until it finds the needed element.

To determine if a certain array element is currently in the buffer, the system must first convert the array subscript into a file address. This conversion requires two steps. First, the system computes the relative distance from the needed element to the first element in the array. This value depends on the array subscript and the number of array elements per block. Then, the system computes the distance from the first element of the array to the first element in the file. This value is a constant defined by the parameters in the **dim** statement that defined the file.

Because the **dim** statement in the program defines the structure of a virtual array file, it is possible to access arrays in a virtual file differently from different programs.

12.4 File Length

As with block input and output files, virtual array files are created with 0 length. As you add records to the file, the system automatically extends the length of the file to accommodate them.

We recommend that you extend the file to the maximum number of blocks you expect to use. The system overhead is the same whether you extend a file by one block or by many blocks. Therefore, we suggest the following technique to extend a new file in one operation:

```
10 dim #7, a(200)
20 for i = 200 to 0
30     a(i) = 1
40 next i
```

NOTES

Chapter 13
Advanced Floating-Point Manipulation

13.1 Exceptions 13-1

- 13.1.1 Invalid 13-1
- 13.1.2 Dividebyzero 13-1
- 13.1.3 Overflow 13-2
- 13.1.4 Cvtoverflow 13-2
- 13.1.5 Underflow 13-2
- 13.1.6 Inexact 13-2

13.2 Set Exception 13-2

13.3 Ask Exception 13-3

13.4 Set Halt 13-3

13.5 Ask Halt 13-3

13.6 Rounding Modes for Floating-Point Values 13-4

13.7 Set Rounding 13-5

13.8 Ask Rounding 13-5

13.9 Exception Handling and Rounding Examples 13-5

Advanced Floating-Point Manipulation

This chapter discusses advanced floating-point manipulation: floating-point exception handling and floating-point rounding.

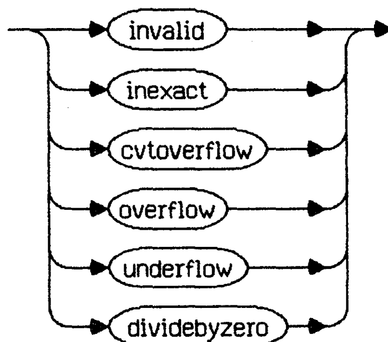
13.1 Exceptions

Certain arithmetic occurrences signal floating-point exceptions. The response of the system to these exceptions can be controlled with the halt flags. If the halt flag is set for an exception, the system will cause a run-time error if that exception occurs. Run-time errors halt program execution, unless an **on error** statement has been executed. If you don't want run-time errors to occur but are curious about the accuracy of your computation, you can test the exception flags at appropriate stages in the computation.

BASIC supports six exceptions: **invalid**, **dividebyzero**, **overflow**, **cvtoverflow**, **underflow**, and **inexact**. The **set exception**, **set halt**, **ask exception**, and **ask halt** statements allow you to set the halt and exception flags in different ways, causing different responses if any of these exceptions occur.

The following syntax diagram lists all the legal exception names. These exceptions are described later in this section.

exception name



13.1.1 Invalid

The **invalid** exception is signaled if an operand is invalid for the specified operation. The result is a NaN ("Not a Number").

13.1.2 Dividebyzero

The **dividebyzero** exception occurs when the divisor is zero and the dividend is a finite non-zero number. The result is positive or negative infinity.

13.1.3 Overflow

The **overflow** exception is signaled when the result of an operation is too large to represent.

13.1.4 Cvtoverflow

The conversion overflow exception, **cvtoverflow**, is signalled when a floating-point value is converted to an integer variable and the resulting integer is too large to represent.

13.1.5 Underflow

The **underflow** exception is signalled when the result of an operation is too small to represent accurately.

13.1.6 Inexact

The **inexact** exception is signalled if the result of an operation is not exact.

13.2 Set Exception

The **set exception** statement is used to set the exception flag to true or false. The following is the syntax for the statement:

set exception

Set exception, true or false, never causes a run-time error. When you set the exception flag to false, you can then test the flag after an operation is completed to see if the exception occurred. If the flag is true, the exception occurred and affected the result of the operation. If the exception flag is false, the operation proceeded normally. The following is an example which uses the exception flag, testing to see if the operation proceeded normally.

```
100 set exception cvtoverflow 0%  ! set flag false (0=false)
120 a% = b                        ! perform computation
130 ask exception cvtoverflow x%  ! obtain flag stutus in x%
140 if x% then print &
    "cvtoverflow exception occurred" ! test flag
```

13.3 Ask Exception

The **ask exception** statement interrogates an exception's flag. The following is the syntax for the **ask exception** statement.

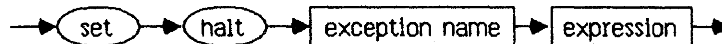
ask exception

Variable is set true (non-zero) or false (zero) according to the status of the exception flag. Each flag is set false by the **run** command, and may be set true during numeric computations. An exception flag may also be set true or false by **set exception**.

13.4 Set Halt

The **set halt** statement is used to set the halt flag. If you set the halt flag for an exception to true, then the system will generate a run-time error if the exception occurs. The following is the syntax for the statement:

set halt



13.5 Ask Halt

The **ask halt** statement interrogates an exception's halt flag. The following is the syntax for the **ask halt** statement.

ask halt

Variable is set true (non-zero) or false (zero) according to the status of the exception's halt flag. Each flag is set false by the **run** command. An exception's halt flag may be set true or false by the **set halt** statement.

13.6 Rounding Modes for Floating-Point Values

BASIC provides four rounding modes for floating-point values. These are illustrated for the intpart function, but they apply to all floating-point operations:

rnear Rounds to the nearest. If halfway, rounds to even.

```
intpart(2.7)  is 3.0
intpart(3.5)  is 4.0
intpart%(2.5) is 2
```

rpos Rounds toward positive infinity

```
intpart(3.5)  is 4.0
intpart(-3.5) is -3.0
```

rneg Rounds toward negative infinity

```
intpart(3.5)  is 3.0
intpart(-3.5) is -4.0
```

rzero Rounds toward zero

```
intpart(3.5)  is 3.0
intpart(-3.5) is -3.0
```

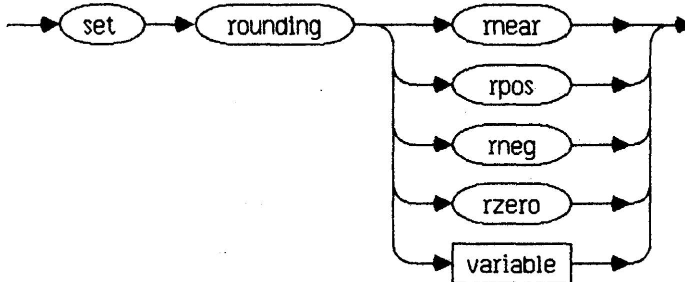
The **run** command sets the default rounding mode to **rnear**.

The rounding modes are set and tested using the **set rounding** and **ask rounding** statements.

13.7 Set Rounding

The **set rounding** statement sets the rounding mode to be used in all computations following execution of the statement. This rounding mode is in effect until another is specified. The following is the syntax:

set rounding



Variable is an integer variable.

The **run** command sets the rounding mode to **rnear.**

13.8 Ask Rounding

The **ask rounding** statement discloses the current rounding mode. The following is the syntax.

ask rounding

13.9 Exception Handling and Rounding Examples

The following example converts floating-point values to integer values with the rounding mode set to positive:

```
300  ask rounding oldr%      ! save old rounding mode
310  set rounding rpos       ! set desired rounding mode
320     x% = intpart%(x)     ! perform computation
330  set rounding oldr%      ! restore old rounding mode
```

The following example tests for a **cvtoverflow** exception:

```
100  ask rounding oldr%
120  set rounding rpos
130  ask exception cvtoverflow oldx%     !save old excep flag
140  set exception cvtoverflow 0%        !set flag false
150     x% = intpart%(x)                 ! perform computation
160  ask exception cvtoverflow newx%     ! obtain flag status
170     if newx% goto 900                ! test flag
180  set exception cvtoverflow oldx%     ! restore old flag
190  set rounding oldr%
```

The following example tests for a **cvtoverflow** exception using **halt**:

```
200  on error goto 1000             ! set up error handling
210  ask rounding oldr%
220  ask halt cvtoverflow oldh%     ! save old halt flag
230  set halt cvtoverflow 1%        ! set halt flag true
240  set rounding rpos
250     x% = intpart%(x)      ! cvtoverflow excep goes to 1000
260  set rounding oldr%
270  set halt cvtoverflow oldh%     ! restore old halt flag
        .
        .
        .
1000  print "cvtoverflow"          ! error handling
```

NOTES

Chapter 14
System Statements

14.1 Wait 14-1
14.2 Sleep 14-1
14.3 Writeprotect 14-1
14.4 Writeallow 14-1
14.5 Unlock 14-2
14.6 Chain 14-2
14.7 Name As 14-3
14.8 Kill 14-3

System Statements

This chapter describes the system statements. The system statements affect the BASIC programming environment from within a program.

14.1 Wait

The **wait** statement instructs the system to wait a specified number of seconds for **input** from the console before issuing a run-time error. **Wait 0** sets the system so that it will wait indefinitely for input. The syntax is as follows:

wait

14.2 Sleep

The **sleep** statement instructs the system to pause for a specified number of seconds. The following is the syntax.

sleep

14.3 Writeprotect

The **writeprotect** statement sets file safety so that all files associated with the channel specified by **expression** cannot be overwritten. The following is the syntax.

writeprotect

14.4 Writeallow

The **writeallow** statement removes file safety so that all files associated with the channel specified by **expression** can be overwritten. The following is the syntax.

writeallow

14.5 Unlock

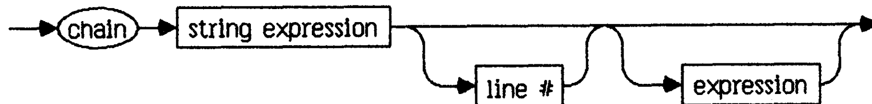
The **unlock** statement is the same as the **writeallow** statement; it removes file safety so that all files associated with the channel specified by **expression** can be overwritten. This statement is provided to maintain compatibility with DEC BASIC-PLUS. The following is the syntax.

unlock

14.6 Chain

The **chain** statement allows you to "chain" more than one program together when a program is too large to fit in the workspace. By splitting the large program into small independent programs and using the **chain** statement to start execution of the next program, you can achieve the effect of a very large program. The syntax for the **chain** statement is as follows:

chain



Expression is an entry point (line number) in the program where you wish execution to begin. If no entry point is specified, execution begins at the lowest line number.

When the **chain** statement is executed, BASIC loads the program specified by **string expression** into the workspace and begins execution. This means that the program specified in the **chain** command *completely* replaces the original program in the workspace. **Chain** closes all open files upon execution. We recommend, however, that you explicity close all open files with the **close** statement before the **chain** statement is executed. Information in buffers may be lost--make sure that the program you chain to is complete. If the new program uses the same files, you must explicitly **open** these files. All variables are re-initialized in the new program (numeric values set to zero, strings set to the null string).

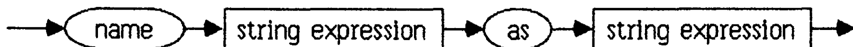
The following is a very simple example of chaining.

```
old SecondPart
Ready
list
10 rem This is the second half of the program
20 print "This is line 20 of SecondPart"
30 end

Ready
new FirstPart
Ready
10 rem This is the first half of the program
20 print "This is line 20 of FirstPart"
30 chain 'SecondPart' line 10

runnh
This is line 20 of FirstPart
This is line 20 of SecondPart
```

14.7 Name As

The **name as** statement renames a file to another specified filename. The following is the syntax.

name as



14.8 Kill

The **kill** statement removes the specified file from the directory. The following is the syntax.

kill

NOTES

Appendixes

A Language Summary A-1
B Floating-Point Arithmetic B-1
C Linear Algebra C-1
D Error Messages D-1
E BASIC Workshop Files E-1

Appendix A
Language Summary

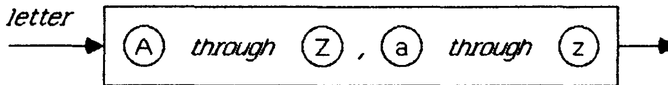
A.1 The BASIC Character Set A-1
A.2 Operators A-2
A.3 Syntax Diagrams A-3
A.4 Reserved Words A-22

Language Summary

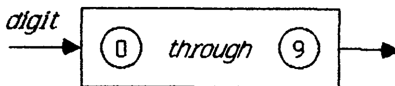
This appendix summarizes the BASIC language for the Lisa for quick reference. Included are the BASIC character set, operator tables, all syntax diagrams for all commands (listed alphabetically), and a list of reserved words.

A.1 The BASIC Character Set

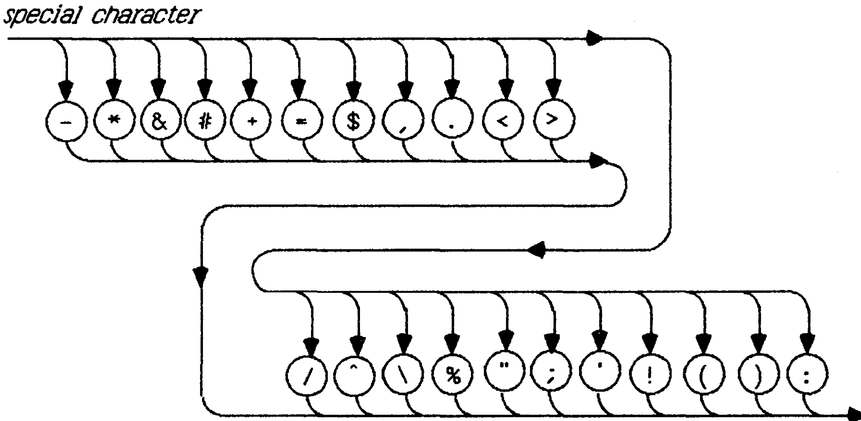
A *letter* is one of the following:



A *digit* is one of the following:



A *special character* is one of the following:



A.2 Operators

Arithmetic Operators	
+	addition
-	subtraction
*	multiplication
/	division
^	exponentiation
**	exponentiation
remainder	remainder

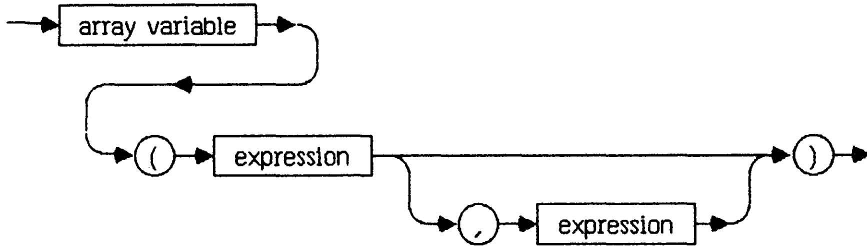
Relational Operators	
=	equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
<>	not equal to
==	approximately equal to

Logical Operators	
and	conjunction
or	disjunction
eqv	equivalence
not	negation
imp	implication
xor	exclusive or

A.3 Syntax Diagrams

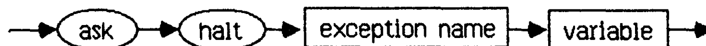
append

array selection



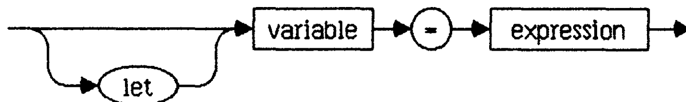
ask exception

ask halt

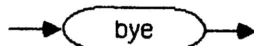


ask rounding

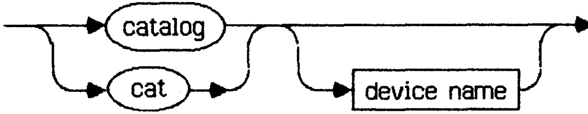
assignment



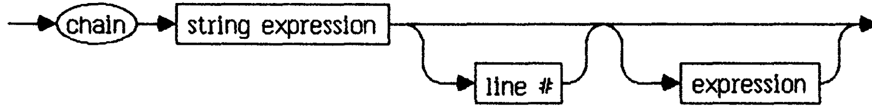
bye



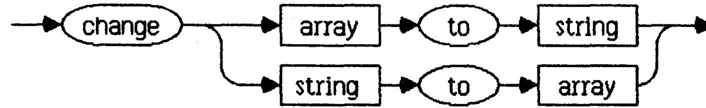
catalog



chain

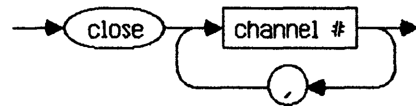


change



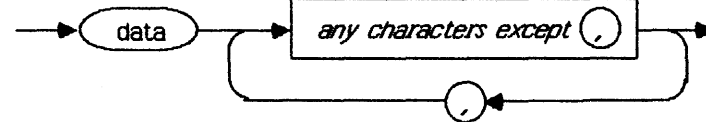
channel #

close

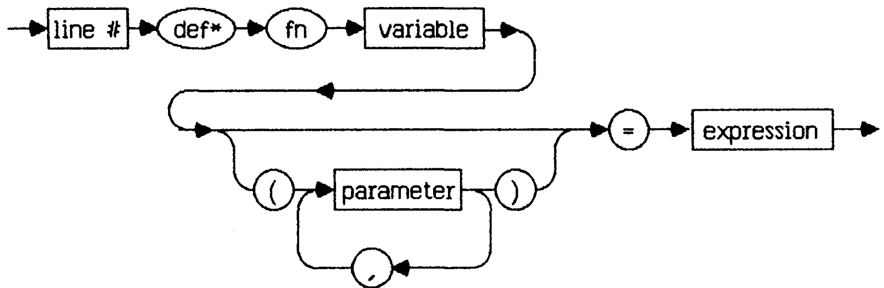


cont

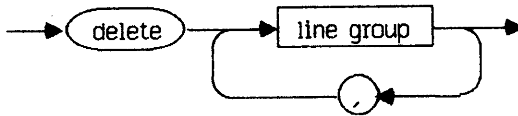
data



*def**

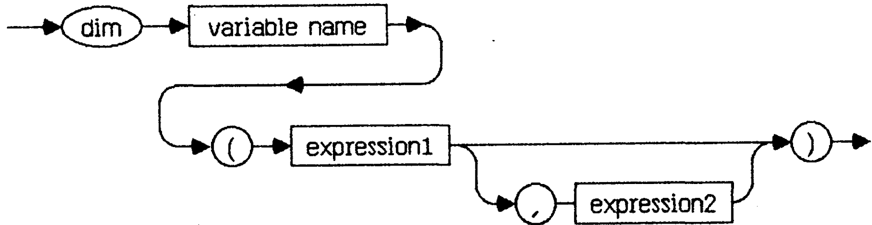


delete

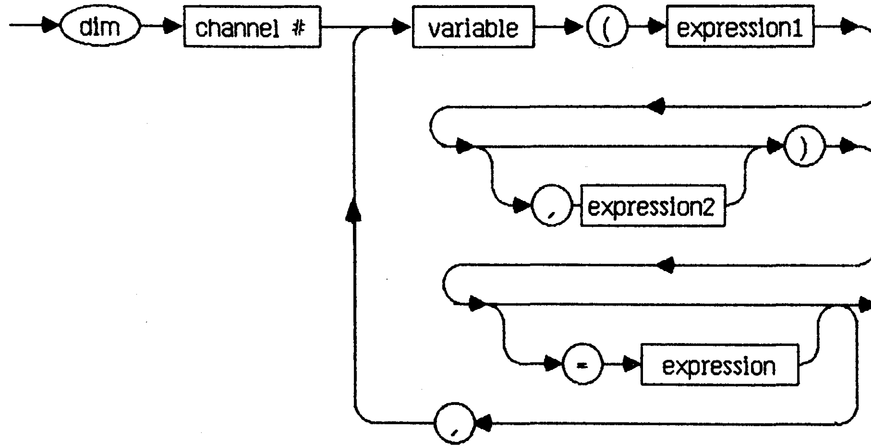


digit

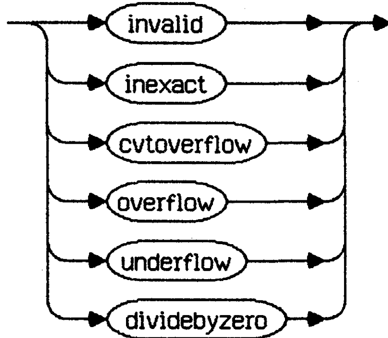
dim



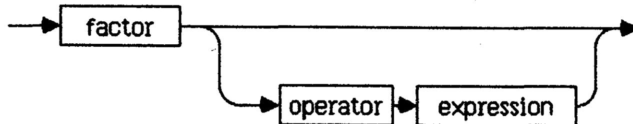
dim (for virtual arrays)



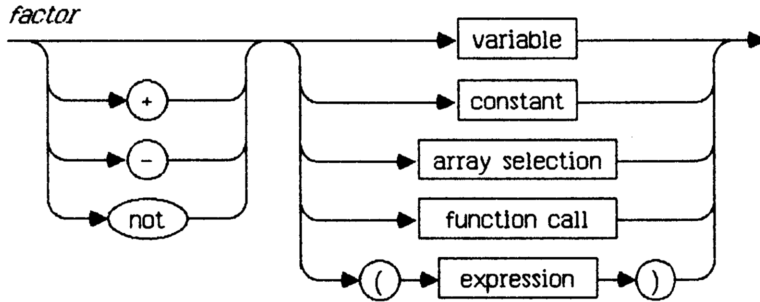
exception name



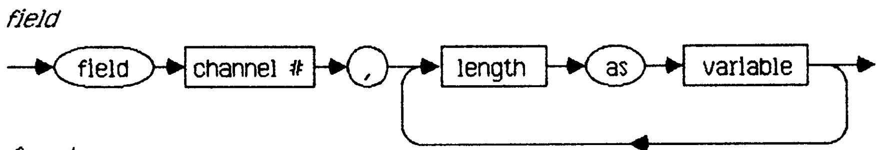
expression



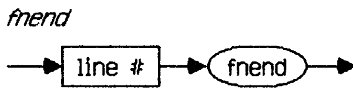
factor



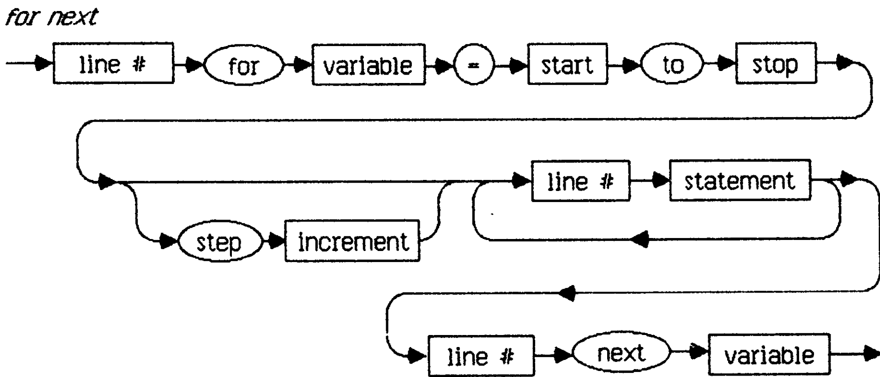
field



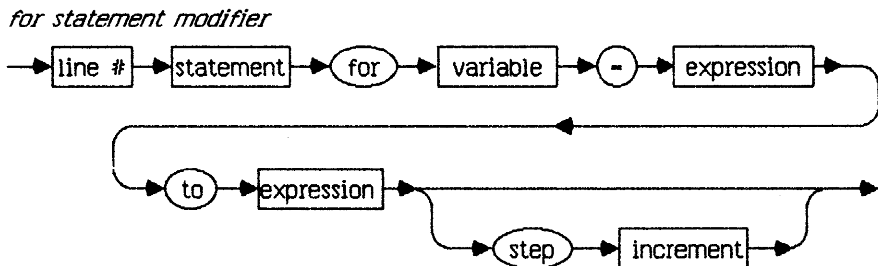
fnend



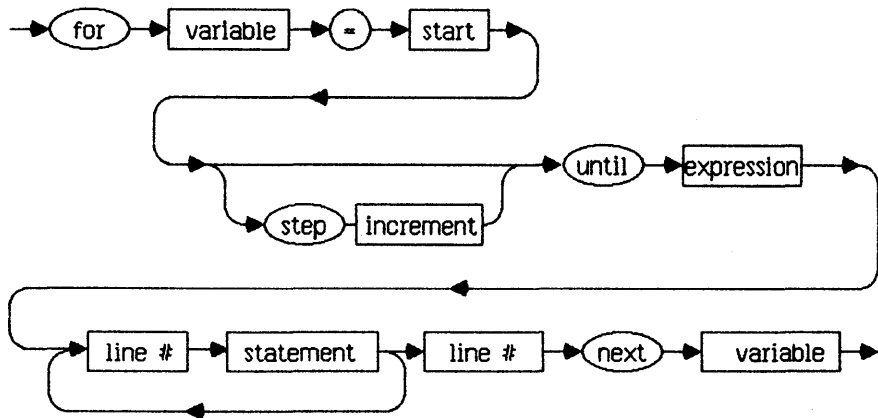
for next



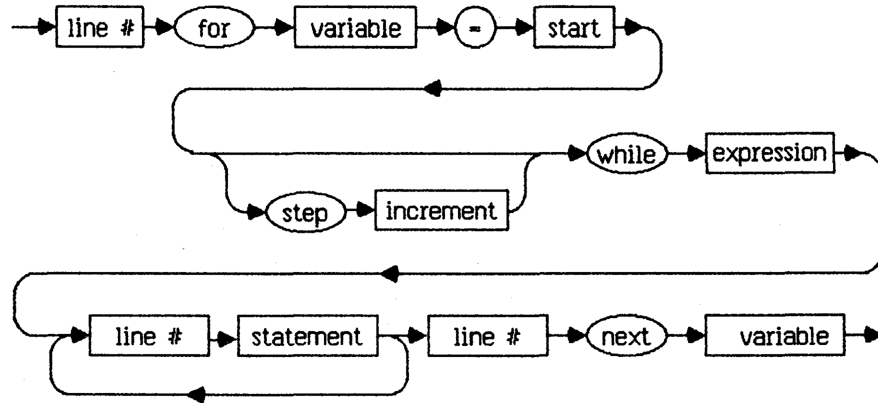
for statement modifier



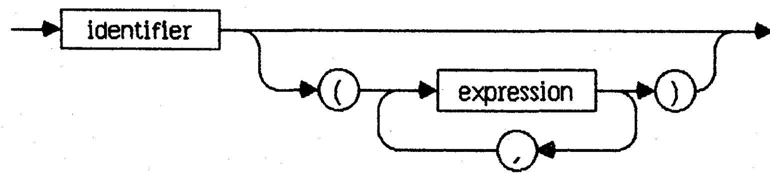
for until



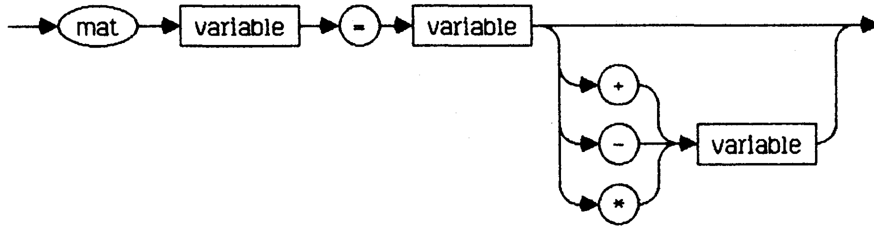
for while



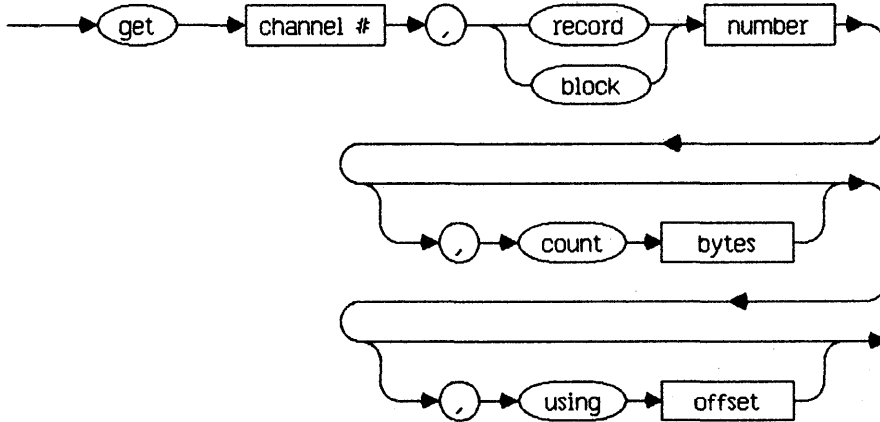
function call



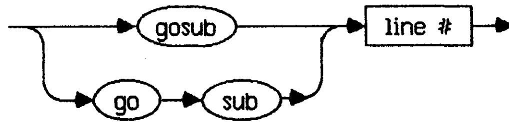
general matrix arithmetic



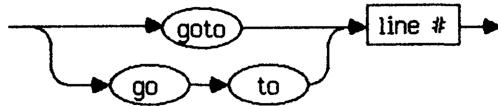
get



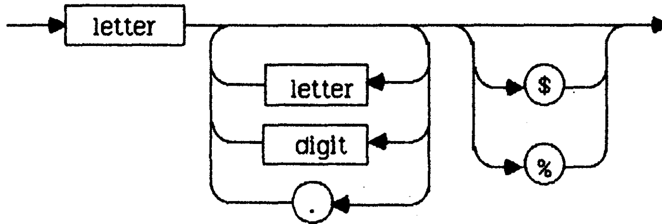
gosub



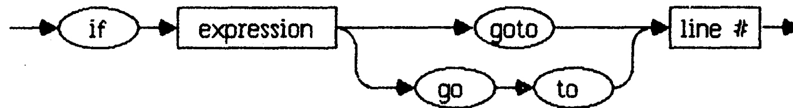
goto



identifier name

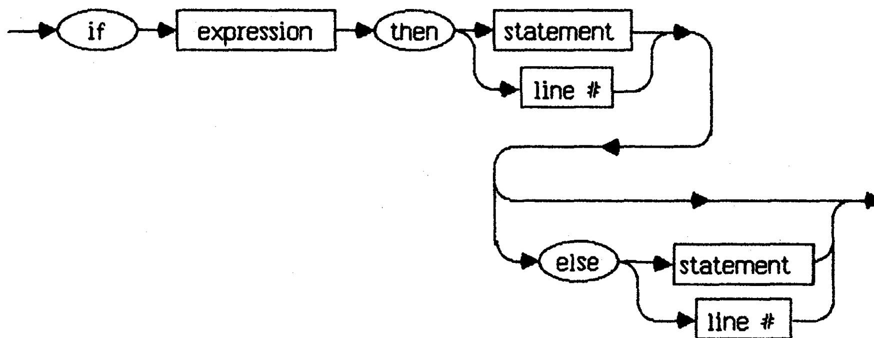


if goto

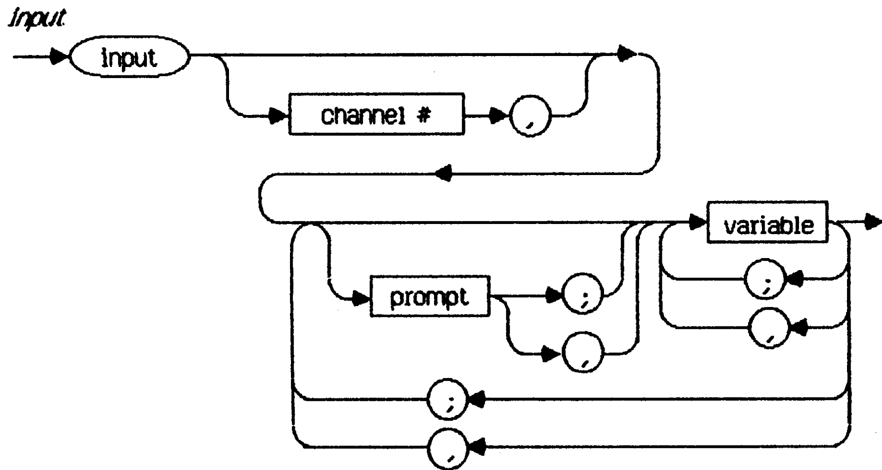


if statement modifier

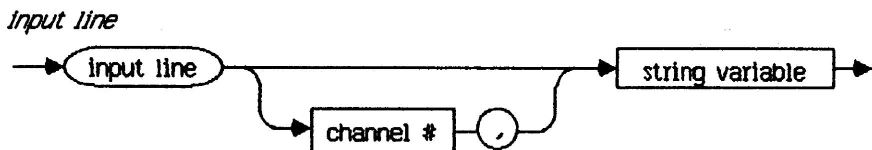
if then else



input



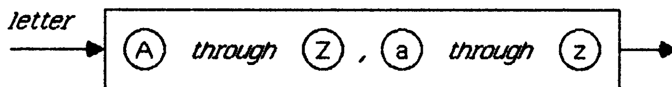
input line



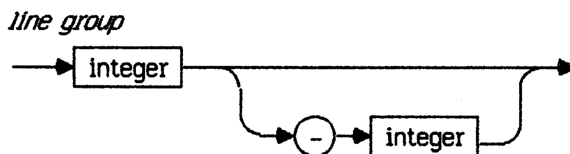
kill

length

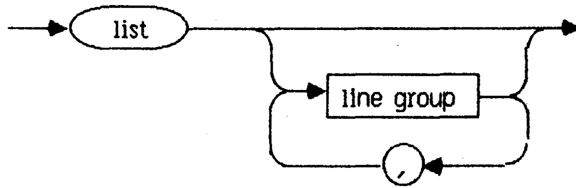
letter



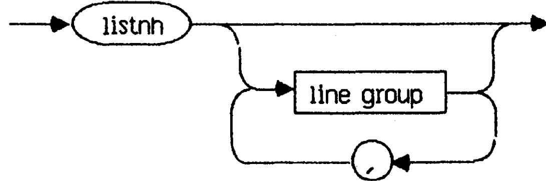
line group



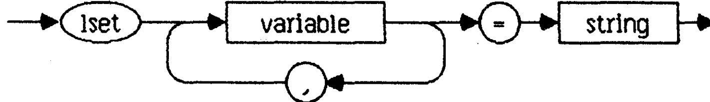
list



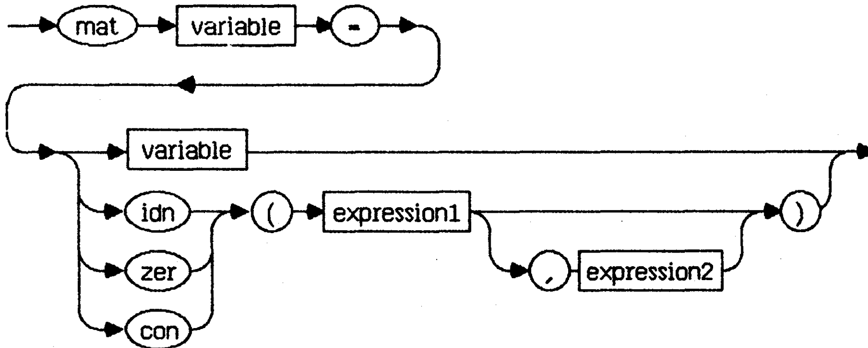
listnh



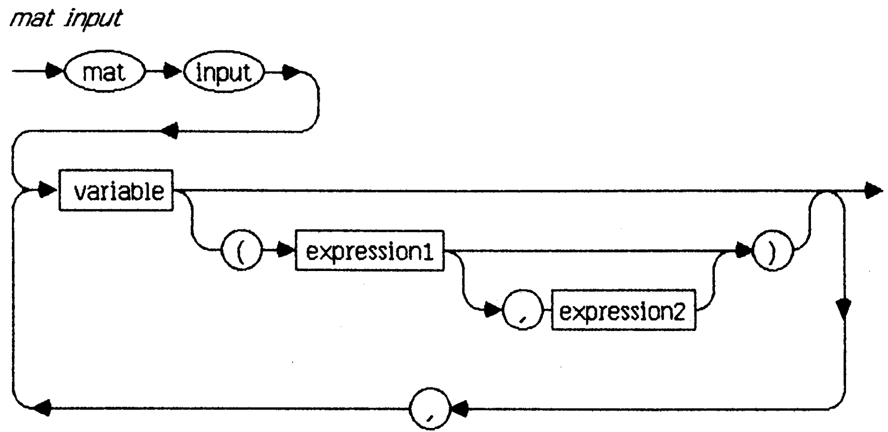
lset



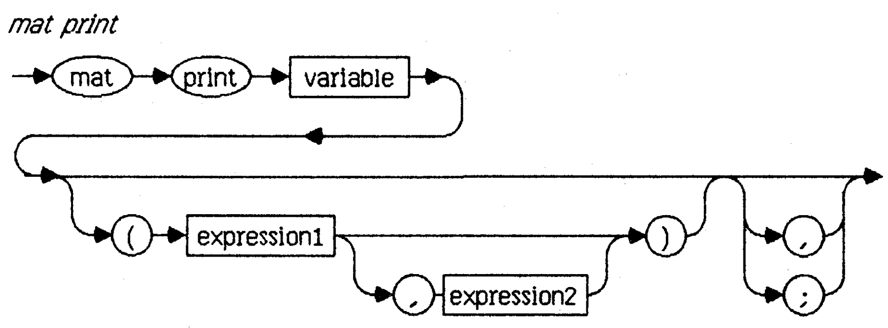
mat



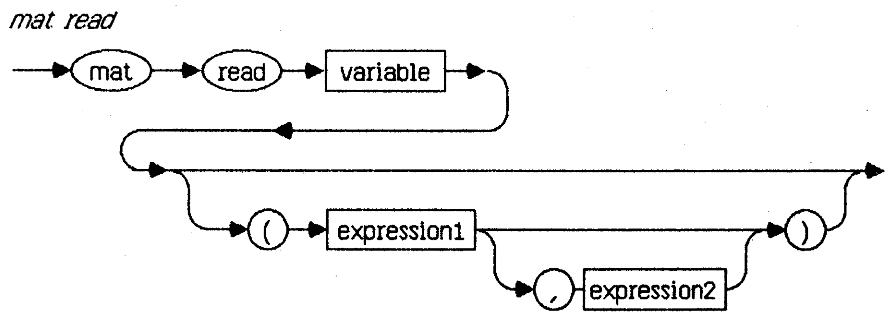
mat input



mat print

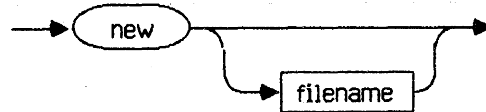


mat read

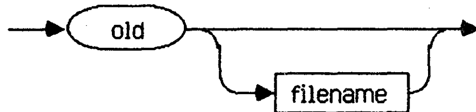


name as

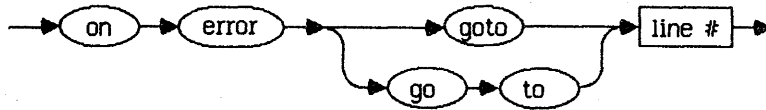
new



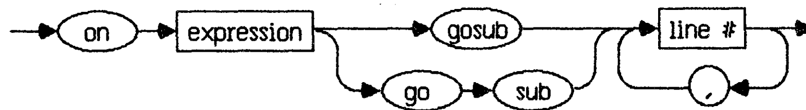
old



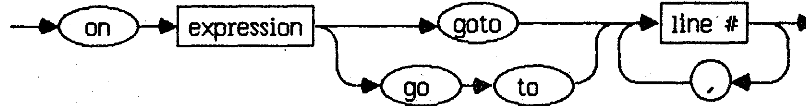
on error goto

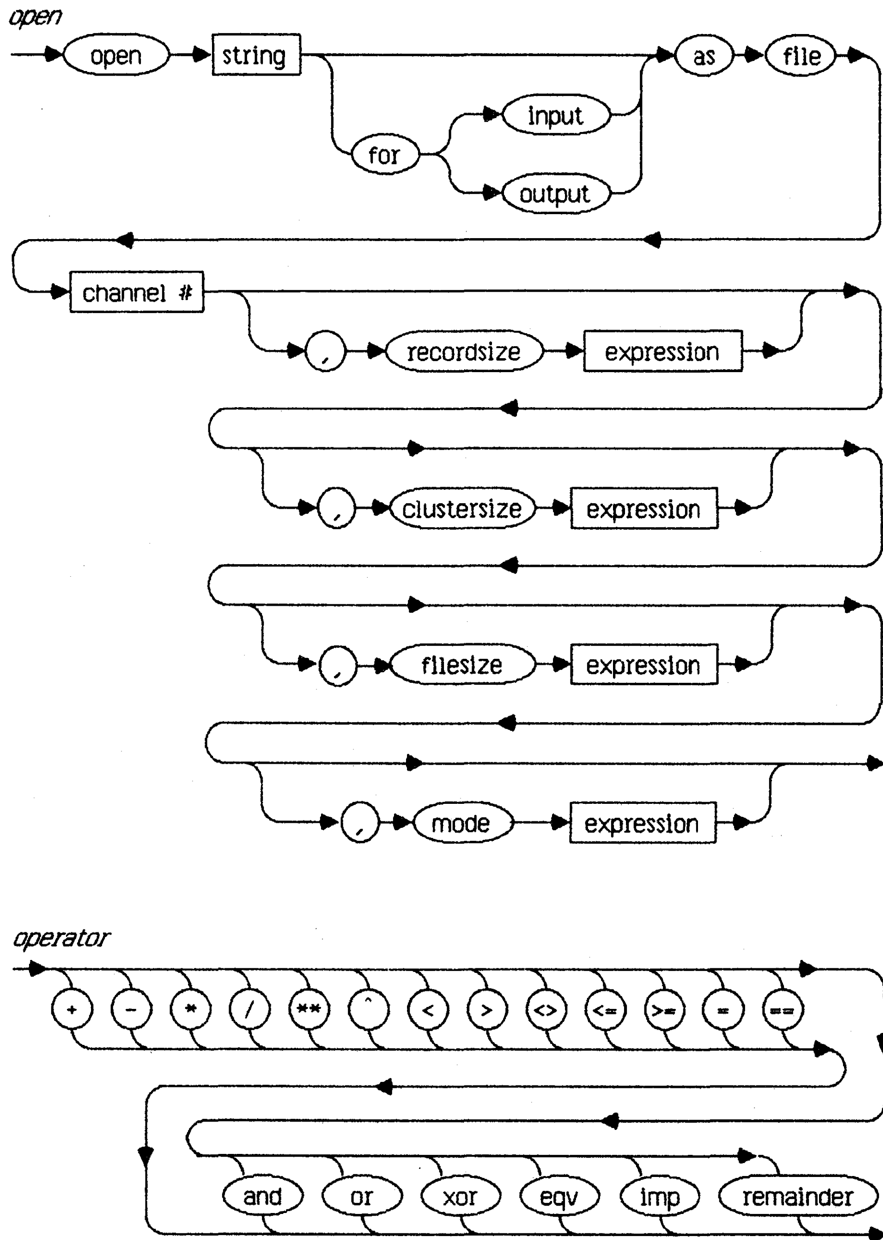


on gosub

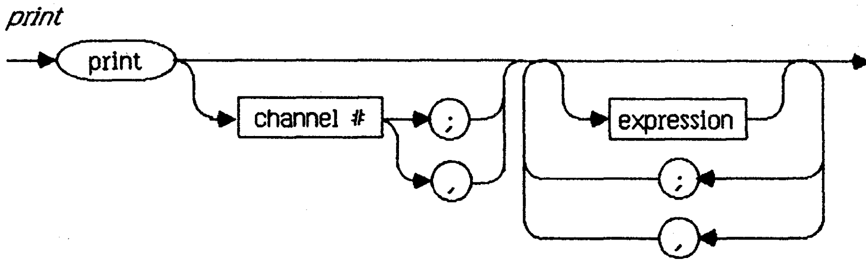


on goto

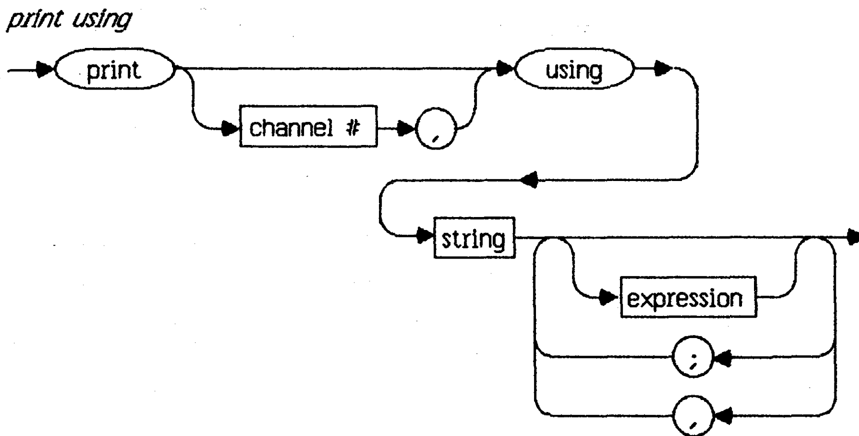



open
open
string
for
input
output
as
file
channel #
,
recordsize
expression
,
clustersize
expression
,
filesize
expression
,
mode
expression
operator
+
-
*
/
**
^
<
>
<>
<=
>=
=
==
and
or
xor
eqv
imp
remainder

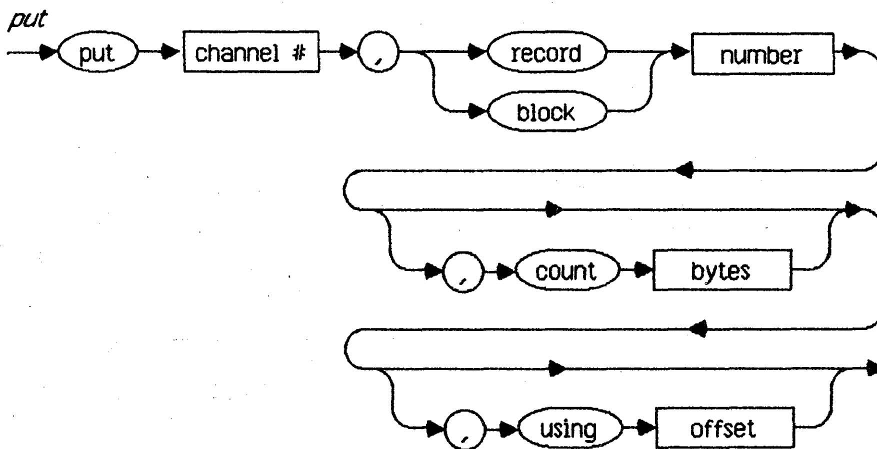
print



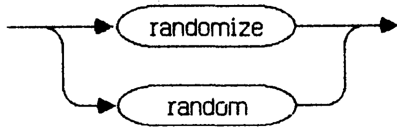
print using



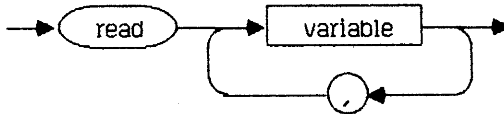
put



randomize

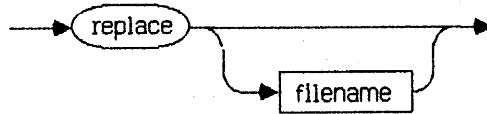


read



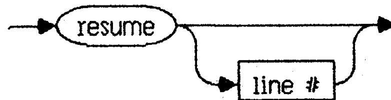
renumber

replace

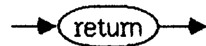


restore

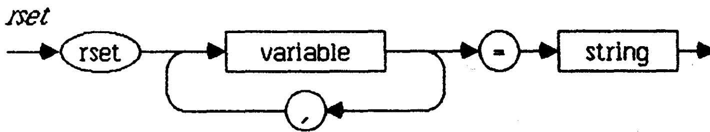
resume



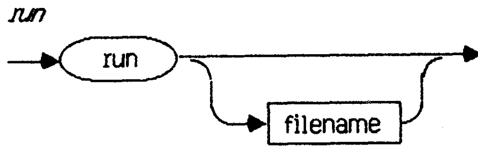
return



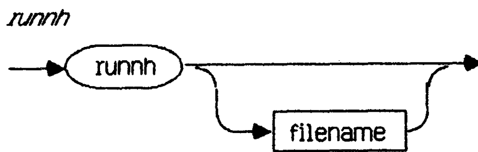
rset



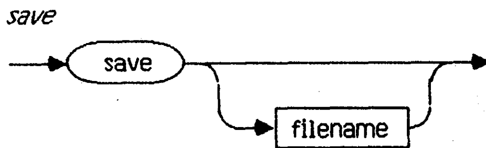
run



runnh

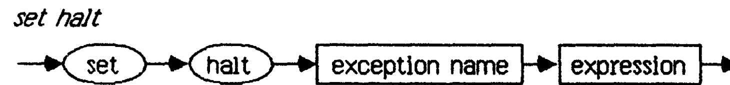


save

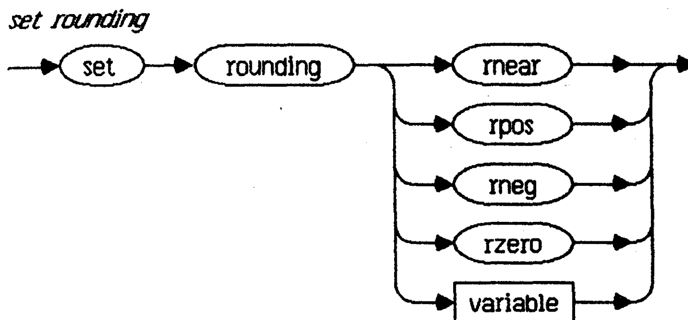


set exception

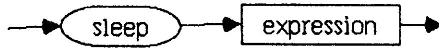
set halt



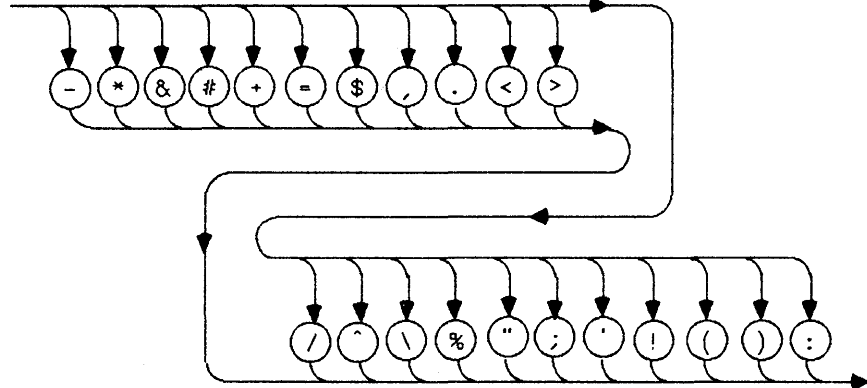
set rounding



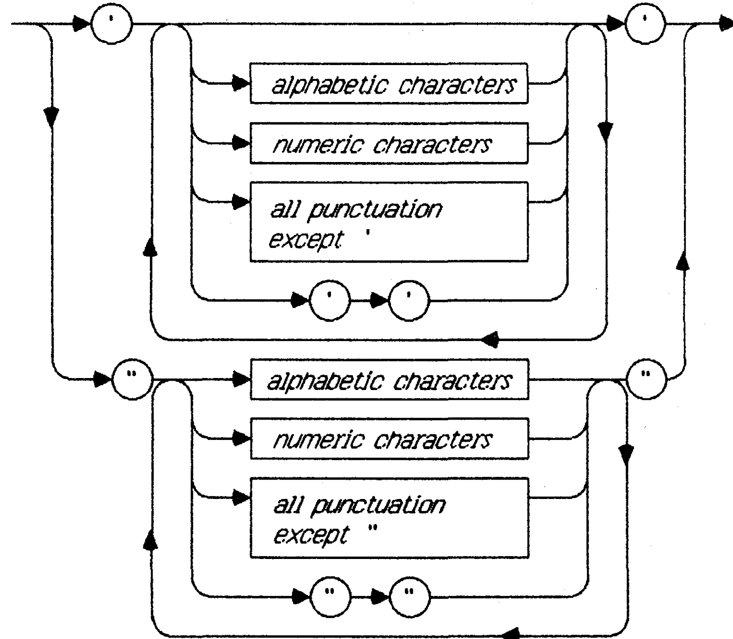
sleep



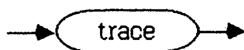
special character



string constant



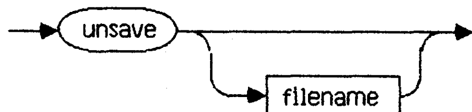
trace



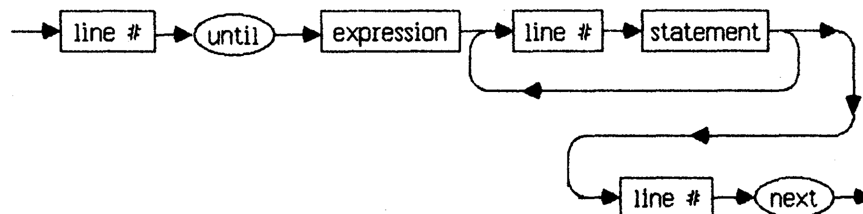
unless statement modifier

unlock

unsave

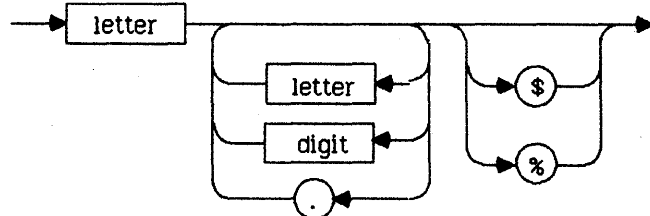


until next



until statement modifier

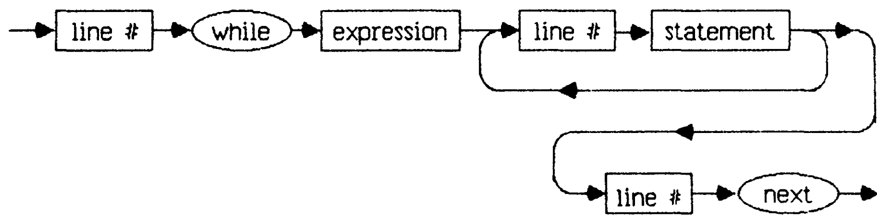
variable name



variables

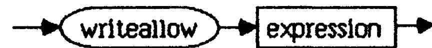
wait

while next

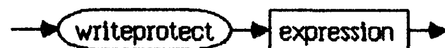


while statement modifier

writeallow



writeprotect



A.4 Reserved Words

BASIC Reserved Words					
abs	close	dif$	find*	intpart%	nochanges*
access*	clustersize*	dim	fix	inv	noduplicates*
allow*	com*	dimension*	fixed*	invalid	noecho*
alternate*	common*	dividebyzero	fnend	key*	none*
and	comp%	duplicates*	fnexit*	kill	norewind*
annuity	compound	echo*	for	left	nospan*
append	con	edit$*	format$*	left$*	not
as	cond	else	from*	len	nul$*
ascii	connect*	end	fsp$*	let	num
ask	contiguous*	eq*	fss$*	lf*	num$
atn	cos	eqv	ge*	line	num1$
back*	count*	erl	get	linput*	num2
bel*	cr*	ern$*	go*	linsys	on
block	ctrlc*	err	gosub	loc*	onechr*
blocksize*	cvt$$	error	goto	log	onerror
bs*	cvt$%	ert$*	gt*	log10	open
bucketsize*	cvt$f	esc*	halt	lset	or
buffer*	cvt%$	exception	ht*	magtape*	organization*
bufsize	cvtf$	exp	idn	map*	output
by*	cvtoverflow	extend*	if	mat	overflow
call*	data	ff*	imp	mid	peek*
ccpos	date$	field	indexed*	mid$*	pi
chain	def	file*	inexact	mode*	place$
change	delete	filesize*	input	modify*	pos
changes*	density*	fill*	instr	move*	primary*
chr$	desc*	fill$*	int	name	print
	det	fill%*	intpart	next	prod$

* Nonoperative reserved words.

BASIC Reserved Words, continued					
put	reset*	seg%*	stop	then	val%*
quo$	restore	sequential*	str$*	time	value*
rad$	resume	set	stream*	time$	variable*
random	return	sgn	string$	to	virtual*
randomize	right	si*	sub	trm$*	vt*
rctrlc*	right$*	sin	subend*	trn	wait
rctrlo*	rnd	sleep	subexit*	undefined*	while
read	rnear	so*	sum$	underflow	windowsize*
record*	rneg	sp*	swap%	unless	write*
recordsize	rounding	space$	sys*	unlock	writeallow
recount	rout*	span*	tab	until	writeprotect
ref*	rpos	spec*	tan	update*	wrkmap*
relative*	rset	sqr	tape*	useropen*	xlate
rem	rzero	status	task*	using	xor
remainder	scratch*	step	temporary*	val	zer

* Nonoperative reserved words.

Appendix B
Floating-Point Arithmetic

B.1 Introduction B-1
B.2 Rounding of Floating-Point Results B-1
B.3 Accuracy of Arithmetic Operations B-1
B.4 Overflow and Division by Zero: Infinite Values B-1
B.5 Invalid Operations and NaN Values B-3
B.6 Integer-Conversion Overflow B-3
B.7 Text-Oriented I/O Conversions B-3
B.8 Bibliography B-4

Floating-Point Arithmetic

B.1 Introduction

BASIC floating-point arithmetic (all arithmetic involving **floating-point** values) conforms to the IEEE's *Proposed Standard for Binary Floating-Point Arithmetic* (Draft 10.0 of IEEE Task P754), except that the only supported precision for floating-point values is double.

IEEE Standard arithmetic provides better accuracy than many other floating-point implementations. It also reduces the problems of overflow, underflow, limited precision, and invalid operations by providing useful ways of dealing with them.

As a general rule, you can write BASIC programs that use floating-point arithmetic without worrying about the differences between IEEE Standard arithmetic and other floating-point implementations.

The following points apply if your program writes out floating-point numbers as textual representations (via **print** or **print using**):

- Anything in the output that looks like a number will be correct (and possibly more accurate than under other implementations).
- If your output contains a string of two or more pluses or minuses, this indicates a value of ∞, resulting from division by zero or an operation that caused a floating-point overflow.
- If your output contains the string *NaN* ("Not a Number"), it indicates the result of an invalid operation that would probably have caused a program halt or a wrong output under other implementations.

B.2 Rounding of Floating-Point Results

When a **floating-point** result must be rounded, it is rounded by default to the nearest representable **floating-point** value. If the unrounded result is exactly halfway between two representable **floating-point** values, it is rounded to the value that has a zero in the least significant digit of its binary fraction (the "even" value).

B.3 Accuracy of Arithmetic Operations

The arithmetic operations +, –, *, /, and **sqr** suffer at most one rounding error. **Remainder** is computed exactly.

B.4 Overflow and Division by Zero: Infinite Values

The result of floating-point overflow is either +∞ or –∞. These are floating-point values that can be used in further calculations and are mathematically well-behaved: for example, a finite number divided by ∞ yields zero.

The default treatment for dividing a finite non-zero value by zero is to yield +∞ or –∞ without a run-time error.

Infinite values have textual representations that can be read by **read** or written out by **print** or **print using**, and input by **input a$** where **a = val(a$)**.

Tables B-1 and B-2 below show the results of arithmetic operations on infinities. Note that any operation involving a NaN as an operand produces a NaN as the result.

Table B-1
Results of Addition and Subtraction on Infinities

Left Operand		*Right Operand* −∞	finite	+∞
−∞	+	−∞	−∞	NaN
finite	+	−∞	finite†	+∞
+∞	+	NaN	+∞	+∞
−∞	−	NaN	−∞	−∞
finite	−	+∞	finite†	−∞
+∞	−	+∞	+∞	NaN

† Result may be an infinity if the operation overflows, depending on the rounding mode.

Table B-2
Results of Multiplication and Division on Infinities

Left Operand		*Right Operand* ±0	finite	±∞
±0	*	±0	±0	NaN
finite	*	±0	finite†	±∞
±∞	*	NaN	±∞	±∞
±0	/	NaN	±0	±0
finite	/	±∞	finite†	±0
±∞	/	±∞	±∞	NaN

† Result may be an infinity if the operation overflows, depending on the rounding mode.

Note: Sign of result is determined by the usual mathematical rules.

B.5 Invalid Operations and NaN Values

The following operations are considered to be invalid:

- $\infty - \infty$ or $\infty + (-\infty)$
- $0 * \pm\infty$
- $0/0$ or $\pm\infty/\pm\infty$
- x **remainder** y, where y is zero or x is infinite.
- square root of an operand less than zero.
- † conversion of a NaN to an integer variable.
- † comparisons other than = and <> involving NaNs.

The default treatment for such an operation is the following:

1. Set the invalid floating-point exception flag to true.
2. Provide a result: if the operation would provide a floating-point result for valid operands, then the floating-point result for invalid operands is a NaN. In the two operations marked †, the result is unspecified and false, respectively.
3. Continue execution.

A NaN resulting from an invalid operation propogates: if used as an operand in another operation, the result will be the same NaN. NaNs can be written out via **print** or **print using**, and read in via **read**: the textual representation is "NaN", which may be followed by a quoted string.

B.6 Integer-Conversion Overflow

Integer-conversion overflow can occur if too large a floating-point value is assigned to an integer variable. The result returned is unspecified.

B.7 Text-Oriented I/O Conversions

The **input**, **print**, and **print using** statements convert numbers from decimal to binary on input and from binary to decimal on output. The error in these conversions is less than 1 unit of the result's least significant digit.

Floating-point values appear as character strings in two different contexts: within BASIC statements, and as data in files. The signed-number syntax of Chapter 4 applies in both cases.

The output textual representation of a floating-point value is rounded to the nearest possible decimal representation. If the unrounded value is exactly halfway between two possible representations, the representation whose least significant digit is even is written out.

For **read** , **print** and **print using**, $+\infty$ is represented by a string of at least two plus signs, and $-\infty$ by a string of at least two minus signs. NaNs are

represented by the characters "NaN", with an optional leading sign, and an optional trailing quoted string of characters, as follows:

-NaN'4'

The character string provides diagnostic data.

B.8 Bibliography

The following articles contain detailed information and discussion of the proposed IEEE floating-point standard. (Articles are listed in order of importance.)

- "A Proposed Standard for Binary Floating-Point Arithmetic", *IEEE Computer*, Vol. 14, No. 3, March 1981.
- Coonen, J.: "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic", *IEEE Computer*, Vol. 13, No. 1, January 1980.
- ACM SIGNUM Newsletter, special issue devoted to the proposed IEEE floating-point standard, October 1979. In particular, see article by Kahan and Palmer.
- Coonen, J.: "Underflow and the Denormalized Numbers", *IEEE Computer*, Vol. 14, No. 3, March 1981.

Appendix C
Linear Algebra

C.1 Vectors and Linear Transformations C-1
C.2 Singular and Nonsingular Linear Transformations C-1
C.3 Inverses of Linear Transformations C-2
C.4 Transformations between Spaces of Different Dimension C-2
C.5 Arrays and Matrices C-2
C.6 Computing the Results of Transformations C-4
C.7 Inverse Transformations C-5
C.8 Solving Linear Equations C-5
C.9 Accuracy C-7
C.9.1 Condition Numbers C-7
C.9.2 Ill-Conditioned Problems C-8
C.10 Determinants C-9
C.11 Linear Least-Squares Problems C-9
C.12 Expert's Corner C-10

Linear Algebra

This appendix describes the matrix algebra facilities in BASIC. Matrix algebra includes the operations of matrix multiplication, solving linear equations, and solving linear least-squares problems. BASIC provides the **mat** keywords *****, **linsys, cond, inv,** and **det** that perform these operations.

Because BASIC for the Lisa provides more general and powerful matrix operations than most other BASICs, it is helpful to review the theoretical setting that underlies matrix algebra.

C.1 Vectors and Linear Transformations

Linear algebra is concerned with elements in vector spaces and the class of linear transformations upon them. Does this sound too abstract? Then think of this concrete example: the vector space is the set of points in a window on the screen, forming a picture. One point, the origin, is special; often it is the lower left corner. Certain collections of these points make lines, and by lines we mean straight lines. Linear transformations are transformations of the points in the window that preserve linear relationships; that is, they map lines into lines. Typical linear transformations include the identity transformation, which does nothing; scaling transformations, which act like a zoom lens to magnify or reduce the picture; and rotations, which rotate the picture about the origin by a fixed angle. It is possible to combine linear transformations by doing one after another to create new ones.

The simplest way to understand the effect of a linear transformation in two dimensions is to consider what it does to the unit circle, which is a circle of radius one around the origin. The identity transformation leaves the circle unchanged; scaling transformations make the circle bigger or smaller; rotations leave the unit circle seemingly unchanged, although circles centered elsewhere are rotated as a whole. Another example is turning the vertical gain of the video screen to zero; that is, projecting all the points onto a horizontal line through the origin. In this case the unit circle gets transformed into a line segment centered at the origin. In three dimensions, replace the unit circle with the unit sphere.

C.2 Singular and Nonsingular Linear Transformations

It turns out that a two-dimensional linear transformation can only map the unit circle in one of three special ways; it can map it onto an ellipse or circle which is centered at the origin; or it can map the unit circle onto a segment of a line passing through the origin; or it maps all the points in the vector space onto the origin. This last linear transformation is a special one called the zero transformation. Transformations that map the unit circle onto a circle or an ellipse are called *nonsingular*. *Singular* transformations are those that map the unit circle into a line segment or point. There are no other possibilities. A singular linear transformation that maps the unit circle to a line segment is not one-to-one; it maps more than one point in the unit

circle into the same point on the line segment. Such a transformation can't be undone by a linear transformation because a point on the line segment may have come from more than one point on the unit circle, and there's no way to tell for certain which one. Nonsingular transformations are always one-to-one.

C.3 Inverses of Linear Transformations

Any nonsingular linear transformation can be undone or inverted. For instance, a scaling transformation that magnifies by two can be undone by a scaling transformation that reduces by two. A 45-degree clockwise rotation can be inverted by a rotation of 45 degrees counterclockwise. An inverse to a transformation is another tranformation such that when it is applied after the original transformation is applied, the *net result* is that all the elements in the vector space are left unchanged.

For transformations that map a vector space into itself, having an inverse is equivalent to being nonsingular. Transformations that have inverses are nonsingular; transformations without inverses are singular. To understand singularity, consider the cases of ordinary multiplication and division of numbers. The transformation "multiply by x", as in z = x * y, is nonsingular unless x = 0. An inverse transformation "divide by x", as in y = z / x, exists as long as x <> 0. When x = 0, "multiply by x" is singular and does not have an inverse transformation. We could define a "pseudo-inverse" transformation:

```
if x = 0 then y = 0 else y = z/x
```

which exists for any x, but we would not always expect to recover the original value of y. Pseudo-inverses that make somewhat arbitrary choices can be defined for any linear transformation. Inverses, when they exist, are unique. Pseudo-inverses are never unique.

C.4 Transformations between Spaces of Different Dimension

Transformations may be defined that map elements of one vector space into elements of another. For instance, a painting of a three-dimensional scene is based on the artistic perspective convention for mapping three dimensions into two.

Linear transformations that map vectors from two dimensions to three can at best map the unit circle into a two-dimensional object in the three-dimensional space. Transformations from three dimensions to two map the unit sphere into at most a two-dimensional object. Generally speaking, a transformation that maps the unit circle or sphere into an object of the maximum possible dimensionality is said to be of full rank. Otherwise it is said to be rank-deficient. When the two spaces are the same, then "full rank" means nonsingular and "rank-deficient" means singular.

C.5 Arrays and Matrices

The BASIC language deals with arrays of numbers, rather than elements of a vector space and transformations upon them. Arrays of numbers can have any meaning that the programmer wishes to assign, but conventionally vectors are represented by an array with one dimension. Thus an element, U, of a

two-dimensional vector space might be declared **dim U(2)**, while an element, V, of a three-dimensional vector space might be declared **dim V(3)**.

Conventionally, for a point in a two-dimensional space, **U(1)** is the first coordinate, along the x axis, and **U(2)** is the second coordinate, along the y axis.

The size of a vector is measured by its Euclidean length, which is the square root of the sum of the squares of its elements:

```
100 LengthU = sqr( U(1)^2 + U(2)^2 )
```

A linear transformation can also be represented by an array of numbers. Linear transformations mapping n-dimensional spaces to m-dimensional spaces are conventionally declared as **dim A(m,n)**. The following discussion uses the term "matrix" to refer to an array representing a linear transformation. The individual components of a matrix **A** depend on the linear transformation that **A** represents.

The components of an array **A** representing a two-dimensional linear transformation can be determined by examining the effect of the transformation on the standard unit vectors E1 and E2; where E1 and E2 are a pair of perpendicular vectors that have coordinates **(1,0)** and **(0,1)** respectively. The elements of the first column of **A** are the coordinates of the result of applying the transformation to E1. Similarly, the second column is the result of applying the transformation to E2.

For example, let's suppose the transformation is a counterclockwise rotation of 90 degrees. Then E1 gets mapped into E2 and E2 gets mapped into -E1, where -E1 has coordinates **(-1,0)**. The matrix A representing this transformation would be declared and its coordinates assigned as follows:

```
100 dim A(2,2)
110 rem  90 degree rotation of E1 gives the result (0,1)
120 A(1,1) = 0 : A(2,1) = 1 : rem the first column of A
130 rem  90 degree rotation of E2 gives the result (-1,0)
140 A(1,2) = -1 : A(2,2) = 0 : rem the second column of A
```

In two dimensions, to represent the identity transformation, write:

```
100 dim I(2,2)
110 mat I = idn(2,2)
```

To represent a three times magnification, write:

```
100 dim M(2,2)
110 mat M = idn(2,2)
120 mat M = (3) * M
```

and to represent a counterclockwise rotation through angle T:

```
100 dim R(2,2)
110 C = cos(T)
120 S = sin(T)
130 R(1,1) = C
140 R(2,1) = S
150 R(1,2) = -S
160 R(2,2) = C
```

The above transformations are all nonsingular. One singular transformation is the zero transformation, which maps everything to the origin:

```
100 dim Z(2,2)
110 mat Z = zer(2,2)
```

Another singular transformation maps any vector onto the x-axis:

```
100 dim P(2,2)
110 mat P = zer(2,2)
120 P(1,1) = 1
```

This maps the unit circle into a line segment on the x-axis.

C.6 Computing the Results of Transformations

With the conventions for vectors and transformations outlined above, the BASIC language provides operations for applying transformations to vectors and combining transformations.

To apply a transformation to a vector, you multiply the vector on its left side by the matrix representation of the transformation. For a two-dimensional vector U, its result V and the transformation A defined above; we have:

```
90  dim U(2,1), V(2,1)
...
210 U(1,1) = 1 : U(2,1) = 1
220 mat V = A * U
```

Then the result V would have coordinates (-1,1).

To combine two transformations, multiply their matrices together. To represent a transformation C which first performs A and then performs B, write:

```
90  dim C(m,n), B(m,p), A(p,n)
...
100 mat C = B * A
```

If you ever wondered why the textbook definition of matrix multiplication is so complicated, it is to insure that transformations can be combined by multiplying their matrices in this way. Matrix multiplication works only when the second dimension of B is the same as the first dimension of A, because it only makes sense to combine two such transformations when the result space of A is the same as the operand space of B.

C.7 Inverse Transformations

We mentioned earlier that nonsingular matrices have inverses. To get the inverse transformation Y of a nonsingular matrix A, write:

```
100 mat Y = inv(A)
```

Inv always returns a result, Y, which is the inverse if A is square and nonsingular; Y is a pseudo-inverse otherwise.

Inv is provided because it is traditional in BASIC systems. It is rarely needed for most BASIC programs. As you will see in the next section, there is a faster and more accurate way of getting the results that **inv** is capable of providing.

C.8 Solving Linear Equations

The BASIC language for the Lisa also provides operations for solving matrix equations and for computing inverse and pseudo-inverse transformations.

We have seen that applying a transformation A to a vector X is simply multiplying them together to get the transformed vector B:

```
100 mat B = A * X
```

We can also go the other way; given B and nonsingular square A, we can find the vector X:

```
90  dim X(3,1), B(3,1), A(3,3)
...
100 mat X = linsys(A, B)
```

The traditional name for this problem is "solving a system of n linear equations in n unknowns". In terms of transformations, finding the X that maps into B is equivalent to inverting or undoing the transformation and applying the inverse transformation to B to get the result X.

If you want to invert the same transformation for p vectors at a time, where A is an nxn singular matrix, then declare **dim B(n,p), A(n,n), X(n,p)** and input the p vectors as the columns of B. Then each column of X will represent the inverse of the transformation A applied to the corresponding column of B.

In the foregoing we assumed that A was nonsingular and square. In general, one can still inquire whether the matrix equation B = A * X has any solutions even when A is not square or is singular. Declare **dim B(n,1), A(n,m), X(m,1)** and **Linsys** will attempt to return a vector X such that the transformation A maps X close to B. Alternatively, you can think of **linsys** as attempting to

find a pseudo-inverse of A that maps B close to X. In this nonsquare case we would write something like

```
90  dim X(4,1), B(3,1), A(3,4)
...
100 mat X = linsys(A, B)
```

If we view the matrix equation A * X = B as a system of linear equations, then the following may be a more familiar representation of "a system of n linear equations in m unknowns":

$$
\begin{array}{ccccccccc}
A_{1,1}*X_1 & + & A_{1,2}*X_2 & + & A_{1,3}*X_3 & \cdots & + & A_{1,m}*X_m = & B_1 \\
A_{2,1}*X_1 & + & A_{2,2}*X_2 & + & A_{2,3}*X_3 & \cdots & + & A_{2,m}*X_m = & B_2 \\
 & & \cdots & & & \cdots & & = & \cdots \\
A_{n,1}*X_1 & + & A_{n,2}*X_2 & + & A_{n,3}*X_3 & \cdots & + & A_{n,m}*X_m = & B_n
\end{array}
$$

When A is singular or not square, **linsys** will still compute X. Sometimes there will be more than one possible solution X that makes B - A*X zero. However, only one solution is found by **linsys**. At other times there are no solutions X. When this is the case, an X is returned that minimizes the length of the residual B - A*X, but may not necessarily be able to make it zero. One way to determine the acceptability of a solution is to compute the residual and compare it using a tolerance **tol**:

```
100 dim A(3,4),B(3,1),X(4,1),R(3,1)
...
200 tol = 1E-14
210 mat X = linsys(A,B)
220 mat R = A * X
230 mat R = B - R
240 normR = 0
250 normB = 0
260 for I% = 1 to 3
270     normR = normR + R(I%,1)^2
280     normB = normB + B(I%,1)^2
290 next I%
300 if sqr(normR) > tol * sqr(normB) then print &
        "Residual exceeds tolerance"
```

If A is square, another way to find X is to find the inverse transformation itself, and apply it to B:

```
100  dim Y(3,1), B(3,1), A(3,3)
...
200 mat Y = inv(A)
210 mat X = Y * B
```

This is much less desirable because it is slower and less accurate to compute the whole inverse matrix than to use **linsys**.

Although it is easier to use **inv**, the multiple solution vector feature of **linsys** may be used to find the inverse of A:

```
100  dim Y(3,3), B(3,3), A(3,3)
...
200 mat B = idn(3,3)
210 mat Y = linsys(A, B)
```

C.9 Accuracy

A considerable amount of computation is involved in the calculation of **linsys** and **inv**. Even though BASIC uses high-precision arithmetic and sophisticated algorithms that minimize numeric errors, rounding errors sometimes do accumulate. However, the reliability of the answers to the equation A*X = B depends on more than just numerical accuracy. There are some systems of linear equations where small changes in the data (the elements of A and B) cause the answers (the elements of X) to differ greatly. This reliability question is an inherent property of linear algebra and exists even when the numeric precision is exact.

C.9.1 Condition Numbers

A set of linear equations is said to be *ill-conditioned* if the solutions are very sensitive to small changes in the coefficients, that is, the elements of the arrays A or B in the matrix equation A*X = B. The following system has a solution X = (11,1).

$$X_1 - X_2 = 10$$

$$X_1 - 1.000001 * X_2 = 9.999999$$

On the other hand, making small changes to the coefficients produces a solution X = (-1,-11).

$$X_1 - X_2 = 10$$

$$1.000001 * X_1 - X_2 = 9.999999$$

The inverses of the versions of the matrix A shown above are considerably different also. The problem is that both versions of the matrix are "nearly singular"; that is, they are very close to being the singular matrix:

```
100 mat A = con(2,2)
```

BASIC provides a way of finding out when a solution vector or an inverse matrix is unreliable by calculating a *condition* value for the matrix A

whenever **linsys** or **inv** is called. After an attempt has been made to solve a system of linear equations, there will be evidence you may examine by calling the **cond** function:

Cond will be zero for singular and rank-deficient matrices A and greater than zero for nonsingular and full rank matrices A. The largest possible value of **cond** is 1, which is attained by the identity and rotation matrices, among others.

```
100 mat X = linsys(A, B)
110 C = cond
120 if (1+C) = 1 then print "A is singular"
```

If 1+C rounds to 1 then you know that C is smaller than the level of rounding error. **Cond** is actually an estimate of the relative change in A to make A into the nearest singular matrix. Matrices with small **cond** are badly conditioned and often cause trouble because they are close to singular. The corresponding transformations map the unit circle into very skinny ellipses, which from a distance look much like the line segments generated by singular transformations. Two points on opposite sides of such a skinny ellipse may be very close together, perhaps within a rounding error, but the corresponding points on the unit circle that they were mapped from may be much further apart. Small errors like rounding errors can thus cause big errors when solving for the inverse transformation.

C.9.2 Ill-Conditioned Problems

All the operations we have discussed are subject to rounding errors after each floating-point operation. This has important implications because rounding errors blur the distinction between singular and nonsingular problems. A matrix may be nonsingular, but if it is close enough to a singular matrix, the result X may not be satisfactory; it may be far from the correct solution X, and the residual R might not be small compared to B. **Cond** supplies an estimate of the effect of rounding. Generally, you cannot count on more than 15 + **log10(cond)** significant digits being correct in the largest component of X, with fewer reliable digits in smaller components:

```
100 mat X = linsys( A, B)
110 C = cond
120 if C = 0 then print "A is singular"
130 if C = 0 then ND = 0 else ND = 15 + log10(C)
140 if ND > 0 then print fix(ND)," digits of X are reliable"
150 if (1+C) = 1 then print "X is completely unreliable"
```

Again, we have checked to see if 1+C rounds to 1.

C.10 Determinants

BASIC provides the **det** function to obtain the determinant of the last matrix supplied to **inv**. Det is traditionally used in BASIC to determine whether A is singular or nonsingular, since the determinant of a singular square matrix is zero and the determinant of a nonsingular square matrix is not zero:

```
100 mat X = inv(A)
110 if det = 0 then print "A is singular"
```

The value of **det** is not related to the condition (**cond**) of the problem. For instance, the statements

```
100 mat A = idn(2,2)
110 mat A = (k) * A
```

produce a matrix A with perfect condition number 1 but with determinant k^2 which could be large or small, while the statements

```
100 mat A = idn(2,2)
110 A(1,1) = k
120 A(2,2) = 1/k
```

produce, for k >= 1, a matrix A with condition 1/k^2 which could be very small, but with determinant 1. Since **det** can be used only to distinguish singular from nonsingular, and rounding errors blur this distinction, the use of **det** is *not* recommended. Use **cond** instead.

*Note: Det is computed only when performing **inv**, not when performing **linsys**. Det is a NaN if the matrix is not square.*

C.11 Linear Least-Squares Problems

Linear least-squares problems are a generalization of linear equation problems. The dimensions are typically **dim** A(n,p), X(p,m), B(n,m). In both cases the solution X minimizes the length of the residual B - A * X; for linear equations with square nonsingular A, the residual would be exactly zero in the absence of rounding errors. In overdetermined least-squares problems where there are more equations than unknowns, R is not zero. In underdetermined least-squares problems where there are more unknowns than equations, R is zero, and although there is more than one solution X, only one is returned.

To obtain a single least-squares solution of a problem with 100 observations and 3 unknowns:

```
100 dim A(100,3),X(3,1),B(100,1)
110 mat input A
120 mat input B
130 mat X = linsys( A, B )
140 mat print X
```

Cond is usually zero if A is rank-deficient and greater than zero if A is of full rank.

C.12 Expert's Corner

Linsys and **inv** use column pivoting to factor the matrix A into a product of an orthogonal matrix Q and an upper triangular matrix R; A = Q*R. This factorization is then used to solve for X, the unknown vectors or pseudo-inverse matrix. **Cond** is an estimate of the inverse of the conventional condition number. When A(n,p) is not square, **cond** is not zero unless A is rank-deficient.

Appendix D
Error Messages

D.1 Recoverable Errors D-1
D.2 Fatal Errors D-4

Error Messages

This appendix lists all the BASIC error messages, recoverable and fatal. Recoverable errors are just that; if the error occurs, you can recover from it if you provide appropriate error-handling routines. Fatal errors cause a nonrecoverable run-time error.

D.1 Recoverable Errors

You can use the **on error goto** statement to direct program execution to error-handling code when these errors occur. The error number that precedes each message may be inspected through the **err** variable.

4 Can't write to file
A write operation failed while transferring output characters or the contents of a file buffer to the file system.

4 Can't write values of virtual array to channel
A failure occurred while a file buffer or output character was being written.

4 Error in writing file
An error occurred while a file buffer or output character was being written.

4 No space for virtual array
No space is available for the virtual array.

4 Error writing virtual array element to channel
An error occurred while a virtual array element was being written to a channel.

5 No file "string1" to NAME AS "string2"
String1 does not exist, or **string2** is an invalid name.

5 Can't open file <name>
The file specified in an **open** statement cannot be opened.

5 Can't find file <name>
The file specified in an **open** statement cannot be found.

9 Attempt to reference an unopened channel
The channel must be opened before it may be referred to in an input or output operation.

9 Channel not open
An input/output operation was requested to a channel which has not been opened by the **open** statement.

9 FIELD channel not open
A reference to an unopened channel has been made in a **field** statement.

11 End of file on device
An attempt was made to read beyond the end of the file.

15 Keyboard wait exhausted
A maximum was set on how long the program will wait for input from the keyboard. This error message appears when the time is up.

31 Buffer sizes smaller than default not supported
A **recordsize** option in an **open** statement requested a buffer smaller than the default (512 bytes).

31 Can't have using value larger than recordsize
A **get** or **put** statement contains a **using** value larger than the size of the buffer.

38 Heap exhausted
There is no more space available in the heap (allocatable memory).

43 Virtual array must be on disk file
A request to open a virtual array lists a device which is not a disk (such as the console).

45 Virtual array not yet open
The file associated with a virtual array must be opened before the first statement referring to the array is executed.

46 Channel number out of range
The channel number (data channel) listed in an I/O statement is out of the legal range. Channel numbers must be between 1% and 12%. (0% is always associated with the console.)

46 Channel number in open out of range
The channel number (data channel) listed in an **open** statement is out of the legal range. Channel numbers must be between 1% and 12%. (0% is always associated with the console.)

46 Channel number in close out of range
The channel number (data channel) in a **close** statement is out of the legal range. Channel numbers must be between 1% and 12%. (0% is always associated with the console.)

50 Bad input format
The system is trying to read a value from an **input** or **read** statement and the data are in an incorrect format, e.g., alphabetic data in a numeric variable.

52 Integer too big
Integers must be within the range -32768 to +32767.

55 **Current matrix dimension smaller than specified**
A dimension of the matrix specified in a **mat print** statement is greater than the actual size of the matrix.

55 **Negative bounds not allowed**
The dimension statement for a matrix contains a negative value; only non-negative integers (0 to +32767) are allowed.

55 **Subscript out of range**
A reference to an array contains a subscript which is outside its predefined range.

55 **Matrix dimension error**
The operands to a matrix operator do not match. In other words, the matrices involved in the operation are not of the appropriate dimensions.

55 **Dimensions or maximum size prevents redimensioning**
This error occurs when an attempt is made to redimension a matrix from one to two dimensions or vice-versa, or to redimension an array, making it larger than was defined in the **dim** statement.

55 **FIELD overflow buffer**
The amount of space requested in a **field** statement exceeds the amount available in the buffer.

57 **Out of data**
A **read** statement ran out of data.

69 **VAL input string too long**
The input string for **val** was too long.

70 **NUM1$ result string too long**
The string exceeds 255 characters.

72 **INV or LINSYS argument dimensions improper**
In **inv**(A) or **linsys**(A,B) one of the dimensions of A or B is less than 1, or in **linsys**(A,B) the number of rows of A is not equal to the number of rows of B.

73 **Must not use $$ format with exponential notation**
You may not specify exponential notation (^^^^) and dollar sign fill characters ($$) in the same **print using** format.

73 **Must not use * fill with exponential notation**
You may not specify exponential notation (^^^^) and asterisk fill characters (*) in the same statement.

73 **Can't use * fill with leading minus sign**
When the asterisk (*) is used to replace leading zeroes in a **print using** statement, negative amounts must be indicated with a trailing minus sign.

73 **Can't use $$ format with leading minus sign**
When the dollar sign ($$) is used to replace leading zeroes in a **print using** statement, negative amounts must be indicated with a trailing minus sign.

80 **Invalid Operation**
An invalid operation was encountered. This error is signaled only if its set halt flag is on.

81 **Numerical conversion overflow**
A floating-point value was too large to convert to an integer variable. This error is signaled only if its set halt flag is on.

82 **Floating point overflow**
A floating-point value was either too large or too small. This error is signaled only if its set halt flag is on.

83 **Floating point underflow**
A floating-point value suffered excessive roundoff because it was too close to zero. This error is signaled only if its set halt flag is on.

84 **Inexact Calculation**
The result of a calculation was too inexact to be represented. This error is signaled only if its set halt flag is on.

85 **Division by zero**
An operation resulted in an illegal floating-point division by zero. This error is signaled only if its set halt flag is on.

D.2 Fatal Errors

The errors below are fatal. When one of these errors occurs, it causes a run-time error.

Virtual array must not be both source and dest
In a matrix operation where the result is a virtual array, the same matrix may not appear on both sides of the equation. If X is a virtual array, the following statements are *not* legal:

```
mat X = X * Y
mat X= trn(X)
```

Result of string arithmetic too long
The result of a string arithmetic operation contains more than 56 characters.

Attempt to divide by zero in string arithmetic
Division by zero is an illegal operation in string arithmetic.

Can't redimension virtual array
Redimensioning a virtual array is an illegal operation.

Must not use file as virtual array and for I/O
Once a file has been opened for virtual array storage, it may not be accessed for ASCII I/O or block I/O.

Must not get or put virtual array or I/O file
Get and **put** are illegal operations on a file that has been opened for ASCII I/O or virtual array storage.

String operand has incorrect format
A character string used in a string arithmetic statement may not contain any characters other than plus (+), minus (-), decimal point (.), and digits (0 through 9).

Invalid label number in CHAIN
An invalid line number was specified in a **chain** statement.

CHAIN file not found
The file name entered in a **chain** statement cannot be found.

Only blanks allowed between \ in USING string
Characters other than blanks appear between backslashes (\) in a **print using** statement.

Missing matching \ in USING string
An odd number of backslashes (\) appear in a **print using** string.

Incorrect USING format to print string
The **using** string contains information which is not in the correct format for the data to be printed.

Missing END statement
The program does not contain an **end** statement.

Syntax error
Incorrect syntax is found by the interpreter.

RETURN without GOSUB
A **return** statement is encountered, but no **gosub** statement has been executed.

RETURN from DEF FNx
A **return** statement was encountered in a function.

GOTO target does not exist
The line number specified in a **goto** statement is not valid.

GOSUB target does not exist
The line number specified in a **gosub** statement is not valid.

Can't LSET or RSET Virtual Arrays
The **lset** and **rset** cannot be used with virtual arrays.

Can't RESUME
A **resume** command was executed, but no **on error goto** routine had been entered.

Can't CONTINUE
You can only use a **cont** command after a **stop** command has been entered.

Call of undefined function
A reference was made in the program to an undefined function.

Can't use Virtual arrays in FIELD statement
The **field** statement cannot be used with virtual arrays.

Negative FIELD width
The **field** statement requires positive field values.

Appendix E
BASIC Workshop Files

Appendix E
BASIC Workshop Files

This appendix lists the files on the BASIC 1.0 diskettes.

File Name	BASIC Diskette	Notes	Description
BASIC.obj	2		Workshop program.
BYE.TEXT	1		Workshop installation exec file.
ByteDiff.obj	2		Utility program.
Cistart.text	1		Workshop installation exec file.
Diff.obj	2		Utility program.
DumpPatch.obj	2		Utility program.
EDIT.MENUS.TEXT	2		Editor support file.
Editor.obj	2		Workshop program.
Filediv.obj	2		Utility program.
Filejoin.obj	2		Utility program.
find.obj	2		Utility program.
FMDATA	1	1,2	Data segment.
font.heur	1	1,2,3	Data needed to support SYS1Lib.
FONT.HEUR	2		Second copy of same file.
font.lib	1	1,2,3	Data needed to support SYS1Lib.
GETPROFILELOC.TEXT	1		Workshop installation exec file.
GETYESNO.TEXT	1		Workshop installation exec file.
INSERTDISK.TEXT	1		Workshop installation exec file.
Intrinsic.lib	1	2,3	Library directory.
IOSFplib.obj	2		Library unit w/interface.
IOSPaslib.obj	1	2,3	Library unit w/interface.
LDSPREFERENCES.OBJ	2		Workshop program.
LDS_RES_PROCS.TEXT	2		Workshop data.
OSERRS.ERR	1	3	Workshop data.
PAPER.TEXT	2		Workshop data.
Portconfig.obj	2		Utility program.
resident_channel	1	1,2,3	System data.

Note 1: These files are identical to Office System Release 1.0 files.

Note 2: These files are identical to Office System Release 1.2 files. Office System 1.2 is functionally identical to Office System 1.0, but is released to ensure compatibility with Pascal 1.0, BASIC-Plus 1.0, and COBOL 1.0.

Note 3: These files are the minimum necessary to run a user program in the Workshop environment. A user program may require other files as well.

File Name	BASIC Diskette	Notes	Description
Shell.WorkShop	1	3	Workshop main program.
Sulib.obj	1	3	Library unit w/interface.
Sxref.obj	2		Utility program.
SXREF.OMIT.TEXT	2		Data.
Sys1lib.obj	1	1,2,3	Library units (no interface).
SYS2LIB.OBJ	2	1,2,3	Library units (no interface).
SYSTEM.BT_PROF	1	1,2,3	System support.
SYSTEM.BT_TWIG	1	1,2,3	System support.
SYSTEM.IUDIRECTORY	1	1,2,3	System data.
SYSTEM.LLD	1	1,2,3	System program.
SYSTEM.LOG	1	1,2,3	System data.
SYSTEM.OS	1	2,3	System program.
System.Shell	1	1,2,3	System program.
SYSTEM.STACK1	1	1,2,3	System data.
SYSTEM.STACK2	1	1,2,3	System data.
SYSTEM.STACK3	1	1,2,3	System data.
SYSTEM.STACK4	1	1,2,3	System data.
SYSTEM.SYSLOC1	1	1,2,3	System data.
SYSTEM.SYSLOC2	1	1,2,3	System data.
SYSTEM.SYSLOC3	1	1,2,3	System data.
SYSTEM.SYSLOC4	1	1,2,3	System data.
SYSTEM.TIMER_PIPE	1	1,2,3	System data.
SYSTEM.UNPACK	1	1,2,3	System data.
term.menus.text	2		Data for transfer program.
transfer.obj	2		Workshop program.
WMDATA	1	1,2	Data segment.
{T11}BUTTONS	2	2	Data.
{T11}MENUS.TEXT	2	2	Data.

Note 1: These files are identical to Office System Release 1.0 files.

Note 2: These files are identical to Office System Release 1.2 files. Office System 1.2 is functionally identical to Office System 1.0, but is released to ensure compatibility with Pascal 1.0, BASIC-Plus 1.0, and COBOL 1.0.

Note 3: These files are the minimum necessary to run a user program in the Workshop environment. A user program may require other files as well.

NOTES

Index

Please note that the topic references in this index are *by section number.*

----------A----------

abs 10.3.1
access, virtual arrays 12.3
ampersand (&) *See* Special Characters *at end of index.*
and 4.6
annuity 10.3.17
append 3.3, 3.3.1.7
Apple-period interrupt 3.3, 3.3.3.3
arithmetic functions 10.3
arithmetic operators 4.5
ascii function 10.4.21
ASCII input and output 5
ask exception 13.3
ask halt 13.5
ask rounding 13.8
assignment statement 2.11
asterisk (* or **) *See* Special Characters *at end of index.*
atn 10.3.8

----------B----------

backslash (\) *See* Special Characters *at end of index.*
block input and output 11.3
branching 6
buffer management 11.3
bufsiz 11.1.3
bye 3.3.5

----------C----------

caret (^) *See* Special Characters *at end of index.*
case of letters 2.2.1
catalog and cat 3.3, 3.3.2.3
ccpos 10.3.19
chain 14.6
change 10.7
channel 11.1.1, 11.2, 11.3.2.1
channels, input and output 5.1
character set 2.2
chr$ 10.4.8
clearing program space 3.3.1.2, 3.3.1.3
close 11.2
clustersize 11.1.3
comma (,) *See* Special Characters *at end of index.*
commands, system 3.3
  debugging 3.3.4
  informational 3.3.2
  leaving BASIC 3.3.5
  program execution 3.3.3
  program space 3.3.1
comments 2.9
communication with files 5.1
comp% 10.4.17
comparing expressions 4.7
compound 10.3.16
con 9.2
concatenation 10.4.6
cond 10.5.5
conditional branching 6
constant
  floating-point 4.1
  integer 4.1
  string 4.2
cont 3.3, 3.3.3.2
conversion overflow 13.1.4
copying BASIC interpreter disk 1.2
cos 10.3.5
creating functions 10.6

cvt functions 10.4.11
cvtoverflow exception 13.1.4

----------D----------
data 5.2
  manipulation 4
  types 4
date$ 10.4.23
debugging commands 3.3.4
def* 10.6.1
delete 3.3, 3.3.1.1
destructive backspace 3.1
det 10.5.3
dif$ 10.4.13
digits 2.2
dim
  matrices 9.1
  virtual arrays 12.1
dimensioning matrices 9.1
dimensioning virtual arrays 12.1
dividebyzero 13.1.2
division by zero 4.5.1
division by zero exception 13.1.2
dollar sign ($) *See* Special Characters *at end of index.*

----------E----------
editing
  in BASIC 3.1
  in Workshop Editor 3.2
elements 2.6
equal sign (= or ==) *See* Special Characters *at end of index.*
eqv 4.6
err and erl 6.5, D
error 6.5, D
exceptions, floating-point 13.1
  checking for 13.3
  setting 13.2
exclamation point (!) *See* Special Characters *at end of index.*
execution modes 2.8
exp 10.3.7
expressions 4.4

----------F----------
factor 4.4
field 11.3.2.1
file length, virtual arrays 12.4
filesize 11.1.3
fix 10.3.11
floating-point
  arithmetic 4.1.2
  constants 4.1
  exceptions 13.1
  rounding modes 13.6
fnend 10.6.2
for next 7.1
for statement modifier 8.2
for until 7.4
for while 7.3
formatting output 5.7
functions 10
functions, creating your own 10.6

----------G----------
get 11.3.1
gosub 6.4, 10.1
goto 6.2, 6.3, 6.5, 6.7
greater than (>) *See* Special Characters *at end of index.*
greater than or equal (>=) *See* Special Characters *at end of index.*

----------I----------
identifiers 2.10
idn 9.2
if goto 6.2
if statement modifier 8.1
if then else 6.1
immediate mode 2.8
imp 4.6
inexact exception 13.1.6
informational commands 3.3.2

initialization of variables 4.3
input
  ASCII 5
  block 11.3.1
  channels 5.1
  matrices 9.4
input line 5.5
instr 10.4.5
int 10.3.11
integer
  arithmetic 4.1.2
  constants 4.1
  range 4.1.2
Interpreter 2.1, 3.1
interrupt character (-period) 3.3, 3.3.3.3
intpart 10.3.14
intpart% 10.3.15
inv 10.5.2
invalid operation exception 13.1.1
I/O
  block 11.3.1
  channels 5.1
  formatted ASCII 5

----------K----------
keywords 2.4
kill 14.8

----------L----------
leaving BASIC 3.3.5
left 10.4.2
len 10.4.1
length 3.3, 3.3.2.2
less than (<) *See* Special Characters *at end of index.*
less than or equal (<=) *See* Special Characters *at end of index.*
let 2.11
letter 2.2
line 2.5, 5.5
line number 2.7
linsys 10.5.4
list and listnh 3.3, 3.3.2.1
log 10.3.9
log10 10.3.10
logical operators 4.6
looping 7
lowercase letters 2.2.1
lset 11.3.2.2

----------M----------
mat 9.2
mat input 9.4
mat print 9.5
mat read 9.3
matrices 9
  addition 9.6.1
  calculations 9.6
  functions 10.5
  initialization 9.2
  input 9.4
  multiplication 9.6.2
  printing 9.5
  subscripts 9
  subtraction 9.6.1
mid 10.4.4
minus sign (-) *See* Special Characters *at end of index.*
mode 2.8, 11.1.3
multiple statement modifiers 8.6

----------N----------
name as 14.7
nested loops 7.6
nesting subroutines 10.2
new 3.3, 3.3.1.2
not 4.6
not equal (<>) *See* Special Characters *at end of index.*
num$ 10.4.19
num1$ 10.4.20
numeric notation 4.1.1

----------O----------
offset 11.3.1
old 3.3, 3.3.1.3
on error goto 6.5, D
on gosub 6.4
on goto 6.3
open 11.1
operators
   logical 4.6
   precedence of 4.8
   relational 4.7
or 4.6
output
   ASCII 5
   block 11.3.1
   channels 5.1
   formatting 5.7
overflow exception 13.1.3

---------P,Q---------
percent sign (%) *See* Special Characters *at end of index.*
pi 10.3.3
place$ 10.4.16
plus sign (+) 4.5, 9.6, 10.4.6
pos 10.3.19
pound sign (#) *See* Special Characters *at end of index.*
precedence of operators 4.8
print 5.6
print using 5.7
print zones 5.6
printing matrices 9.5
prod$ 10.4.14
program line 2.5
program mode 2.8
program space 3.1
prompt 5.4
put 11.3.1
quo$ 10.4.15
quotation marks (' or ") *See* Special Characters *at end of index.*

----------R----------
rad$ 10.4.22
randomize 10.3.12
read 5.2
record 11.3.1
recordsize 11.1.3
recount 11.3.1
recursion 10.2
relational operators 4.7
rem 2.9
renumber 3.3, 3.3.1.8
replace 3.3, 3.3.1.4
reserved words 2.4, A.4
restore 5.3
resume 6.6
return 10.1
right 10.4.3
rnd 10.3.12
rnear 13.6
rneg 13.6
rounding modes 13.6
rpos 13.6
rset 11.3.2.2
run and runnh 3.3, 3.3.3.1
rzero 13.6

----------S----------
save 3.3, 3.3.1.5
scientific notation 4.1.1
semicolon (;) *See* Special Characters *at end of index.*
set exception 13.2
set halt 13.4
set rounding 13.7
sgn 10.3.13
sin 10.3.4
slash (/) *See* Special Characters *at end of index.*
sleep 14.2
space$ 10.4.7
spaces 2.3

special characters 2.2; *see also end of index.*
sqr 10.3.2
statement 2.5
statement modifiers 8
status 11.1.3
storage, virtual arrays 12.2
string 2.6
string constant 4.2
string functions 10.4
string$ 10.4.9
subroutines 10
subroutines, branching to 6.4, 10.1
sum$ 10.4.12
swap% 10.3.21
system commands 3.3
system statements 14

----------T----------
tab 10.3.20
tan 10.3.6
time 10.3.18
time$ 10.4.24
trace 3.3, 3.3.4.1
translating string characters 10.4.10
trn 10.5.1

----------U----------
unconditional branching 6.7
underflow exception 13.1.5
unless statement modifier 8.5
unlock 14.5
unsave 3.3, 3.3.1.6
until next 7.5
until statement modifier 8.4
uppercase letters 2.2.1
using 11.3.1, *See* print using.

----------V----------
val 10.4.18
variable 4.3
variables command 3.3, 3.3.4.2
virtual arrays 12
  access 12.3
  dim 12.1
  file length 12.4
  storage 12.2

----------W----------
wait 14.1
while next 7.2
while statement modifier 8.3
Workshop editor 3.2
writeallow 14.4
writeprotect 14.3

---------X, Z---------
xlate 10.4.10
xor 4.6
zer 9.2

--------Special Characters--------
-period 3.3, 3.3.3.3
! 2.9, 5.7
" 4.2
5.7
$ 2.10, 4.3, 5.2, 5.7
% 2.10, 4.1, 4.3, 5.2
& 2.5
' 4.2
* 4.5, 9.6
** 4.5, 5.7
+ 4.5, 9.6, 10.4.6
, 5.6, 5.7
- 4.5, 5.7, 9.6
/ 4.5
; 5.6
< 4.7
<= 4.7
<> 4.7
= 2.11, 4.7
== 4.7
> 4.7

>= 4.7
\ 2.5, 5.7
^ 5.7

THIS MANUAL was produced using LisaWrite, LisaDraw, and LisaList.

ALL PRINTING was done with an Apple Dot Matrix Printer.

the Lisa™
...we use it ourselves.

Apple publications would like to learn about readers and what you think about this manual in order to make better manuals in the future. Please fill out this form, or write all over it, and send it to us. We promise to read it.

How are you using this manual?
[ ] learning to use the product [ ] reference [ ] both reference and learning

[ ] other ____________________

Is it quick and easy to find the information you need in this manual?
[ ] always [ ] often [ ] sometimes [ ] seldom [ ] never

Comments ____________________

What makes this manual easy to use? ____________________

____

What makes this manual hard to use? ____________________

____

What do you like most about the manual? ____________________

____

What do you like least about the manual? ____________________

____

Please comment on, for example, accuracy, level of detail, number and usefulness of examples, length or brevity of explanation, style, use of graphics, usefulness of the index, organization, suitability to your particular needs, readability.

____

____

____

What languages do you use on your Lisa? (check each)

[ ] Pascal [ ] BASIC [ ] COBOL [ ] other ____________________

How long have you been programming?

[ ] 0-1 years [ ] 1-3 [ ] 4-7 [ ] over 7 [ ] not a programmer

What is your job title? ____________________

Have you completed:

[ ] high school [ ] some college [ ] BA/BS [ ] MA/MS [ ] more

What magazines do you read? ____________________

____

____

Other comments (please attach more sheets if necessary) ____________________

____

____

____

FOLD

FOLD

TAPE OR STAPLE