UNIVERSITY COMPUTING CENTER

ALISP

UNIVERSITY OF MASSACHUSETTS AT AMHERST

INTRODUCTION

ALISP is a timesharing and batch LISP 1.5 system operating on the CDC Cyber-74 installation at UMASS. It is similiar to the ALISP system previously operating on UMASS timesharing with the CDC 3600/3800. Some of the features of ALISP include: automatic dynamic adjustment of all storage areas up to the user's field length limit; uniform definition of functions using the value cell of literal atoms; explicit typing of all data forms for faster execution and redundancy tests; overlay complier and assembler; filing, editing and pretty-print packages for time-sharing. Applications packages include a full-scale relational dB embedded directly in ALISP, and GRASPER, a graph language (see separate manuals for these packages).

A few words about the manual. It was written with care and an attempt at precision and clarity, and should be read in the same spirit. There are some unavoidable omissions and ambiguities, but most of the information you need is findable inside; exercise patience. Since LISP is found by many to be a difficult language conceptually, I heartily recommend that you do not try to learn it simply from this manual, since I have pre-supposed a modicum of LISP ability. Using one of the learning manuals available (such as Weissman's _LISP 1.5 Primer_ or _The Little Lisper_), along with this as a reference manual, is a very fine and painless way to learn LISP.

This is more than a dense reference manual, however. Inside you will find lots of goodies and tips on LISP programming, as well as discussions of some exotic parts of LISP barely touched on by the learning manuals (such as READ macros). While it does not read like a novel, it would be most helpful if you could skim through it and familiarize yourself with its contents before sitting down for a heavy session at the terminal. Much temper and riffling through pages will be saved

ON CONVENTIONS

Conventions help make this manual more readable and less wordy. Certain pre-defined words are used extensively throughout; consult the Glossary when in doubt as to their meaning.

Numbers are always written using base 10 (decimal) representation, unless the letter "B" appears at the end of the digits; indicating an octal number. An optional exponent can appear after the "B"; this exponent is a base 10 integer specifying a left-shift count for the octal digits preceeding it.

Examples:

```
13B    = 11
10B21  =  8 * 2**21
```

in an obvious fortran-type notation for the exponent.

The boxed notation for representing S-expression is used several times, especially in section I.3. A good reference for this convention is Weissman's Primer. Usual notation for S-expressions is the parenthesized linear structure used on input to READ. Note that a comma "," is used rather than a dot "." for a general S-expression, as it usually does not make any difference which is used; when it is important to distinguish the two, it will be done.

Syntactic variables are used to avoid difficulty with evaluation of arguments when describing functions. Syntactic variables are indicated by use of the lower-case, as opposed to upper-case, which is reserved for actual ALISP code. If the arguments of the function are evaluated (SUBR, SUBR*, and LAMBDA functions), then a syntactic variable in the argument position stands for whatever the argument evaluates to. If the arguments of the function are not evaluated (FSUBR, FSUBR*, LSUBR, and FLAMBDA functions), then the syntactic variable stands for the actual argument. The differences can be seen by looking at QUOTE (FSUBR) and CONS (SUBR) examples:

```
(QUOTE x) -- x stands for:

                (FOO) in (QUOTE (FOO))
                BAR in (QUOTE BAR)

(CONS x y) -- x stands for:

                (FOO) in (CONS '(FOO) BAR)
                (BAR) in (CONS (LIST 'BAR) 1)
```

Syntactic variables allow easy description of functions without regard to their argument evaluation conventions. Note that the variable is underlined when used in descriptive text.

SUPPORT AND DISTRIBUTION

The Computer Center at UMass at Amherst will be supporting ALISP staring in Fall 1977. All inquiries on system problems should be addressed to:

Richard Hudson
University Computing Center,
Graduate Research Center,
University of Massachusetts

Amherst, MA 01003

Tel: (413) 545-2690

ALISP runs under KRONOS or NOS operating systems on CDC 6600 series computers. It will be distributed to requestors upon receipt of a tape. Distributed materials include the ALISP system in source form; ALISP, Relational DB, and GRASPER manuals; and an installation and internal specifications manual. The ALISP, Relational DB, and GRASPER manuals may be copied for non-profit purposes with permission from their respective authors.

Updates and news should be sent periodically from UMass. Any fixes to bugs should be sent to UMass so they can be distributed to other users.

Table of Contents

Part 1: The ALISP Language

Part II: Editing, Filing, and Pretty-Printing

I Chapter 1

Signing On and Keeping Up

This section describes the procedure for calling the ALISP system and exiting from it. (Note: Installations other than UMass may have a different procedure for staring ALISP.) The ALISP control card is described.

1.1 Signing On and Getting Off

Get a terminal and sign on properly (see the UMASS Timesharing Manual). At UMass, ALISP is a TELEX command which can be invoked from any subsystem (for BATCH operation, see Chapter I.15). To run, type "ALISP". The ALISP system will respond by printing "ALISP VERSION n" and then requesting input. You are now at the top level of ALISP, under the EVAL supervisor. The top-level ALISP prompt is a "?".

The interpreter will keep on evaluating stuff thrown at it until it evaluates the EXIT function, a SUBR of no arguments. The EXIT function gets you out of ALISP, back to the batch subsystem. It also prints out the CP time (in thousandths of a second) spent in ALISP, the number of garbage collects, and the maximum field length used, all in base 10 representation. A sample session by a beginning LISPer is given in Dialogue 1.1 below.

1.1.1 ALISP Control Card

ALISP allows control card parameters to be specified on execution. Legal parameter values and their effects are given in Appendix B. Unless the overlay option is used (LD parameter), ALISP will attempt to process all control card parameters when it is called. The parameters are processed from left to right; this order is important if, for instance, a file is to be read into the system (IF parameter) and the executing field length is lengthened (FL parameter). If the file read parameter occurs before the field length parameter, the ALISP system may not have enough room to perform the read, and will abort. If there are any illegal control card parameters, or if any errors occur during control card processing, an error message will be printed, and execution aborted.

The control card parameters are available to anyone who wishes to do his own control card processing. To bypass system processing of the control card, either the LD or OWN paramter.

Dialogue 1.1
Sample Terminal Session

Terminal Dialogue	Comments
TERMINAL: 110,TELEX	Sign-on message from TELEX
RECOVER/SYSTEM: BATCH $RFL,0	In batch subsystem of TELEX
/ALISP	Execute
ALISP VERSION 1.1	Now in ALISP
?(CONS 'A 'B)	Top-level supervisor in effect
(A,B)	
?(CAR '(FOO BAR))	
FOO	
?(EXIT)	Exit from ALISP
END ALISP RUN	
CP: 26 FL: 12400 GC: 0 /BYE	Statistics of the run back in batch subsystem of TELEX, sign off
xxx LOG OFF nnn xxx CP nnn	

The user may then examine the parameters via the PARAMCP function, a SUBR of no arguments. This function returns a list of the control card parameters. COMMAS, SLASHES and equal signs in the control card are returned as separate atoms in the list. Table 1.1 gives some examples.

Table 1.1
The PARAMCP Function

Control Card	PARAMCP Value
ALISP,IF=MYFNS	(IF=MYFNS)
ALISP,FL,PR	(FL,PR)
ALISP,IF=MYFNS/LISP000	(IF=MYFNS/LISP000)
ALISP,IF=MYFNS=YFNS,FL	(IF=MYFNS=YFNS,FL)

1.2 NEWS

The most recent ALISP news can be printed on entry to ALISP by using the NEWS parameter. To get the most recent news, use:

ALISP,NEWS.

To get all the news, use:

```
ALISP,NEWS=T.
```

The news can also be printed by evaluating the function NEWS, a SUBR of one argument. (NEWS NIL) prints the most recent news on SYSOUT, while (NEWS T) prints all news.

I Chapter 2

ALISP Data Types

This section describes all types of ALISP data, and gives information on their internal representation. This information is not crucial or even necessary for running ALISP, except perhaps for the section on number tokens; the incurious user may skip this section entirely without penalty.

2.1 ALISP Pointers

An ALISP pointer (or simply pointer) is the basic data format. It consists of 30 bits (half of a CDC Cyber-74 60-bit word) divided into two parts, the address and the indicator. The address is in the lower (right-half) 18 bits, the indicator in the upper 12 bits:

29 18	17 0
IND.	ADDRESS

The indicator tells what type of data the pointer is, e.g., an indicator of 1400B specifies an SNUM. The address portion holds either the data (SNUM), or an address in core that has more data. Only the lower 17 bits of the address are used at present, giving an addressing capability of 2B18 - 1, about 121K decimal words.

Two pointers can be stuffed together into an ALISP word (60 bits). The left half of the word is the CAR, the right half the CDR of the word.

The indicator bits allow fast testing of the data type. Indicator bit assignments are given in Table 2.1. Future data types, such as a binary tree data type may use the currently unused bits. Note that individual data types can set more than one indicator bit, e.g., SNUM's have an indicator of 1400B, specifying a number (bit 27) and an SNUM (bit 26).

2.2 Data Types

Here are descriptions of the pointers for the various data types. For more information on atomic data, see the sections on numeric operations (I.12) and literal atoms (I.5).

Table 2.1
Indicator Bit Assignments

bit	if set	if not set
29	list	atom
28	garbage-collect	info
27	number	not number
26	SNUM	not SNUM
25	BNUM	not BNUM
24	LNUM	not LNUM
23	FNUM	not FNUM
22	not used	
21	ANUM	not ANUM
20	STRING	not STRING
19	not used	
18	not used	

2.2.1 Literal Atoms

An atom which is not a number token or a string is a literal atom. Literal atoms are distinguished as having unique print-names, and (except for NIL) property-list (plist) and value attributes which are user-definable. Literal atoms will often be called litats. A literal atom which is not NIL will be called an nlitat. A literal atom which is neither NIL nor a GENSYM atom will be called an nglitat.

Indicator: 0000B
Address: points to the first atom data word

NIL has no plist or value attributes, and consequently does not use any atom data words. It has an address of 0B; it thus has a pointer of 30 zero bits.

2.2.2 Number-Tokens

Any atom which is not a litat or string is a number token, often referred to simply as a number. Not all numbers are amenable to standard arithmetic operations (FNUM's and ANUM's). Number tokens have bit 27 set in the indicator. There are four types:

i. SNUM (Small NUMber)

Indicator: 1400B
Address: integer value of the SNUM

Note that the SNUM address uses all 18 bits as a signed integer with maximum magnitude 2B17 - 1. This is the cheapest ALISP number storage, requiring only 30 bits of storage.

ii. BNUM (Big NUMber)

Indicator: 1200B
Address: points to floating-pt. number

The BNUM address points to a core location which holds a 59-bit floating-pt. number. This number is obtained from the CDC 60-bit floating-pt. format by clearing the low-order bit (bit zero), and shifting left circular by 58. This puts the cleared bit at bit position 58, where it is needed by the garbage-collect routine. BNUM's thus have one bit less precision than CDC floating-pt. numbers.

iii. LNUM (Logical NUMber)

Indicator: 1100B
Address: points to 48-bit octal digit.

The LNUM address points to a core location which has the 48-bit octal integer right-justified. The upper 12 bits are not used except for bit 58, which is used by the garbage-collect routine. Like BNUM's, LNUM's require 60 bits of storage.

iv. PNUM (Program NUMber)

Indicator: 1040B
Address: points to function definition word

A PNUM datum defines a machine-language subroutine. The address points to core location which has a function definition word. This word is divided as follows:

59 42	41 36	35 18	17 0
0	type	# of args	address

Type assignment is as follows:

type	function
1B	LSUBR
2B	FSUBR*
4B	FSUBR
10B	SUBR*
20B	SUBR

Only SUBR and FSUBR functions types use the #_of_args field to specify the number of arguments a particular machine function takes; the other types can take an indefinite number of arguments, and this field is zero (for more information on machine function types, see section I.6).

The address portion of the FNUM definition word holds the absolute address of the machine subroutine which actually performs the function.

v. ANUM'S -- (Arrays)

Arrays are also considered to be number tokens; they are called ANUM's when passed around as a data type.

Indicator: 1010B
Address: points to array list word

The arraylist word holds a pointer to the array in array space. Since the array itself is relocatable, all array references via the ANUM go indirectly through the arraylist word.

2.2.3 Strings

String data represents a compromise between compact storage of strings and ease of manipulation. Characters are stored at most 5 per word in free space, with a pointer to the next word in the string.

Indicator: 0004B
Address: points to the first string data word

Each string data word has from one to five 7-bit ascii characters, left-justified:

```
59                      23 22  18 17      0
+-------+-----+-----+-----+------+-----+--------+
|   C1  | C2  | C3  | C4  | C5   |  CT |ADDR    |
+-------+-----+-----+-----+------+-----+--------+
```

CT is the count of characters, and ADDR is the free-space address of the next string word.

2.2.4 Lists

A non-atomic pointer is a list pointer.

Indicator: 4000B
Address: points to the list word

The address points to a core location which holds a full ALISP word, that is, two ALISP pointers, one in the upper or CAR half, the other in the lower or CDR half of the word. These pointers may themselves be list pointers. A true linked list is formed by having the CDR pointer be a list pointer to another

ALISP word, and its CDR be a list pointer to another ALISP word, and so on; the last CDR pointer must be the NIL pointer. There is a simple correspondence between boxed list diagrams and list pointers: every arrow in a boxed diagram is a list pointer. Table 2.2 below gives an example of internal ALISP representation of a list structure.

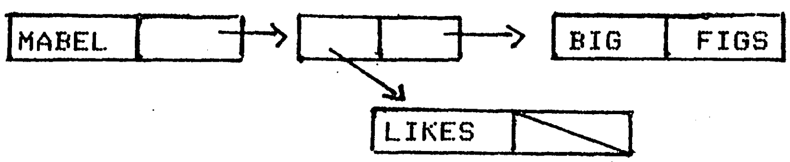
Two predicate functions are provided to distinguish between lists and atoms. ATOM, a SUBR of one argument, returns T if its argument is atomic, NIL if not. LISTP, also a SUBR of one argument, returns T if its argument is a non-atomic S-expression, NIL if not.

Table 2.2
Internal List Representation

Parenthesized expression:

(MABEL (LIKES) BIG, FIGS)

boxed diagram:



core representation (assume the following addresses for the atoms: MABEL=1, LIKES=2, BIG=3, FIGS=4), in base 8:

core location	contents
10	0000000001 4000000013
13	4000000017 4000000022
17	0000000002 0000000000
22	0000000003 0000000004

The list pointer for the whole expression would be:

4000000010B

I Chapter 3

Input Stream

ALISP does its own input stream handling, resurrecting the user from some of the deeper pitfalls in the KRONOS/NOS time-sharing system. This section describes the ALISP input stream and read functions in general, and the particular conventions for terminal input. For special batch and file characteristics of the input stream, see sections I.14 and I.15 in this manual. One change from normal LISP read syntax should be noted: the comma is used in place of the dot in dotted S-expressions. The dot is used solely in reading floating-pt. numbers.

3.1 Input Lines

The input stream is line-oriented, that is, it looks at only one line at a time. Lines are delimited from the terminal by a carriage return (CR), which ends the line and sends it to the input buffers in ALISP. Maximum line length, including control characters, is 150. If you type more than this, and then hit CR, the message *OVL* will be printed, and the line ignored.

3.1.1 INUNIT

The value of INUNIT is used by the read functions whenever a line must be input from somewhere. If INUNIT is set to the SNUM zero, a line is requested from the terminal. See sections I.15 and I.16 on batch and files for other values of INUNIT. Initial value of INUNIT is 0.

3.1.2 End-of-Line Processing

The CR character is normally appended to the end of the line on input. ALISP sees this character as a space (see section on STATUS below), but it can also be used to check for end-of-line or special input handling. A CR character is not tacked on the end of a line if the atom EOLR (End-Of-Line on Read) is set to NIL. This is useful for files or special input procedures where a character string runs past the end-of-line onto the next line, and a CR insertion would be undesirable. The only effect EOLR has on normal READ syntax is that literal atom, and number, and string tokens cannot be continued past the end-of-line. Initial value of EOLR is T.

3.1.3 PROMPT

The ALISP system normally signals that it wants input from the terminal by typing "Ø?" on the terminal and then waiting for the user to type a line (special prompt handling is taken for batch and file input--see section I.15 and I.16). The prompt character can be changed by resetting the value of the atom PROMPT, initially NIL. PROMPT should be set to the integer value of the character desired, i.e., (SETQ PROMPT 65) causes "A" to be used as the prompt character. For integer values to all characters, see Appendix A. Setting PROMPT to anything but an SNUM integer or NIL will give a SYN-ERR on the next terminal input request, and reset PROMPT to NIL.

There are two special cases for values of PROMPT. As noted above, NIL causes the characters "Ø?" to be used for the prompt. A zero value for PROMPT will have the same effect. If you wish to use a null character as a prompt, then set PROMPT to 32B rather than OB.

3.1.4 Input Line Editing

After a CR is typed at the terminal, the line just typed is transferred to the ALISP input buffers. Before any ALISP function sees this line, control characters within the line are used to edit it. The control characters are the same as those used in normal mode input to NOS 1.2 operating system: backspace, line feed, and escape.

Backspace deletes the most recent non-backspace character.

Line-feeds are deleted from the input stream.

Escape (control-shift-K on TT33, ATTN-ATTN on correspondence terminals) causes the entire line to be deleted, and a new line requested. The message "*DEL*" is printed on the terminal, and the prompt is re-issued. (Because ALISP uses transparent mode input, deleting a line takes longer than under normal input, since ALISP has to be rolled in). A control-C at the end of a non-empty line will also cause an escape; control-C on an empty line will cause an interrupt.

Input lines are edited before they are moved to the input buffer, so that ALISP sees only the edited input line. Lines inputted from files or ALISP batch are never edited.

3.1.5 TTYCHAR and Character Sets

ALISP can support a full ASCII character set, because it used a 7-bit internal character code There are times, however, when using the full set is cumbersome because of the shift required for upper-case alphabetic characters. This occurs on ASCII and correspondence terminals which have both upper and lower case alphabetics. For these terminals, typing "setq", for example, will not be the same as typing "SETQ", since the internal representation of upper and lower case is different. Most likely, "SETQ" was desired, since it corresponds to the internal print-name of the commonly-used atom SETQ; the user must resort to many upper-case shifts to input this atom. The remedy is to translate lower-case alphabetic characters directly to upper case before they reach the ALISP input buffer. This is done automatically if the atom TTYCHAR has a non-NIL value, as it does initially.

3.1.6 ECHO Control

The input line can be echoed on the current output device (OUTUNIT) by setting the value of the atom ECHO to the current input device (INUNIT). The echo is immediate, that is, no control character functions are performed, and the CR character is not tacked onto the end-of-line. ECHO has inital value of NIL in interactive mode: no echoing takes place. It is mainly useful under batch and in dealing with files (see the appropriate sections in this manual).

3.1.7 Problems with TELEX

Several mysterious behaviorial problems of TELEX deserve mention. ALISP operates using a TELEX input mode called transparent mode. What this means is that ALISP is responsible for handling all input line editing characters, and for outputting a prompt when it requests an input line. The keyword here is patience. Since ALISP does all of this input line processing, the user must wait at least the TELEX job response time in order to have things like escape (delete line) processed. Since this response time is typically on the order of 5 seconds, and can grow at times to above 20 seconds on a very busy system, a frustrated user might well wonder if his input line is being processed at all. To find out if the system is doing anything, the user has two commands at his disposal. Typing a simple CR will cause TELEX to print "JOB ACTIVE" if ALISP is running. Typing "STATUS" will print the current TELEX status of the ALISP job. Any other commands typed at this point will cause TELEX to respond with the message, "ILLEGAL COMMAND".

Another problem occurs frequently on a busy system. In this case, ALISP will print out the input prompt. When the user responds by typing in a line and pressing CR, TELEX prints the reply, "ILLEGAL COMMAND". What has happened is that there is a slight time lag between the printing of the prompt and issuing of an input line command from TELEX; consequently the user is talking to TELEX rather than ALISP for that short lag. On very busy systems this lag can be several seconds long. TELEX will eventually get around to issuing the ALISP input request; wait for a question mark prompt (distinguishable from ALISP's prompt because it has no preceeding blank). A line can now be entered to ALISP. This line is not entered in transparent mode, however, but normal TELEX mode, so that only a restricted character set can be received by ALISP, no matter what the value of TTYCHAR. Note finally that this problem will not occur with correspondence terminals, since the keyboard will not unlock until the ALISP input request is actually issued.

A final problem occurs with the trapping of interrupts. There is a short time span immediately after a CR is typed to end an input line, when the interrupt (control-C or ATTN-S-ATTN) will not be trapped by ALISP. If an interrupt is given during this time, TELEX will abort the ALISP run with the *TERMINATED* message. Again, the problem is worse on very busy systems.

3.2 READ Structure

READ, a SUBR of no arguments, is the primary function used in forming the ALISP input syntax. This function reads characters from the input buffer, forming them into internal ALISP S-expressions. Associated with the input buffer are character positions pointers (see section 3.3 below). READ takes characters from the input buffer until it forms a complete S-expression; if end-of-line is reached before the S-expression is completed, another line is requested from the current INUNIT. It also advances the buffer pointers past this S-expression, so that subsequent READ's will return successive S-expressions from the input buffer. Thus, more than one S-expression can be READ from a single input line. Note, however, that the top-level loop of the supervisor uses READENT rather than READ, and READENT returns at most one S-expression per input line (see section 3.2.4).

An S-expression may also extend over several input lines. READ automatically requests new lines until the S-expression is completed. This has caused some users heartache because they think they are in an infinite input request loop. They keep getting input requests on the terminal, and type in S-expressions, and get more input requests, without anything else happening. What has happened is that the user forgot to close a left-parenthesis in his first S-expression, and READ keeps asking for lines until he closes it. The remedy is to type many right-parentheses if you think you are in this type of loop.

3.2.1 STATUS

Associated with each character is a 3-bit integer called its STATUS, used by the READ function to decide syntax. The initial STATUS of all characters is given below in Table 3.1.

Table 3.1
STATUS of Characters

type	STATUS	Characters
alphabetic	0	A to Z and all chars. except those below.
numeric	1	0 to 9 and -.#
separator	2	␢ and CR
lpar	3	(
rpar	4	)
dot	5	.
slash	6	/
macro	7	"'$;@

The status of characters can be fetched or changed using the function STATUS, a SUBR* of one or two arguments. The first argument should be an nslitat (non-NIL, non-GENSYM literal atom); the first character of this atom's pname is used by STATUS. If there is only one argument, STATUS returns by character's status as an SNUM from 0 to 7. If the second argument is present, it must be an SNUM; the lowest three bits of this SNUM integer are used to set the new status of the character and STATUS returns an SNUM giving the previous status of the character. Several examples of the STATUS function are given in Dialogue 3.1 below. The STATUS function should be used with care, as it can drastically redefine the action of the READ syntax.

3.2.2 READ Syntax

This is the standard ALISP syntax for inputting strings of characters and assembling them as LISP data types. To some extent this standard syntax can be manipulated with user-defined READ macros (section 3.2.3) and the STATUS function. The alternative to READ syntax is single-character manipulation (see section 3.3).

If an incorrectly formatted character string is given to READ (e.g., an alphabetic character in a numeric string), then READ will complain by issuing a SYN-ERR, which is a fatal error. If you are at the top level of ALISP, this simply means you will have to retype the entire S-expression you were inputting. The SYN-ERR also prints a message telling what caused the error, and the character position in the input buffer at which the error

Dialogue 3.1
The Function STATUS

```
?(STATUS 'Z)
0
```

Fetch the current STATUS of "Z".

```
?(STATUS 'Z 2)
0
```

Change the STATUS of "Z" to 2, a separator (blank). Zero is the previous STATUS of "Z".

```
?'FOOZBAR
FOO
```

Since "Z" has the STATUS of a separator character, only the first three characters of the line were assembled into an atom, FOO.

```
?(STATUS '/, 0)
```

This defines the character "," as an alphabetic.

```
?(CDR '(FOO , BAR))
(, BAR)
```

Now "," no longer acts as the dot of dotted S-expressions, but as a pname character. Note that, so long as no character has a dot STATUS (5), no dotted S-expressions can be inputted.

---

occurred. Syntax errors for particular data types are noted below.

Almost all ALISP data types can be read in using READ. READ scans the input line for the first non-separator (STATUS not 2) character, and uses this character to determine the syntax of the rest of the string. If the first non-separator character is:

a. Alphabetic - assemble literal atom.

Characters are fetched from the input stream to form a print-name (pname) until a character which is neither alphabetic or numeric (STATUS greater than 2) is encountered. The pname is then internalized on the OBLIST and the pointer is returned. Pnames of more than 322 characters cannot be assembled, but cause a SYN-ERR.

Atoms having pnames beginning with numeric characters, or containing characters with STATUS other than alphabetic or numeric, are called exotic atoms. Exotic atoms can be read in with the help of a slash convention. Any character with a slash status (6) is skipped in forming the pname, but the character immediately following, no matter what its status, is interpreted as alphabetic. A slash can be put in the pname by using two consecutive slashes. All terminal characters except LF (line-feed) and backspace can be inputted into pnames in this way.

Note that atom print-names cannot normally be continued past the end of a line when EOLR is non-NIL, since the CR character tacked onto the end of the line is interpreted as a space (STATUS = 2). However, if a slash is the last character typed before the CR, the CR character will be assembled into the pname, and the pname string will continue onto the next line. Examples of pname formation are given in Dialogue 3.2 below.

Dialogue 3.2
Pname Formation

input stream	assembles to
ƀƀƀFOOƀƀ...	atom with pname "FOO"
ƀƀƀFOO2ƀƀ...	atom with pname "FOO2"
ƀƀƀ2FOOƀƀ...	SYN-ERR, first character not alphabetic
ƀƀƀ/2FOOƀƀ...	atom with pname "2FOO"
ƀƀƀFO///ƀOƀƀ...	atom with pname "FO/ƀO"

---

b. Numeric - assemble numeric atom

There are three types of numbers which can be assembled by READ:

i. LNUM - Logical NUMber.
   16-digit octal integers
ii. SNUM - Small NUMber.
   integers in the range -2B17 + 1 to 2B17 - 1
iii. BNUM - Big NUMber.
   floating-point numbers in the range ±10E-298 to ±10E312

SNUM's and BNUM's (but not LNUM's, which always use an octal base) are assembled with reference to the base contained as the value of the atom INBASE. The value of INBASE must be an LNUM (thus making it independent of INBASE or OUTBASE); it is initially set to 12B (base 10) but can be reset by the user to any value from 2B to 20B (base 2 to base 16). A SYN-ERR is issued on the next number assembly if INBASE is set to an illegal value, and INBASE is reset to 12B.

i. LNUM's

They are signalled by an initial number-sign ("#"). Format is:

```
#sdddd....
```

where

s is an optional minus sign ("-") which complements the assembled octal integer
d are octal digits, up to 16 of them.

The assembled number is:

```
dddd...B
```

SYN-ERR caused by:

1. 8,9, or A-Z appearing in the string
2. More than 16 digits in the string

Examples:

input stream	number assembled
ØØ#222ØØ...	0000000000000222B
Ø#-222ØØ...	7777777777777555B
Ø#ØØ...	0000000000000000B
Ø#-0ØØ...	7777777777777777B
Ø#923ØØ...	SYN-ERR, 9 is not an octal digit.

ii. SNUM's

Input format is:

```
sdddd....
```

where

s is an optional minus sign ("-")
d are digits from 0 to INBASE-1

The digits are assembled into an integer using the INBASE base representation.

SYN-ERR caused by: Alphabetic character in the string

Examples (assume INBASE set to 12B):

input stream	number assembled
Ø18ØØ...	18
Ø-18ØØ...	-18
Ø-0ØØ...	0
Ø000ØØ...	0
Ø18636302222ØØ...	converted to floating-pt

Note that negative zero is read in as positive zero, and that integers out of SNUM range are automatically converted by READ to BNUM's. Largest magnitude for an SNUM is 2B17 -1.

iii. BNUM's

BNUM's are signalled by the presense of the decimal point "." or the exponent marker "E", or if an SNUM formatted number is too large in magnitude to be an SNUM. BNUM's are assembled using the INBASE input base. Format is:

sdd.dd...Eseee...

where

s is an optional minus sign ("-")
d is a digit, from 0 to INBASE-1
E is an optional exponent marker
e are optional exponent digits, from 0 to INBASE-1
. is an optional "decimal" point

The assembled number is formed by assembling the coefficient digits d into a floating-pt. number using the base INBASE, then multiplying this coefficient by INBASE raised to the assembled exponent power. The exponent digits e are also assembled into an integer using INBASE representation.

SYN-ERR caused by:

1. Alphabetic character (except "E") in the string
2. More than one decimal point or "E"
3. A decimal point in the exponent
4. Exponent too large or small

Examples (assume INBASE set to 12B):

input stream	number assembled
Ø18.ØØ...	18
Ø.18E2ØØ ...	18
Ø-.18E0ØØ...	-.18
Ø0.0ØØ...	0

Note that the presence of an "E" or "." distinguishes BNUM's from SNUM's; even in the case of 0.0, a BNUM is assembled. Internal accuracy for BNUM's is 47 bits in the coefficient, or about 14 decimal digits.

d. lpar - assemble non-atomic S-expression

When the READ function encounters an initial left-parenthesis (STATUS of 4), it assembles a complex LISP structure. Format is:

(string1 string2....stringn,stringm)

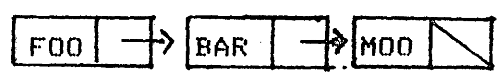
Each stringi is any character string which assembles into a valid ALISP expression, even another list string. The comma is an optional dotted-pair indicator; if it is present, stringm rather than NIL will be stuffed into the CDR of the last word being assembled, so that a general non-atomic S-expression, rather than a list, will be assembled. The comma character must always appear just before the last string, since its stuffs the last assembled string into the CDR of the last assembled S-expression word. If the lpar is immediately followed by an rpar, NIL (an empty list) is assembled.

Examples of READ list assembly are given in 3.3.

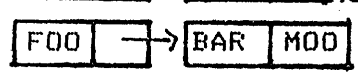
Dialogue 3.3
List Assembly

input stream — assembles to

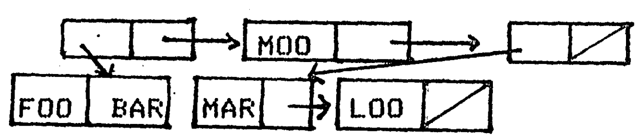
Ø(FOOØBARØMOO)Ø...



Ø(FOOØBAR,MOO)Ø...



Ø((FOO,BAR)ØMOO(MARØLOO))Ø...



e. Macro - evaluate macro expression

The use of READ macros is explained more fully in section 3.2.3. The initially defined ALISP macro characters are five: ' " ; $ @ (single quote, double quote, semicolon, dollar-sign, and at-sign. Do not use them as single-character atom pnames, as it will conflict with their macro usage (for instance, do not use the atom with pname ' as a lambda-expression variable). Their effects are:

i. Quoting -- '

The ' character is used when a quoted expression must be assembled from the input stream. The effect of the quote character, when encountered by READ, is to cause the next string in the input stream to be assembled (by a recursive call to READ), and then used as part of a QUOTE expression. The following examples should make this clear:

```
input stream                    assembles into

ƀ'FOOƀƀ...                      (QUOTE FOO)
ƀ'(FOOƀBAR)ƀƀ ...               (QUOTE (FOO BAR))
ƀ(FOOƀ'BARƀMOO)ƀƀ...            (FOO(QUOTE BAR)MOO)
ƀ(FOO'BARƀMOO)ƀ ƀ...            (FOO(QUOTE BAR)MOO)
```

ii. Commenting -- ;

The ; character is used to add comments to the input stream. When the ; character is encountered by READ, it causes the rest of the line to be ignored. Note that pname or numeric strings cannot be continued beyond the comment character onto the next line. Examples:

```
input stream                          assembles into

(FOO ; THIS IS A COMMENT
       BAR)                             (FOO BAR)
(FOO ; 2ND COMMENT
            ; 3RD COMMENT
              BAR)                      (FOO BAR)
FOO;                                    (FOO BAR)
BAR)
```

iii. Immediate evaluation -- $

The $ character is used to evaluate expression assembled from the input stream before adding them to the READ result. When the $ character is encountered by READ, it causes the string immediately following to be assembled and evaluated (using EVAL) before inserting it into the result. Examples:

```
 input stream                      assembles into

(FOO$(LIST 'MOO 'MAR)BAR)   (FOO (MOO MAR) BAR)
   $'FOO                           FOO
   $''FOO                          (QUOTE FOO)
```

The $ macro can be used very handily to add longer S-expression to the input stream without having to re-type them each time. For example, suppose the atom FOO is set to a long list. To insert that list into the input stream at any given point, $FOO is all that's needed.

iv. Read string -- "

The " character is used to assemble string data from the input stream. Characters are read from the input buffer and assembled into a string until another " character is encountered. A double quote may be included in the string by using two successive double quotes.

Examples:

```
   input stream        assembles into string

   "ABCD"                    ABCD
   "AB""CD"                  AB"CD
   "ABC)D[CR]                ABC)D([CR]EFG
   EFG"
```

v. Read array -- @

Arrays are assembled when the @ character is encountered in the input stream. See the chapter on arrays for the exact format.

3.2.3 READ Macros Explained

The macro facility on input is a most valuable method for custom-tailoring input syntax within the READ syntax structure. This section explains the action of READ macros, and how to implement them.

Two properties define a valid macro character -- a character STATUS of 7, and a valid function definition stored in the value cell of an atom with that single macro character as its pname. The five pre-defined macro characters, ' " ; $ @, are all defined as FSUBR's. User defined macros must be lambda expressions of no variables (either FLAMBDA or LAMBDA will do). When the macro character is encountered by READ in the input stream, its function definition is fetched and evaluated, and the result placed in the appropriate part of the result being assembled by READ. Since the macro function itself can call READ, some very clever things can be done. For example, try to re-define the ' macro using a lambda-expression. One way would be:

```
(DE /' NIL (LIST (QUOTE QUOTE) (READ))
```

Now the atom with pname ' has the above function definition. Take a sample input stream:

```
...ØØ(FOOØ'BARØMOO)ØØ..
```

READ starts to assemble a list when it hits the left parenthesis. It assembles the atom FOO as the first element of the list. READ is now at this point:

```
input stream                      assembled result

..ØØ(FOOØ'BARØMOO)ØØ..
         ^
input buffer pointer
```

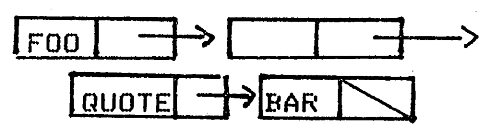
The macro character ' is next encountered and READ looks up its function definition and evaluates it as a function of no arguments. It evaluates:

```
(LIST (QUOTE QUOTE) (READ))
```

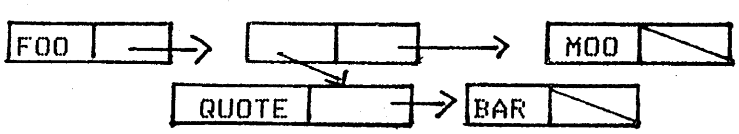
The READ call returns the next S-expression in the input stream, namely, the atom BAR. Then the LIST function gives (QUOTE BAR) as its result, and this is the result which the macro function returns. READ takes this result and inserts it as the next element on the list it is assembling. READ is now at this point:

```
input stream                      assembled result

..ØØ(FOOØ'BARØMOO)ØØ..
           ^
  input buffer pointer
```



Finally, MOO is read as the last element and the list is completed. The final result is:



This is same as if the input stream had been:

```
..ØØ(FOOØ(QUOTEØBAR)ØMOO)ØØ...
```

READ macros are a very powerful tool for manipulating the input stream; the inventive ALISPer will find many uses for them in front-end translators for his programs. Unfortunately making read macros work as intended is tricky and most LISP'ers require some time before they become proficient.

3.2.4 TEREAD and READENT

The input line can be flushed using the function TEREAD, a SUBR of no arguments. Evaluating (TEREAD) causes the input buffer (see 3.3) to be blanked, and a new line requested from the current input device. The value of TEREAD is NIL.

The function READENT is a SUBR of no arguments. It does a TEREAD and then a READ, returning the result of the READ as its value. READENT is used when it is desirable to READ at most one S-expression per input line.

3.3 Input Buffer Pointers

The input buffer is directly accessible from ALISP at the single-character level. The buffer pointers are the values of the atoms READBEG, READLEN, and READEND. The values of these atoms are SNUM integers designating character positions in the input buffer. (The first character position in the input buffer is at position zero.) Whenever a new input line is read in, the buffer pointers are reset as follows:

> READBEG is the first character position in the input buffer that will be read. Initial value for READBEG is zero, so that all characters in the input buffer are used. If READBEG is set larger than zero, initial characters in the input buffer will be skipped. Setting READBEG negative or greater than READEND causes a NUM-ERR, and resets READBEG to zero.
>
> READEND is set to the number of characters in the line when it is read in including the CR character, if there is one. If EOLR is non-NIL, this will be 1 + the number of characters typed in. Inputting a null line by hitting just a CR will thus set READEND to 1 (if EOLR is NIL, READEND is set to zero).
>
> READLEN is set to the value of READBEG when a new line is read into the input buffer. READLEN is the current character position used by all the read functions. If READBEG is zero, READLEN is initially set to the beginning of the input line. Reading a character causes READLEN to be incremented to the next character. When READLEN=READEND, the last character

has been read from the line, and subsequent read requests will cause an automatic TEREAD to be performed, resetting the input line.

The input line pointers allow single-character control over the input stream. The length of the current input line as well as the current character position within the line can be extracted from them; and they can be reset with SETQ to skip or back up over characters in the input line. In conjunction with the functions described in the following section, explicit single-character control over the input stream is possible.

3.3.1 Single-character Read Functions

The functions described below read and return single characters from the input line. They are all functions of no arguments. The character is read from the current position of READLEN, and READLEN is incremented to point to the character after the one read (except for READPK). If READLEN=READEND when the function is called, an automatic TEREAD is first executed, and the function reads from the next input line.

READCH reads the next character from the input stream, and returns an atom whose pname consists of that single character.

READNM read the next character from the input stream, and returns its internal integer representation as an SNUM. "A" would be 101B, "B" would be 102B, etc. See Appendix A for internal character codes.

READNB keeps reading single characters from the input line until a non-blank (STATUS ≠ 2) character is found or until the end-of-line is encountered. It returns an atom whose pname is that single non-blank character or NIL if the end-of-line was found first.

READPK reads the next character from the input stream, but does not advance the READLEN pointer. It returns an atom whose pname is the single character read. If READLEN=READEND no TEREAD is called, and READPK returns NIL.

I Chapter 4

Output Stream

This section describes the general characteristics of the ALISP output stream, as well as specific time-sharing characteristics (for special batch and file considerations, see sections I.14 and I.15).

4.1 Output Lines

The output stream, like the input stream, is line oriented, that is, it looks at only a single line at a time. Lines automatically have an end-of-line byte tacked onto the end when they are written to an output device. Maximum line length is 150 characters; the actual printed line length (before an end of line is automatically transmitted) is given by the current value of the atom PRINEND (initially, PRINEND is set to the SNUM 72).

4.1.1 OUTUNIT

The current value of OUTUNIT is used by the print functions whenever a line must be output somewhere. If OUTUNIT is set to the SNUM zero, lines are printed on the terminal. See the sections on batch and files for other values of OUTUNIT.

4.1.2 Character Sets

If you have signed on to the terminal correctly, then all characters which are available on the terminal type element will print correctly (even the APL typeball is compatible). See Appendix A for characters available from different terminals.

ALISP uses full ASCII mode when communicating with the terminal, so that problems caused by the 64 character set in KRONOS or NOS are minimized.

4.1.3 End-Of-Line Processing and TERPRI

The atom PRINEND contains the line length for the output line (maximum is 150 characters) When the output buffer is full, it is dumped to the current output device. An end-of-line byte is tacked onto the end of each line as a line delimiter. There are actually two types of end-of-line bytes; which one is used is controlled by the atom EOLW (End Of Line on Write). If EOLW is non-NIL, the end-of-line byte causes a carriage-return line-feed (CR-LF) when the line is output, resetting the terminal carriage to the beginning of the next line. If EOLW is NIL, the end-of-line byte has no effect on the terminal printing, and the carriage is left where it stopped after printing the last character. Normally, the CR-LF end-of-line byte is the one that's wanted, and thus the initial value of EOLW is T. For some special applications, such as output formatting or control of graphic devices, it is necessary that an end-of-line not print the CR-LF, and for this purpose EOLW should be set to NIL.

The function TERPRI, a SUBR of no arguments, is provided to end an output line and dump it before the PRINEND limit is reached. Evaluating (TERPRI) terminates the output line and dumps the output buffer. If there was nothing in the buffer, an empty line is outputted. For further considerations on output line formatting, see section 4.3 below.

4.2 PRINT Structure

The function PRINT, a SUBR of one argument, provides communication between internal ALISP structures and the output line. In general, any ALISP expression that can be read with READ (and a few that can't, also) can be printed by PRINT, in a format compatible with the original READ syntax. When PRINT finishes outputting its argument, it issues a TERPRI to dump the final output line.

4.2.1 PRINT Syntax

Intimate knowledge of PRINT syntax is really not very important, unless one is concerned with output formatting or files. Still, a knowledge of the PRINT syntax enables the informed user to know exactly what internal ALISP structures are represented by the output.

a. List Structures

A list (or non-atomic S-expression in general) is printed as a series of elements between parentheses. This format is recursive, that is, if an element of a list is itself a list, it too is printed as a set of elements between parentheses. Between every element of a list a space is inserted. If the final CDR of the S-expression is non-NIL, a comma is printed, and then the final CDR element. Note that the only place a comma will appear is just before the last element of an S-expression.* Examples of list structure print format

---

* The comma is used instead of a dot in dotted S-expressions.

are given below in Dialogue 4.1.

Dialogue 4.1
List Structure PRINT Format

S-expression			Print as
FOO	BAR		(FOO BAR)
	MOO	BOO	MAR
			((FOO,BAR) MOO BOO,MAR)
FOO BAR			

b. Literal atoms

i. The atom NIL prints as "NIL".

ii. Gensym atoms print as "Xdddd....", where X is the gensym character (see GENCHAR) and the d are gensym digits, unique to each gensym atom. Gensym atoms cannot be read back into the system; their print characters are only to enable the user to identify them on output.

iii. All other literal atoms use their print-names (pnames) to form a printable character string. This string is normally exactly that used to input the atom with READ, e.g., "FOO" read in will print as "FOO". A problem arises with exotic atoms, however (exotic atoms contain pname characters with STATUS $\geq$ 2). Since exotic atoms are inputted using slash convention, they will not look the same on output, when no slashes are present. This is usually what is desired, though, since terminal output cannot be re-inputted directly anyhow. For files which can be read back in, however, it would be nice if exotic atoms could by output and read back in properly. To this end, the switch SLASHES is provided. If it is set to NIL, no slashes will be inserted in exotic atom pnames when they are

---

Because there is no confusion with the dot as used in floating-point numbers, there is no need to put spaces around the comma.

printed; if non-NIL, slashes will be inserted at the correct positions, to enable the atom to be read back in (this does not solve all problems with exotic atoms -- see the section on files). Initial value of SLASHES is NIL. Examples of literal atom printing are given in Dialogue 4.2

Dialogue 4.2
Literal Atom Printing

?(GENSYM) G0 ?(SETQ GENCHAR 'A) A ?(GENSYM) A1	GENSYM atoms have the default character of 'G' in their pnames, unless GENCHAR is set non-NIIL.
?() NIL	NIL prints as "NIL".
?SLASHES NIL	SLASHES is initially set to NIL.
?(SETQ FOO'/2BAR) 2BAR ?'THIS/ØMESS THISØMESS	No slashes on output.
?(SETQ SLASHES T) T	SLASHES set to T
?FOO /2BAR ?'THISØMESS THISØMMESS	Slashes now appear in the outputted pnames, as they were when inputted.

---

c. Numeric atoms

Numeric output format is completely compatible with input format for LNUM's, BNUM's, and SNUM's. Thus, if they are written to a file, they will be read back in correctly. The type of numeric atom represented is apparent from the format.

Both SNUM's and BNUM's use a variable base representation on output. The value of the atom OUTBASE must be an LNUM; this LNUM is the base representation on output. Initially, OUTBASE is set to #12 (base 10). Legal values for OUTBASE are from #2 (base 2) to #20 (base 16).

i. LNUM's

Output format is:

#sddd....

where

s is an optional minus sign ("-") which is used if the left-most bit of the LNUM is set.

d are the the octal digits representing the 48-bit LNUM (up to 16 digits). If the minus sign is present, these digits represent the one's complement of the LNUM bits. Left zeros are suppressed; if all bits are zero or one, "#0" and "#-0" are output respectively.

ii. SNUM's

Output format is:

sddd...

where

s is an optional minus sign ("-").

d are digits representing the SNUM integer. The integer printed uses OUTBASE for its base representation. Leading left zeros are suppressed; zero always prints as "0".

iii. BNUM's

Output format is:

s.dddd....Eseee...

where

s is optional minus sign ("-").

d are digits of the coefficient. They are output using OUTBASE representation.

E is an exponent indicator (always present).

e are exponent digits. They are also output using the OUTBASE base representation.

The number of digits d in the coefficient of a printed BNUM can be controlled by the atom DIGITS. The value of DIGITS should be an SNUM positive integer indicating the number of digits desired. All digits asked for are printed, so that right trailing zeros are not suppressed. Rounding is done on the DIGITS + 1 digit to make the result more readable. If DIGITS is set to 0 all significant digits will be printed (14 or 15 for base 10), and no rounding will take place. DIGITS is initially set to 13. Note that digits does not control the total length of the BNUM output, just the number of significant digits in the coefficient. Neat formatters must resort to other tricks.

iv. PNUM's

This number type is used internally by the interpreter, and cannot be read into the system. Nevertheless, there are times when a PNUM will sneak into an S-expression being output, so the user may as well know what he's got. A PNUM is the value of an atom which has a machine-language function definition (SUBR, LSUBR, etc.) such as SETQ or CONS (for more specific information, see the section on ALISP data types). Format is:

P#ddd...

where

P# is the PNUM indicator.
d are octal digits representing the function type and its machine address.

v. ANUM's

This is a number type which cannot be read back in by READ. An ANUM is an internal array pointer, and can only be created by the function ARRAY (or the @ real macro). Format on output is:

A#nnnn

where oooo is the octal address of the arraylist word for the array. The file package will print arrays specially so that they can be read back in, rather than printing ANUM's. The functions PRINARRAY and READARRAY are available to the user for storing his own file I/O with arrays (see section I-12 on arrays).

4.2.2 PRINT Syntax Functions

PRINT is the most commonly used printing function. It is a SUBR of one argument; it prints its argument according to the PRINT syntax just described, then issues a TERPRI to flush the output buffer. The value of PRINT is its argument.

PRIN1 is just like print except it issues no TERPRI. The difference between PRINT and PRIN1 can be seen in Dialogue 4.3 below. The value of the function PRIN1 is its argument.

The function HALFPRIs is used to print out part of a long list. It is a SUBR of one argument which acts just like PRIN1, except it will only output a limited number of atoms in a long list. The limit is fixed by the atom HPRNUM, which should be set to an SNUM. If HPRNUM is not an SNUM, the default value 4 is used (this is the initial value of HPRNUM). If a list with more than HPRNUM atoms in it is given to HALFPRI, it will correctly print the list up to the first HPRNUM atoms, then print an elipsis "...", and close all parentheses in the list. Examples of HALFPRI calls are given in Dialogue 4.3.

TERPRI is a SUBR of no arguments, which terminates and dumps the output buffer. Value of TERPRI is NIL.

4.3 Unified Output Buffer

Like the input buffer, the output buffer is directly accessible from ALISP on the single-character level. Three buffer pointers control the buffer flow: PRINBEG, PRINLEN, and PRINEND; they must all have SNUM values.

PRINBEG is the first position to begin stuffing characters into the output buffer. Initially, PRINBEG is set to 0, i.e., the leftmost character position in the buffer. If PRINBEG is set negative or greater than PRINEND, a SYN-ERR will be issued at the next output buffer flush, and PRINBEG will be reset to 0.

PRINLEN is the current position to stuff a character into the output buffer. When the buffer is

Dialogue 4.3
Print Functions

```
 ?(PROGN (PRINT 'FOO)(PRINT 'BAR) NIL)
FOO
BAR
NIL
```

PRINT terminates the output line on each call.

```
 ?(PROGN (PRIN1 'FOO)(PRIN1 'BAR) NIL)
FOOBARNIL
```

PRIN1 does not terminate the output line, so successive calls are strung together. Note that the result PROGN, NIL, was tacked onto the end of PRIN1's output. If the number of characters in the output buffer exceeds PRINEND, the output buffer is dumped, even if PRIN1 is doing the printing.

```
 ?HPRNUM
NIL
 ?(HALFPRI '(A B C D E F))
(A B C D ...) (A B C D E F)
```

HPRNUM is set to NIL, so only 4 atoms are outputted on a call to HALFPRI.

Note that HALFPRI, like PRIN1, does not dump the output buffer after it prints, so that the value of HALFPRI immediately follows its output.

```
 ?(SETQ HPRNUM 10)
10
 ?(PROGN (HALFPRI '(A B C D E F))
 ? (TERPRI))
(A B C D E F)
NIL
 ?
```

Since HPRNUM was 10, all of the atoms in the argument of HALFPRI was printed.

---

emptied, PRINLEN is set to PRINBEG, the first available position. Stuffing a character into the buffer causes a PRINLEN to advance by one, until it reaches PRINEND. If a character is to be stuffed when PRINLEN=PRINEND, the output buffer is first flushed to the current output device, PRINLEN reset to PRINBEG, and then the character is inserted into the buffer. Explicitly setting PRINLEN greater than PRINEND gives a SYN-ERR on the next buffer operation, and resets PRINLEN to PRINBEG.

PRINEND is the last position of the output buffer; it should not be set larger than 150, or a SYN-ERR will be issued, and PRINEND reset to 72. Initial value for PRINEND is 72.

The output buffer pointers are all used and updated by the printing functions; the user can change their values explicitly by using SETQ (or SET or QSETQ).

Three hints on the use of these pointers. If PRINBEG is greater than 0, then the character positions before PRINBEG are filled with blanks. Thus if PRINBEG is set to 2, all outputted will start with 2 blanks.

When the output buffer is flushed, all character positions are filled with blanks. Thus advancing PRINLEN as a character position (using SETQ) without stuffing anything into that position, will cause it to print as a blank. Also, if PRINLEN is set back to a buffer position that already has a character stuffed into it, a new stuff will replace the old character with the new.

Finally, note that a buffer flush (using TERPRI) takes the value of PRINLEN as the end of line position, so that only the first PRINLEN characters in the whole buffer are outputted. This is cool unless you reset PRINLEN to a previous character position (in order to replace a character, say), and then do a TERPRI. Only the characters up to the PRINLEN position will be output, everything past that is lost.

4.3.1 Character Printing Functions

These functions provide the ability to send individual characters to the output buffer. All of them use and update the buffer pointers.

PRINB is a SUBR of one argument. (PRINB x), where x is an SNUM, packs x blanks into the output buffer. If x is zero or negative , no blanks are outputted. Returns NIL for its result.

PACK1 is a SUBR of one argument. (PACK1 x), where x is an SNUM, sends the single ASCII character represented by the integer x to the output buffer. See Appendix A for integer representations of characters. If x is larger than 177B, it is truncated to provide a 7-bit integer. Some caution should be used for odd values of x; for instance, certain integers do not represent any characters, and will not print.

I Chapter 5

Literal Atom Structure

This section describes the attributes and internal representations of literal atoms, as well as the methods used to internalize and update the literal atom object list (OBLIST).

5.1 OBLIST

The OBLIST is an internal hash array of 128 buckets holding all non-NIL, non-gensym literal atoms in the system (except for literal atoms which have been WIPE'd, see section 5.1.2 below). Each bucket is a list of atoms corresponding to its particular hash. Whenever a literal atom is read in (using a READ syntax function), its pname is hashed and the appropriate bucket searched to find a match. If none is found, a new entry is created on the bucket.

The OBLIST can be explicitly retrieved using the function OBLIST, a SUBR of no arguments. It returns a list of all 128 atom buckets. OBLIST actually returns a copy of the internal hash buckets, so that its result can be manipulated with impunity by RPLACA, RPLACD, NCONC or any other permanent list-altering function, without fear that the OBLIST will be wrecked and bomb out at the next garbage-collect.

An atom's position on the OBLIST can be obtained with the function ATMHASH, a SUBR of one argument. ATMHASH returns the bucket number of its argument as an SNUM from 1 to 128; the first bucket on the OBLIST is bucket 1, the last is 128. If ATMHASH is given anything but an nglitat argument, it will complain with an ARG-ERR. If it cannot find its argument on the OBLIST, it returns NIL. An example of the ATMHASH function is given in Dialogue 5.1 below.

5.1.1 Truly Worthless Atoms

Also called TWA's for short, these are non-NIL literal atoms which have no other attribute than a pname, and are not referenced by any other dat structure in the ALISP system. TWA's are purged from the OBLIST at the next garbage collect, unless they have been set with the function SPECIAL (see below). Clearing out TWA's has the effect of freeing up free storage and unclogging the OBLIST, especially when large numbers of new literal atoms are created and then abandoned during the course of program execution.

If for some strange reason (such as freeing up more storage) you desire to turn a worthy atom into a TWA, then you must remove its value attribute with REMOB and set its plist to NIL with the function PLIST. If there are no references to this atom in list structures or other atom values, then you have created a TWA.

There are times when it is desirable to keep a literal atom around even when it is not being actively referenced, or has no value or plist attributes. The function SPECIAL is used to mark an nlitat so that it will not be garbage-collected even if it is a TWA. This function is a SUBR* of one or two arguments. The first argument is the nlitat to be set or queried for SPECIAL status. The second argument is optional; if absent, the SPECIAL status of the atom is returned as T or NIL. If present, the SPECIAL status of the atom is set if it is non-NIL, cleared if NIL. Value is the atom. Examples of the SPECIAL function are given in Dialogue 5.1 below.

All system atoms such as SETQ, CONS, etc. have their SPECIAL status set so they won't get clobbered even if you turn them into TWA's. You can, if you wish, intentionally destroy the system by an overt act of un-SPECIAL'ing and cloberring a vital atom such as PRINLEN. The atom NIL cannot be SPECIAL'ed.

5.1.2 WIPE

In some cases it is desirable to make a literal atom invisible to the READ functions. This is true, for example, if you have a large program which uses local variables, and does some READ calls; if nothing in the READ should conflict with the local variables, the locals can first be WIPE'd. WIPE'ing a literal atom removes it from the OBLIST and places it on the WIPELIST, where it will be garbage-collected correctly but not looked up by READ when an atom with a similar pname is inputted. All other attributes of the atom remain as they were. In this way it is possible to create two different nglitats with the same pname.

Literal atoms on the WIPELIST are not unique. If the atom FOO, for example, is WIPE'd onto the WIPELIST, then another atom with the same pname FOO is put onto the OBLIST, WIPE'ing the OBLIST atom FOO will put this new atom on the WIPELIST. There will then be two atoms with the same pname on the WIPELIST. By successive applications of WIPE, an indefinite number of atoms with the same pname can be put onto the WIPELIST; they will all be different internal atoms, and not EQ to each other.

Once an atom is WIPE'd it cannot be put back on the OBLIST. If an atom on the WIPELIST is a TWA (section 5.1.1 above), then

Dialogue 5.1
The Function SPECIAL

?(LIST 'FOO 'BAR) (FOO BAR)	The atoms FOO and BAR are TWA's, since they have no value or plist attributes, and their only reference, in (LIST 'FOO 'BAR), is lost as soon as it is evaluated.
?(ATMHASH 'FOO) 62 ?(ATMHASH 'BAR) 120 ?(ARGN (OBLIST) 62) (* FOO) ?(ARGN (OBLIST) 120) (SUBR* RETURN BAR)	ATMHASH returns the hash bucket on the OBLIST which holds its arguments. Both FOO and BAR are on the OBLIST, since no GC's have been performed yet. ARGEN is a function which returns the other element of a list.
?(SPECIAL 'FOO) NIL ?(SPECIAL 'BAR) NIL	The SPECIAL status of FOO and BAR is NIL.
?(SPECIAL 'FOO T) FOO	This sets the SPECIAL status of FOO to T.
?(GC) NIL ?(ARGN (OBLIST) 62) (* FOO) ?(ARGN (OBLIST) 120) (SUBR* RETURN)	The GC function calls an immediate garbage-collect. The atom FOO is still on the OBLIST, but BAR has disappeared.

it is removed from the WIPELIST at the next garbage collection.

The function WIPE is a SUBR of one argument. This argument should be an nglitat to be removed from the OBLIST. If the atom is not on the OBLIST, or is a GENSYM atom or NIL, no action is taken. WIPE returns its argument if it was successful in WIPE'ing it, or NIL if it was not. Examples of the WIPE function are given below in Dialogue 5.2.

Dialogue 5.2
The Functions WIPE and WIPELIST

?(SETQ FOO '(BAR BOO MOO)) (BAR BOO MOO)	The value of the atom FOO is the list (BAR BOO MOO).
?(EQ (CAR FOO) 'BAR) T	The first element of the list, the atom BAR, is on the OBLIST and EQ to the atom BAR just read in.
?(WIPE (CAR FOO)) BAR ?(WIPELIST) (BAR)	This WIPE's the atom BAR from the OBLIST and places it on the WIPELIST.
?(EQ (CAR FOO) 'BAR) NIL	The atom BAR placed on the WIPELIST is now no longer the same as the atom BAR read in and placed on the OBLIST.

The function WIPELIST, a SUBR of no arguments, returns the copy of the WIPELIST. This copy can be manipulated by list-altering functions such as NCONC, RPLACA, etc., without fear of wrecking the internal WIPELIST and screwing the system.

5.2 Literal Atom Types

There are three types of literal atoms: NIL, Gensym atoms, and non-Gensym, non-NIL literal atoms (nglitat's). They are mostly interchangeable when used in ALISP programs, but the user should be aware of their specific peculiarities.

There is no single predicate to determine if an S-expression is a literal atom or not. LITP will return T for non-NIL literal atoms. The following expression will return T if an S-expression x is any literal atom:

```
(OR (NULL x) (LITP x))
```

5.2.1 NIL

NIL is dear to the heart of every LISP user. It is ubiquitous and fills a multitude of needs in its dual role as empty list and literal atom.

The test for NIL is the function NULL, a SUBR of one argument, which returns T if its argument is NIL, and NIL if it is non-NIL. The NULL function is equivalent to the logical NOT function found on some LISP systems, since in ALISP logical truth is signalled by any non-NIL value, logical falsity by NIL.

NIL is represented in core by an ALISP pointer of all zeros. Since this points to the first word in free space, which is a word of all zeros, the CAR and CDR of NIL are also both NIL. NIL is the only atom which CAR and CDR will accept as an argument.

The pname of NIL is, of course "NIL", although NIL can also be input as "()". NIL is not on the OBLIST, and cannot be WIPE'd or SPECIAL'ed or garbage-collected.

The value of NIL is always NIL, and cannot be changed with any of the functions SETQ, SET, or QSETQ. Nor can its value be REMOB'ed (see 5.3.2 below). The plist of NIL does not exist; any of the plist functions will complain on being given NIL as an atom which is supposed to have a plist.

5.2.2 Gensym Atoms

Gensym atoms are very much like other non-NIL literal atoms, except that they are not on the OBLIST and have funny (but unique) pnames. Because they are not on the OBLIST, Gensym's can never be recognized by READ.

Gensym atoms are useful in LISP programs which must generate symbols, usually as tags to list structures. Tree-building programs will often use Gensym atoms as node names. The MILISY (MIni-LInguistics SYstem) program uses Gensym atoms as internal names for objects in its data-base. Gensym atoms can be created on the fly by ALISP programs, and each new Gensym is guaranteed unique.

Gensym atoms are created using the function Gensym (GENerate SYMbol), a SUBR of no arguments. It returns as its value, each time it is called, a newly-minted Gensym with pname Xnnn... The n are digits for an integer (in OUTBASE representation) unique to that Gensym; the X is the Gensym prefix character. Each time Gensym is called, the Gensym digit is advanced by one, so that subsequent Gensym calls will return pnames with an incremented integer.

The atom GENCHAR controls the Gensym pname character prefix. If GENCHAR has value NIL, the default character "G" is used.

GENCHAR can also be set to any non-NIL, non-Gensym literal atom. The first character of the pname of this atom is used as the Gensym pname prefix. Gensym will complain when called if GENCHAR is not NIL or an nglitat. Initial value for GENCHAR is NIL. Examples of the Gensym function are given in Dialogue 5.3 below.

Dialogue 5.3
The Function Gensym

?GENCHAR NIL ?(Gensym) G0	Initial value for GENCHAR is NIL, so "G" is used as the Gensym character.
?(SETQ GENCHAR 'ANOTHER) ANOTHER	This sets the Gensym character to "A".
?(Gensym) A1	Note that the Gensym counter has been incremented for each new Gensym call.
?(SETQ FOO (Gensym)) A2 ?(EQ 'A2 FOO) NIL	The atom with pname "A2" is different from the Gensym atom "A2".

The pname manipulating functions PACK and UNPACK do not work on Gensyms, and will complain if given such.

To find out whether an S-expression is a Gensym atom, the function Gensymp, a SUBR of one argument, can be used. Gensymp returns T if its argument is a Gensym atom, NIL if it is not.

5.2.3 Nglitats

These are non-NIL, non-Gensym literal atoms. All of the properties described in section 5.3 apply to nglitats. The property which distinguishes nglitats from the other two types of literal atoms above is a print-name consisting of an arbitrary string of characters (but less than 322) used by the ALISP system when inputting and outputting the atom; and internalization of the atom on either the OBLIST or the WIPELIST (see section 5.1 above). The pname functions PACK and UNPACK will only work on nglitats and NIL.

5.3 Literal Atom Properties

These properties apply to nlitats, i.e., non-NIL literal atoms, with the exceptions noted.

5.3.1 Pname

The pname or print-name is a character string associated with an nlitat, and used for communication between the ALISP system and the user. Whenever the atom must be printed, the pname character string is outputted; if the atom can be input, it is by means of the pname character string.

Gensym atoms will print as a single character followed by an integer unique to that atom (see section 5.2.2). The slash convention for printing exotic atom characters is not used, even if the switch SLASHES is set. Gensym atoms cannot be input.

Nglitats can have pnames of up to 322 characters. An nglitat is output by printing its character string, using the exotic atom slash convention if the switch SLASHES is set. Nglitats can be input by typing in their character strings, using the slash convention on input to READ for exotic atoms (see section 3.2.2).

The functions PACK and UNPACK enable the user to explicitly manipulate literal atom pnames. These functions will not work with Gensym atoms; however, they do work with NIL.

UNPACK, a SUBR of one argument, returns a list of atoms whose pnames consist of the individual characters in the pname of its argument. The atom NIL UNPACK's as the list (N I L).

PACK, a SUBR of one argument, packs the first characters of the pname of each element in its argument into a new pname, which it then internalizes and returns as a literal atom. Its argument must therefore be a non-empty list of nglitats.

Examples of the PACK and UNPACK functions are given in Dialogue 5.4 below.

It is sometimes valuable to know just how many characters are in a pname. The function ATLENGTH, a SUBR of one argument, does just that. If its argument is a literal atom, it counts the number of characters in its pname and returns that count as an SNUM. ATLENGTH will actually take any atomic S-expression as its argument. It returns the following values:

NIL	:	3
Gensym	:	6
Number	:	print length of number, including decimal point and minus sign.
nglitat	:	length of pname

Dialogue 5.4
The Functions PACK and UNPACK

```
 ?(UNPACK NIL)
(N I L)
 ?(UNPACK 'FOO)
(F O O)
 ?(UNPACK 'HI/ØTHERE)
(H I Ø T H E R E)
```

The atom "HIØTHERE" has a blank character in its pname. UNPACK handles this character just as it would any other, by forming an atom with the blank character as its pname.

```
 ?(PACK '(F O O))
FOO
 ?(EQ (PACK '(F O O))'FOO)
T
```

Note that the atom formed by PACK is the same as that inputted by the READ function.

```
 ?(PACK '(FOO BAR))
FB
```

Note that only the first character in the atoms FOO and BAR is used by PACK.

```
 ?(PACK (UNPACK 'FOO))
FOO
```

PACK and UNPACK are inverse functions.

5.3.2 Value

All nlitats have a value cell which holds the current value of the atom. The value can be any valid ALISP S-expression.

When an nlitat is initially created or read in for the first time, it is given the atom ILLEGAL as a value. This special atom is checked for by the interpreter, which considers an nlitat to have no value if it finds ILLEGAL in the value cell.

The value of an nlitat can be changed at any time by one of the functions SETQ, SET, or QSETQ. SET is a SUBR of two arguments; it sets the value cell of the first argument to the second argument. SETQ and QSETQ are both FSUBR*'s of an indefinite number of arguments. The first of each pair of arguments is an nlitat whose value cell is to be set, the second is the value to set it to. SETQ differs from QSETQ in that it evaluates the second of each pair of arguments. All of these functions return the value of the last set made as their result.

Examples of the SET functions are given below in Dialogue 5.5.

Dialogue 5.5
The Functions SET, SETQ, and QSETQ

```
 ?(SET 'FOO 'BAR)
BAR
 ?FOO
BAR
```

This sets the value of FOO to the atom BAR. Note that SET evaluates its first argument.

```
 ?(SETQ FOO 'MOO BAR 'MAR)
MAR
 ?FOO
MOO
 ?BAR
MAR
```

SETQ takes any number of pairs of arguments, and only evaluates the second of each pair. It returns the value of the last set.

```
 ?(QSETQ FOO MOM BAR DAD MOO COW)
COW
 ?(LIST FOO BAR MOO)
(MOM DAD COW)
```

QSETQ has the same format as SETQ, but evaluates none of its arguments.

It is impossible to set the value cell of an atom to the atom ILLEGAL using the above functions, since ILLEGAL can neither be read in nor passed from one expression to another with any ALISP function (EVAL always intercepts it and complains with a VAL-ERR). The function REMOB, a SUBR of one argument, is provided to stuff ILLEGAL into the value cell of its argument. The reason you might want this to be done is to remove the atom from the ALISP system. If an nlitat has value ILLEGAL, a plist of NIL, is not SPECIAL'ed, and is not pointed to by any reachable ALISP data structure, then it is a TWA and will be collected on the next garbage collect (see section 5.1.1). The function REMOB thus does not immediately remove its nlitat argument from the ALISP system, but sets one of the conditions that will allow it to be removed on a garbage-collect. NIL cannot be REMOB'ed. REMOB returns NIL as its result.

The value cell of a literal atom is automatically accessed by EVAL whenever the interpreter evaluates that atom during program execution. The function EVAL, a SUBR of one argument, will thus return the value of a litat if it is given one as an argument (see section 6.2). The value of NIL is NIL. EVAL will issue a VAL-ERR if the litat has an ILLEGAL value.

VALUEP is a SUBR predicate of one argument. If its argument is a litat which has a legal value, it returns T; if not, it returns NIL.

GETVAL is a SUBR of one argument which returns the value of its argument if it is a litat with a legal value; if not, it returns the atom NOVAL.

Examples of these two functions are given in Dialogue 5.6.

5.3.3 Plist

All nlitats have a plist (property-list) cell which holds the plist for that atom. NIL has no plist. The plist is a true list of indicator and value pairs of the form:

(ind1 val1 ind2 val2 ... indn valn)

When an nlitat is created or read in for the first time, it is given NIL, the empty list, for its plist.

The plist of an atom can be fetched and set using the function PLIST, a SUBR of one or two arguments. The first argument should be an nlitat. The second argument is optional; if present, the plist of the nlitat is set to it; if not present, the plist of nlitat is just returned as the value of PLIST. Examples of the PLIST function are given below in Dialogue 5.7.

PLIST is not the usual access function for plists, however, plists are very handy because of the function primitives which are provided to work with them; see section 10.1. The interpreter does not use the plists of any user-created atoms, and so the user has full control over their contents.

Dialogue 5.6
The Functions VALUEP and GETVAL

```
?(SETQ FOO 'BAR)
BAR
?(EVAL 'FOO)
BAR
```

Note that, since EVAL is a SUBR and evaluates its argument, FOO is quoted if the value of FOO is desired.

```
?(GETVAL 'FOO)
BAR
```

GETVAL returns the same result as EVAL if FOO has a value.

```
?(VALUEP 'FOO)
T
```

VALUEP returns T if FOO has a value.

```
?(VALUEP 'BAR)
NIL
```

BAR has no value.

```
?(GETVAL 'BAR)
NOVAL
?(EVAL 'BAR)
*** VAL-ERR FROM EVAL
```

GETVAL returns NOVAL. EVAL, on the other hand, issues a VAL-ERR. Note that there is no way to distinguish between an atom having the value NOVAL and having the value ILLEGAL, if you are just using GETVAL. The function VALUEP can always be used to check this case, however, since it returns NIL only if its argument has the value ILLEGAL, or is not a literal atom.

Dialogue 5.7
The Function PLIST

```
?(PLIST 'FOO)
NIL
```

Initially, the plist of an nlitat is NIL.

```
?(PLIST 'FOO '(FOO BAR MOM DAD))
(FOO BAR MOM DAD)
```

This sets the plist of FOO to the four-element list (FOO BAR MOM DAD).

```
?(PLIST 'FOO)
(FOO BAR MOM DAD)
```

Chapter 6

The Supervisor and EVAL

This chapter describes the action of the top-level supervisor and the modified EVAL function used by ALISP. The sections on EVAL and Lambda-expressions are especially important.

6.1 Top Level

When the ALISP system is called, it enters the top-level loop of the supervisor. Initially, this is a READENT-EVAL-PRINT loop that eats up S-expressions typed at it (at most one S-expression per line), evaluates them with EVAL, and outputs the results to the terminal. The basic structure of this loop can be modified in several ways, as the following sections show.

6.1.1 SYS

The value of the atom SYS controls the type of evaluation done by the supervisor. If SYS is NIL, EVAL is used. If SYS is T, an EVALQUOTE function is used. In EVALQUOTE mode, two reads are performed. The EVALQUOTE supervisor takes this pair of S-expressions, uses the first of the pair as a function, the second as a list of arguments which will be passed to the function, always without evaluation. EVALQUOTE prints the result and asks for more input. In Dialogue 6.1 a simple example of the use of EVAL and EVALQUOTE supervisors is given.

Both EVAL and EVALQUOTE modes can be discarded in favor of a user-defined evaluation function. To use this mode, SYS should be defined as a function of no arguments (either FLAMBDA or LAMBDA). This function is then used in place of the READENT-EVAL part of the READENT-EVAL-PRINT loop. The user defined function must do all of its own reading; it is called until SYS is set to T or NIL. As an example, Dialogue 6.2 defines a top-level supervisor which allows more than one S-expression per line to be evaluated.

It is important that SYS be defined correctly when it is a user-defined supervisor, since errors will just be trapped and start the evaluation of the faulty supervisor all over. An infinite, unbreakable error loop is thus established. There seems to be no neat way out of this problem if user control over

Dialogue 6.1
The Switch SYS

?SYS NIL	Initial value of SYS is NIL, calling the EVAL supervisor.
?(CONS 'FOO 'BAR) (FOO,BAR) ?(QUOTE FOO) FOO	Under EVAL mode, one S-expression is read, and evaluated with EVAL.
?(SETQ SYS T) T	This sets the supervisor to EVALQUOTE mode.
?CONS (FOO BAR) (FOO,BAR)	In EVALQUOTE mode, the first S-expression read is used as the function, the second as a list of arguments. Note that the arguments to functions which normally evaluate their arguments, such as CONS, remain unevaluated at top level.
?SET (SYS NIL) NIL	This sets the supervisor back to EVAL mode.

---

the supervisor is desired.

Note also that user-defined supervisors do not have to call the EVAL function, but can do any type of interpretation that it is possible to do with ALISP functions. It is a good idea to be able to get back to the normal supervisor, however, by setting SYS to NIL or T from the user-defined supervisor.

Once the value of SYS is reset from a user-defined supervisor, the supervisor is lost (unless it is stored somewhere besides the value cell of SYS), since it, like all named function definitions, is contained in the value cell of a literal atom.

The switches SYSIN, SYSOUT, SYSPRIN and * are all active under a user-defined supervisor (see below, 6.1.2 and 6.1.3).

One good feature of the SYS=NIL supervisor, as opposed to the user-defined supervisor given in the example of Dialogue 6.2, is that it will read only one S-expression from an input line.

Dialogue 6.2
User-Defined Supervisor

```
?(DE SYS () (EVAL (READ)) )
SYS
```

Defining SYS as a LAMBDA function of no arguments causes it to be used as the new supervisor. Note that now READ rather than READENT is used, so that multiple S-expressions can be read from the same line at top level.

```
?(CONS 'FOO 'BAR) (LIST 'FOO)
(FOO,BAR)
(FOO)
```

Both the CONS and LIST expressions are evaluated. Their results are printed in succession.

```
?'A 'B 'C 'D (SETQ SYS NIL)
A
B
C
D
NIL
```

Five S-expressions are read and evaluated from this line; the last one sets the supervisor back to EVAL mode.

```
? 'A 'B 'C
A
?
```

Now only one S-expression per line is evaluated.

S-expressions can be closed with excess right parentheses if READENT is used. This enables the user, at the end of a long S-expression that perhaps extends over several lines, to forget about matching parentheses exactly, and just type 10 or so, certain that the S-expression will be closed and no SYN-ERR will be given. READENT starts a new line and flushes all the excess parentheses when it asks for the next S-expression.

6.1.2 SYSIN and SYSOUT

These atoms control where the supervisor reads S-expressions from and where it prints them out. Initially, both SYSIN and SYSOUT are set to zero, so reading and printing take place on the terminal (for special batch considerations, see the section I.16). These switches work in the following way: before an

S-expression is read at the top-level loop of the supervisor INUNIT is set to SYSIN; before the result of evaluation is printed, OUTUNIT is set to SYSOUT. Changing values of INUNIT and OUTUNIT during an evaluation will not therefore affect the top-level supervisor read and print device assignments. Changing either SYSIN or SYSOUT will, however. Suppose, for example, that you have a permanent file containing S-expressions you would like to have evaluated by the supervisor. Simply open the file as a local file with unit n (see section I.15), and set SYSIN to n. The supervisor will then read through the file and evaluate each S-expression, printing them on the SYSOUT device. If the last statement in the file is (SETQ SYSIN 0), reading will continue from the terminal when the file is exhausted. For more information, see the section on file primitives.

SYSIN and SYSOUT work for a user-defined supervisor as well as the SYS=T or NIL supervisors.

6.1.3 SYSPRIN and *

At times it is desirable to turn off printing by the top-level supervisor. The switch SYSPRIN is provided for this purpose. Setting SYSPRIN to NIL shuts off the printing of results by the supervisor; if SYSPRIN is non-NIL results will be printed on the SYSOUT device. SYSPRIN is initially set to T.

At other times it is nice to be able to reference the value printed by the supervisor as the result of an evaluation, during the next evaluation. The atom * is provided to always hold the result of the last supervisor operation in its value cell, and can be used to access this value. Examples of the use of the atom * are given below in Dialogue 6.3.

Dialogue 6.3
The Atom * at Top Level

```
?(APPEND '(A B C) ' (D E F G))
(A B C D E F G)                    Now * is set  to the result of
                                   the APPEND  operation, namely,
                                   the list (A B C D E F G).

?(SETQ FOO (CONS 'BAR *))
(BAR A B C D E F G)
?FOO
(BAR A B C D E F G)                Now *  is set to the  value of
                                   FOO.

?(CAR *)
BAR
```

6.1.4 EXIT

At any point in an evaluation, the top-level supervisor and the ALISP system can be abandoned by evaluating the function EXIT, a SUBR of no arguments. A sign-off message giving execution statistics will be printed (see section 1.1), and control returns to KRONOS.

6.2 EVAL

The workhorse of the interpreter. I have used a modified version of the McCarthy EVAL, which lends itself well to a speedy implementation, less ambigious syntax for function evaluation, and better conventions for compilation of functional arguments. For most common cases of evaluation, however, the McCarthy EVAL works just the same as a standard EVAL. The only great difference appears with functional arguments (see below and section I.7).

A compressed definition of the EVAL and APPLY functions can be found in Appendix E. The following sections are more descriptive of the action of these two functions, and much more readable than the Appendix. All ALISP data types can currently be EVAL'ed; these sections describe the results.

6.2.1 Atomic Evaluation

i. Number Tokens

EVAL simply returns the number, without doing anything. This applies to all number tokens: SNUM, BNUM, LNUM, ANUM, and PNUM types.

ii. Literal Atoms

If the atom is NIL, NIL is returned. If not, EVAL gets the value cell of the atom and returns that. Note that an atom may have no value, in which case EVAL complains with a VAL-ERR. The atom ILLEGAL is used to indicate that a litat has no value, that is, the value cell of the litat contains the atom ILLEGAL (see section I.5.3.2). Examples of atomic evaluations are given in Dialogue 6.4 below.

The values of litats are always contained in the value cell; there is no association-list which EVAL searches to find litat bindings. As a consequence, a litat can have only one binding at a time. This binding is always in the value cell, and can always be changed using SET, SETQ, or QSETQ. If a litat is used as a variable in a PROG or LAMBDA expression, then its original value is preserved on a stack (the SPDL, or

Dialogue 6.4
Atomic Evaluation

```
 ? 123
123
 ?-5.6E4
-.56E5
 ? #77
#77
```

Number tokens evaluate to themselves.

```
 ?NIL
NIL
```

NIL evaluates to NIL.

```
 ?(SETQ FOO 'BAR)
BAR
 ?FOO
BAR
```

Literal atoms evaluate to their values.

```
 ?BAR

*** VAL-ERR FROM BAR
 ?
```

If a literal atom has no value (i.e., is set to the atom ILLEGAL), then EVAL complains with a VAL-ERR.

Special Push-Down List) until the function has finished execution, at which point the original value of the litat is popped from the SPDL and placed back in the value cell. Such a binding scheme is called shallow-binding. It sacrifices the ability to save binding environments for better exectuion speed.

6.2.2 List Evaluation

When EVAL is given a non-atomic S-expression, it evaluates it as a function form (the S-expression should be a true list; if it is not, the last non-NIL CDR is treated as if it were NIL). A typical function form is:

(fn arg1 arg2 ... argn)

As the mnemonic suggests, the first element of the list is treated as a function, the rest of the elements of the list as arguments to the function.

The first thing EVAL does is try to decide what type of function fn is. It eventually wants to find either a lambda-expression or a PNUM, which are the only valid function types (see section 6.3 below).

If fn is a list, then it must be a lambda-expression; if it is a list and not a lambda-expression, EVAL complains with a FUN-ERR. A lambda-expression is a list beginning with the atom LAMBDA, FLAMBDA, or LABEL; see 6.3 below. Examples of fn as a list are given in Dialogue 6.5.

Dialogue 6.5
Lambda-Expression Evaluation

?((LAMBDA (X) X) 'FOO) FOO	fn is the lambda-expression: (LAMBDA (X) X).
?((FLAMBDA (X) X) 'FOO) (QUOTE FOO)	fn is the lambda-expression: (FLAMBDA (X) X). Note that FLAMBDA's do not evaluate their arguments, so that 'FOO is returned as (QUOTE FOO).
?((LIST 'LAMBDA '(X) 'X) 'FOO)	fn is not a lambda-expression, even though it would evaluate to one.

```
*** FUN-ERR
OFFENDING VAL = (LIST (QUOTE LAMBDA)
  (QUOTE (X)) (QUOTE X))
```

If fn is an atom, it must be a non-NIL literal atom. A number or NIL for fn causes EVAL to issue a FUN-ERR.*

If fn is a non-NIL literal atom, then EVAL looks at its value cell. The value cell must contain a valid function type, either a lambda-expression or a PNUM, or EVAL will complain. All system functions, such as SETQ and CONS, are defined in this way: they have a PNUM in their value cells, indicating a machine subroutine. Examples of atomic fn are given below in Dialogue 6.6.

The search order for fn is summarized in Table 6.1. This search order works extremely well with functional arguments (see section I.7).

---

* The exception to this is if fn is a PNUM; however, since PNUM's cannot be input by READ, it is unlikely that one will end up as the first element of a list. If it does, then it is treated as a machine subroutine function.

Dialogue 6.6
Atomic Function Evaluation

?(NIL 'FOO)	NIL is not a legal value for fn.
*** FUN-ERR FROM EVAL OFFENDING VAL = NIL	
?(123 'FOO)	Any number type except PNUM is also an illegal value for fn.
*** FUN-ERR OFFENDING VAL = 123	
?CAR P#20000001006023 ?(SETQ A (LIST CAR ' '(FOO BAR))) (P#20000001006023 (QUOTE (FOO BAR)))	CAR has a PNUM value. A now has a PNUM in the fn position.
?(EVAL A) FOO	The value of A was evaluated correctly by EVAL because fn has a PNUM value.
?(CONS 'BAR 'BOO) (BAR,BOO)	CONS has a PNUM definition.
?(SETQ FOO CONS) P#20000002006166 ?(FOO 'BAR 'BOO)	Since FOO was set to the PNUM value of CONS, it too had a valid function definition as its value.
?(SETQ FOO (LIST 'LAMBDA (X Y) (CONS X Y))) (LAMBDA (X Y) (CONS X Y)) ?(FOO 'BAR 'BOO) (BAR,BOO)	FOO now has a valid function type, namely a lambda-expression, as its value.
?(BAR 'FOO)	BAR has no value.
*** FUN-ERR FROM BAR OFFENDING VAL = ILLEGAL	

Once a valid function has been found, EVAL makes a decision as to whether or not the arguments are to be evaluated. If the function is a LAMBDA, SUBR, or SUBR*, then the arguments are evaluated before they are passed to the function; if the function is a FLAMBDA, FSUBR, FSUBR*, or LSUBR, then the arguments are not evaluated. Arguments are evaluated from left to right. Several examples of lambda-expression argument evaluation are given below in Dialogue 6.7. For machine subroutines, one can look up their

---

Dialogue 6.7
Lambda-Expression Argument Evaluation

```
?(LAMBDA (X) X) 'FOO)
FOO
```

The single argument to the lambda-expression (QUOTE FOO), was evaluated before the lambda-expression worked on it.

```
?((FLAMBDA (X) X) 'FOO)
(QUOTE FOO)
```

FLAMBDA functions do not evaluate their arguments.

```
?((LAMBDA (X Y) (LIST X Y)) (PRINT 'FOO) (PRINT 'BAR)))
FOO
BAR
(FOO BAR)
?
```

This lambda-expression has two arguments which are evaluated. They are evaluated from left to right, so that FOO prints first, then BAR.

---

type in Appendix C to see if they evaluate their arguments or not (see section 6.3 also).

After deciding whether or not to evaluate the arguments, EVAL passes them to the function and evaluates the function according to its type. For machine functions such as CAR and CONS, this simply involves branching to the routine address in core. Lambda-expressions must bind their variables and have their forms evaluated; see section 6.3 below for more information.

6.2.3 The Function EVAL

EVAL is available as a SUBR of one argument, as well as through the top-level supervisor. The function EVAL evaluates its argument according to the rules given above and returns the result. Note that, because EVAL is a SUBR, its argument is first evaluated before it is passed to the function EVAL; and the function EVAL does aootber evaluation.

Table 6.1
Search Order for fn

```
PNUM or
lambda-exp              --(yes)--done
      /
     (no)
      /
non-NIL
literal atom            --(no)--FUN-ERR
      /
     (yes)
      /
value cell is
PNUM or                 --(no)--FUN-ERR
lambda-exp
      /
     (yes)
      /
   done
```

The function EVLIST, a SUBR of one argument, applies EVAL to each element of its argument, and returns the result of the last evaluation. Examples of EVAL and EVLIST functions are given below in Dialogue 6.8. If EVLIST is given in atomic argument, it does no evaluations, and returns NIL.

6.2.4 APPLY

It is sometimes desirable to apply a function without having its arguments evaluated. The function APPLY is used for this. APPLY is a SUBR of two arguments; the first argument must be a valid function type, the second is an argument list. APPLY applies the function directly to the arguments wihout evaluating them. The first argument must be a valid function type, either a PNUM or a lambda-expression. Usually the unquoted name of a function is used as the first argument to APPLY, as in the examples in Dialogue 6.9 below.

APPLY* is like APPLY except that it is a SUBR* and takes an indefinite number of arguments. The first argument must again be a valid function type; the remaining arguments to APPLY* are used as arguments of this function.

APPLY and APPLY* work with all function types. Examples of these functions are given in Dialogue 6.9 below.

The APPLY functions initially evolved as partners to EVAL in the evaluation process of LISP. Modern systems have streamlined

Dialogue 6.8
The Functions EVAL and EVLIST

```
?(EVAL 1)
1
```

1 evaluates to itself.

```
?(EVAL ' 'FOO)
FOO
```

Here two quoting operations on FOO are needed. EVAL first has its argument evaluated from (QUOTE (QUOTE FOO)) to (QUOTE FOO), then evaluates that to FOO.

```
?(SETQ FOO 'BAR)
BAR
?(EVAL (LIST 'CAR ''(FOO BAR)))
FOO
?(EVAL (LIST 'LIST 'FOO))
(BAR)
?
```

These two examples show the effects of the double evaluation inherent in the EVAL function. In the first, the argument of EVAL is evaluated to (CAR (QUOTE (FOO BAR))) before being passed to EVAL. Then EVAL evaluates that to FOO. In the second, the argument is evaluated to (LIST FOO), which EVAL evaluates to (BAR).

```
?(EVLIST '((PRINT 1) 2))
1
2
```

EVLIST evaluates the first element of its argument, printing a 1, and returns the result of evaluating the last element, 2.

EVAL to work independently; APPLY becomes a subsidiary entry point to EVAL, where no argument evaluation is done. Nevertheless APPLY still has great usefulness in LISP, as an alternative method of passing arguments to a function. Two particularly neat uses of APPLY are described.

The first makes use of the fact that APPLY takes an argument list (unlike APPLY*) to be used with a function. Suppose, for example, that you have a list of SNUM's whose maximum you wish to find. The function MAX (see section I.9.2.1) is the one that you want; but MAX takes an indefinite number of single elements as

Dialogue 6.9
The Functions APPLY and APPLY*

?(APPLY CONS '(FOO BAR)) (FOO,BAR)	APPLY evaluates its arguments, so that it received a PNUM (value of CONS) as its first argument, the list (FOO BAR) as its second. Note that the arguments to the function CONS, FOO and BAR, were not evaluated.
?(APPLY 'CONS '(FOO BAR)) *** FUN-ERR FROM APPLY OFFENDING VAL = CONS	Here the first argument of APPLY evaluates to the atom CONS, which is not a valid function type (although its value is).
?(APPLY (LAMBDA (X) X) '(FOO)) FOO	The lambda-expression evaluapes to itself (see section 6.3.1 below), and is a valid function type.
?(APPLY* CONS 'FOO 'BAR) (FOO,BAR)	With APPLY*, arguments are strung out instead of being in a list.

arguments, rather than a single list of numbers. Thus:

```
(MAX  1 2 3 4)
```

is a valid way to call MAX, but:

```
(MAX '(1 2 3 4))
```

is not. However, using the function APPLY, it is possible to take a list of numbers and apply the function MAX to them. The format is:

```
(APPLY MAX '(1 2 3 4))
```

Since the second argument to APPLY is an argument list, this is the same as if:

```
(MAX 1 2 3 4)
```

had been evaluated.

Secondly, APPLY normalizes the argument evaluation conventions of LAMBDA and FLAMBDA functions by never evaluating the arguments to either. This is most useful when FLAMBDA functions are considered. Suppose, for example, that you have defined a FLAMBDA function FOO of one argument; suppose also that you wish to use the value of the atom BAR as an argument to FOO. Obviously, evaluating:

```
(FOO BAR)
```

will not work, since FOO does not evaluate its argument. However, evaluating:

```
(APPLY* FOO BAR)
```

will do the job, since APPLY* evaluates BAR and applies FOO to it directly.

6.3 Function Types

Function types are completely characterized by three criteria: lambda-expression or machine subroutine, evaluated or unevaluated arguments, and definite or indefinite number of arguments. These criteria are summarized in Table 6.2 below.

Table 6.2
Function Types

Function	Type	Arguments	# of args
LAMBDA	lambda-	evaled	**definite and indef.**
FLAMBDA	expressions	unevaled	
SUBR	machine-	evaled	definite
SUBR*	language	evaled	indefinite
FSUBR	subroutines	unevaled	definite
FSUBR*	(PNUM's)	unevaled	indefinite
LSUBR		unevaled	indefinite

6.3.1 Lambda-expressions

The format for a lambda-expression is:

```
LAMBDA
(or        varlist YA1 exp2 ... expn)
FLAMBDA
```

varlist can be one of three things:

i. NIL. The lambda-expression takes no arguments.

ii. Single literal atom. The lambda-expression takes a variable number of arguments. The literal atom is bound to a list of the arguments (or to NIL if there are no arguments).

iii. List of literal atoms. The lambda-expression takes a fixed number of arguments. Each of the arguments is bound to the corresponding variable in varlist.

These variable binding conventions apply to both LAMBDA's and FLAMBDA's. They are summarized in Table 6.3; examples of

Table 6.3
Lambda Binding Conventions

Varlist	# of args	Variables Bound
NIL	none	no bindings
nlitat X	indefinite (A B C D ...)	X to the list (A B C D ...)
list of nlitats (X Y Z ...)	number of vars (A B C ...)	X to A Y to B Z to C, etc.

bindings are given in Dialogue 6.10 below.

Evaluation of the lambda-expression proceeds as follows. If it is a LAMBDA list, then the arguments are evaluated in order from left to right, and bound to the corresponding variables in varlist according to the conventions just described. A FLAMBDA list is the same except no evaluation of the arguments takes place. Then each of the exp is evaluated in order from left to right, and the value of expn is returned. The exp's are any valid ALISP data types which can be evaluated; there must be at least one of them or an ARG-ERR will be issued.

After all of expi have been evaluated, all lambda variables are restored in their original values. The variable bindings in a lambda-expression only hold for the extent of the lambda-expression execution. Thus, in the example in Dialogue 6.11 below, the variable VAR had value TWADDLE within the lambda-expression, but then had its original value of FOO restored when the lambda-expression was exited. Only one value of VAR is available at a time, however. Everything evaluated

Dialogue 6.10
Variable Bindings

```
?((LAMBDA () T) )
T
```

varlist is NIL, so there are no argumewW to the lambda-expression.

```
?((LAMBDA N N) 'FOO 'BAR 'MOO)
(FOO BAR MOO)
```

Indefinite number of arguments. Note that N is bound to a list of these arguments.

```
?((LAMBDA(X Y) (LIST X Y))
?'FOO 'BAR)
(FOO BAR)
```

X is bound to FOO, Y is bound to BAR.

```
?((LAMBDA (X Y)(LIST X Y)) 'FOO)

*** ARG-ERR
WRONG NO. OF ARGS
OFFENDING VAL = (X Y)
```

If the wrong number of arguments is given to a lambda-expression with a non-atomic varlist, an ARG-ERR results. The varlist in question is printed as part of the error message.

---

within the lambda-expression will see VAR as having value FOO. One says that the effects of binding variables are local to a given lambda-expression.

The function ARGN is very useful when dealing with lambda-expressions of an indefinite number of arguments. Since, in this case, the lambda variable is bound to a list of arguments, it is often necessary to retrieve a particular element from that list. ARGN, a SUBR of two arguments, will do just that; see section 10.2.1 below.

The state at any given moment of all literal atoms and their values is called the environment. Another way to state the fact that only one litat value is available at any given moment is to say that there is only one environment available at any given time. When an exe is evaluated within a lambda-expression, it is evaluated with respect to the environment produced by the binding of the lambda-expression's variables. There is thus no way to always evaluate a function in the environment in which it was defined, the classic FUNARG problem. Most users are not affected

Dialogue 6.11
Lambda-Expression Evaluation

```
?((LAMBDA (X) (PRINT X) (CAR X)) '(FOO BAR))
(FOO BAR)
FOO
```

X was bound to the evaluated argument, (FOO BAR). Then the PRINT function was evaluated, printing the value of X; finally, the CAR function was evaluated, and its result returned as the value of the lambda-expression.

```
?((FLAMBDA (X) (PRINT X) (CAR X)) '(FOO BAR))
(QUOTE (FOO BAR))
QUOTE
```

This is the same as the previous lambda-expression, except that FLAMBDA is used. Thus, the arguments are not evaluated, and X is bound to (QUOTE (FOO BAR)).

```
?(SETQ VAR 'FOO)
FOO
```

This sets the value of VAR to FOO

```
?((LAMBDA (VAR) (PRINT VAR) (SET VAR 'BAR)) 'TWADDLE)
TWADDLE
BAR
```

Within the lambda-expression, VAR is bound to the atom TWADDLE. The set function sets the value of the value of VAR (that is, the value of TWADDLE) to BAR.

```
?VAR
FOO
```

VAR is restored to its old value outside the lambda-expression.

```
?TWADDLE
BAR
```

TWADDLE, set within the lambda-expression to BAR, retains this value even after the lambda-expression is exited, because it was not used as a lambda variable.

---

by this problem. For further information on the FUNARG problem, see Weissman's Primer. An example of this type of problem is given in Dialogue 6.12 below.

The function LABEL is an alternate form of lambda-expression, used when it is desired to name and call the

Dialogue 6.12
The FUNARG Problem

```
 ?(SETQ VAR 'FOO)
FOO
 ?(SETQ VALSET
 ?   (LAMBDA (VALUE) (SET VAR VALUE)))
(LAMBDA (VALUE) (SET VAR VALUE))
```

The atom VALSET is now defined as a lambda-expression, a valid function. VAR is set to FOO.

The environment in which the function VALSET was defined thus had VAR set to FOO; and evaluating VALSET in this environment should set FOO to the value of VALUE when the expression (SET VAR VALUE) is evaluated.

```
 ?((LAMBDA (VAR) (VALSET 20)) 'BAR)
20
 ?FOO

*** VAL-ERR FROM FOO
 ?BAR
20
```

The function VALSET, evaluated within the lambda-expression, saw the value of VAR as BAR and not FOO, and so set BAR rather than FOO to 20.

---

lambda-expression from within its own form. Format is:

(LABEL lname lexp)

where lname is an nlitat, and lexp is a valid lambda-expression. LABEL is used just like an ordinary lambda-expression. When it is evaluated, the litat lname is bound to lexp, so that it becomes a valid function name within lexp. LABEL can be used to define recursive functions: the following example defines the recursive factorial function and supplies it to an integer.

```
?((LABEL FACT (LAMBDA (X) (COND ((ZEROP X) 1)
          (T(TIMES X(FACT (SUB1 X))))))) 6)
24
```

Within the LAMBDA form, FACT is used to refer to the lambda-expression. LABEL is actually of little practical use except as a teaching tool. When it is necessary to give a lambda-expression a name, the function definition procedures given in section 6.4 below are much handier, and also more permanent.

All of the lambda-expression forms described in this section (LAMBDA, FLAMBDA, and LABEL) are defined as LSUBR's. They are identity functions under EVAL, returning themselves. This property is useful mainly in connection with passing lambda-expressions as functional arguments (see section I.7). Examples of the identity functions are given in Dialogue 6.13.

Dialogue 6.13
Lambda-Expressions as Identity Functions

?(LAMBDA (X) (CONS X 'FOO)) (LAMBDA (X) (CONS X 'FOO)) ?(LABEL NN (LAMBDA (X) X)) (LABEL NN (LAMBDA (X) X))	All LAMBDA, FLAMBDA, and LABEL expressions are identity functions.

6.3.2 Machine Language Subroutines

These functions do not bind variables like lambda-expressions. You should be aware, however, that some of them use the values of litats during the course of their execution (the read and print functions use the buffer pointers, GENSYM uses GENCHAR, etc.).

i. SUBR -- evaluates a definite number of arguments.

Typical examples of SUBR's are CONS, CAR, CDR, ATOM, etc.--most of the familiar LISP functions. A SUBR function will complain if it is given too many or two few arguments by issuing an ARG-ERR.

ii. SUBR* -- evaluates an indefinite number of arguments.

Typical examples of SUBR*'s are the numeric functions TIMES and PLUS. These two functions will take as many arguments as you care to give them, but you must give them at least two. Although it is true that SUBR* functions in general have the ability to take any number of arguments, most have restrictions like TIMES and PLUS so that they will accept a variable number of arguments within a certain range. The function PLIST, for example, takes only one or two arguments, and gives an ARG-ERR for any other number. Individual restrictions for SUBR*'s are given in descriptions of the functions in the text.

iii. FSUBR -- unevaluated, definite number of arguments.

Typical examples of FSUBR's are QUOTE and DEFPROP, which take one and three arguments, respectively. An FSUBR function will complain if given the wrong number of arguments by issuing an ARG-ERR.

iv. FSUBR* and LSUBR -- unevaluated, indefinite number of arguments.

The difference between these two is an internal one in the way arguments are passed on the stack, and invisible to the user. Most of the program flow controlling functions (COND, IF, PROG, AND, OR, etc.) are of this type. FSUBR* and LSUBR functions act just like SUBR*'s, except that their arguments are passed without evaluation.

6.4 Defining Functions

Since valid functions are always put into the value cell of an nlitat, there are several ways to define user functions. Any of the functions SET, SETQ, or QSETQ used to put a lambda-expression list into the value cell of a litat will work; the standard DE and DF are also provided. DE and DF are LSUBR's which use the following standard format:

```
  DE
 (or fn varlist exp1 exp2 ... expn)
  DF
```

The result of evaluating the above form is to place in the value cell of fn the following lambda-expression:

```
  LAMBDA
 (or    varlist exp1 exp2 ... expn)
  FLAMBDA
```

DE defines a LAMBDA, DF a FLABMDA function. fn must be an nlitat or DE (or DF) will complain with an ARG-ERR. DE and DF return fn. Examples of DE and DF functions are given below in Dialogue 6.14.

*** NOTE *** NOTE *** NOTE *** NOTE *** NOTE ***

Because functions are contained in the value cell of an nlitat, that nlitat cannot have both a function definition and a value at the same time; its function definition is a lambda-expression which is also its value. Unlike most other

Dialogue 6.14
The Functions DE and DF

?(DE PLUS2 (X) (PLUS X 2)) PLUS2 ?PLUS2 (LAMBDA (X) (PLUS X 2))	PLUS2 is defined as a lambda-expression. Note that a lambda-expression is actually stuffed as the value of PLUS2.
?(SETQ PLUS2 '(LAMBDA (X) (PLUS X 2))) (LAMBDA (X) (PLUS X 2))	This SETQ has the same effect as the DE.

LISP systems, ALISP does not have a separate value and function definition slot for each nlitat. If you want to use an nlitat to name a function, then you cannot use it as a variable at the same time. The only circumstance where this is bothersome is where you'd like to use an atom which is already a system function (such as LIST, ATOM, etc.) as a variable within a lambda-expression. This is ok as long as the system function is not needed during the evaluation of the lambda-expression, because then the atom will have (in general) a non-PNUM value. An example of what not to do is the following:

```
 ?(DE NEXT2 (LIST) (LIST (CAR LIST)(CADR LIST)))
NEXT2
 ?(LIST 'FOO 'BAR)
(FOO BAR)
 ?(NEXT2 '(FOO BAR MOO))

*** FUN-ERR FROM LIST
OFFENDING VAL = (FOO BAR MOO)
 ?(SETQ LIST 1)
1
 ?(LIST 'FOO 'BAR)

*** FUN-ERR FROM LIST
OFFENDING VAL = 1
 ?
```

The first mistake made above was to try to use LIST as both a function and variable within NEXT2. Since the atom LIST was set to (FOO BAR MOO) when NEXT2 was called, it lost its function definition (PNUM) within the scope of NEXT2, and EVAL could not find a valid function definition for LIST. After the FUN-ERR, LIST gets reset to its value before NEXT2 was called, and is once again a valid function (PNUM). Setting LIST to 1 at top level, however, erases its PNUM value and thus wipes out irretrievably the function definition associated with LIST.

Because of the dual nature of value cells and the ease with which function definitions can be erased, it is recommended that you not try to use an nlitat as both a variable and a function, even if you can set up these uses so they do not conflict. In any case, never use system function litats as variables, and never change them with SETQ (SET, QSETQ) or REMOB.

6.4.1 Checking for Function Definition

An nlitat can be checked for a valid function definition by the function GETFUN, a SUBR of one argument, or FNTYPE, also a SUBR of one argument. FNTYPE will return the function type of its argument, as the atom LAMBDA, FLAMBDA, SUBR, etc. GETFUN returns the function itself, either a lambda-expression or a PNUM. Both return NIL if their arguments do not have valid function definitions. Note that GETFUN and FNTYPE can take either a valid function type or an nlitat with a valid function type in its value cell, as an argument. Examples of these two functions can be found below in Dialogue 6.15.

Dialogue 6.15
The Functions FNTYPE and GETFUN

```
 ?(FNTYPE 'SETQ)
LSUBR
 ?(FNTYPE 'SET)
SUBR
```

SETQ is an LSUBR functions. Note that FNTYPE evaluates its argument.

```
 ?(DE PLUS2 (X) (PLUS X 2))
PLUS2
 ?(FNTYPE 'PLUS2)
LAMBDA
 ?(FNTYPE PLUS2)
LAMBDA
```

PLUS2 is defined as a lambda-expression. FNTYPE returns LAMBDA as the type of function of PLUS2. Note that, even when PLUS2 is not quoted as an argument to FNTYPE, FNTYPE still gets the correct function definition. This is because PLUS2 evaluates to a lambda-expression, and FNTYPE will recognize lambda--expressions.

```
 ?(GETFUN 'SET)
P#20000002006123
```

The function definition of SET is a PNUM, which GETFUN

6.5

returns.

```
?(GETFUN 'PLUS2)
(LAMBDA (X) (PLUS X 2))
```

GETFUN returns the lambda-expression function definition of PLUS2.

6.4.2 Erasing Function Definitions

Function definitions can be overwritten by using DE or DF on the already-defined function name. The new definition replaces the old.

Function definitions can be erased using REMOB (see section 5.3.2) which sets the value of the nlitat to ILLEGAL. To change the name associated with a function, just do:

```
(SETQ  newname oldname)
(REMOB 'oldname)
```

6.5 Switches

Switches are ALISP nslitats whose value cells are important to certain functions. The value is used by the function as a switch to determine a particular course of evaluation. An example of this switching action is found in the atom HPRNUM, which is used by the function HALFPRI. The value of HPRNUM is an SNUM positive integer signalling the function HALFPRI as to how many atoms it should output before it stops printing (see section 4.2.2). The user can change the value of HPRNUM at any time by using one of the SET functions.

Switches are a way of communicating with an ALISP function without passing its arguments. HALFPRI could just as easily have been defined as a SUBR of two arguments, the second of which specified a limit on the number of atoms printed. The advantage of using switches lies in their external nature. Once HPRNUM is set, all HALFPRI calls, no matter what their origin, will print using the HPRNUM limit. A function which uses HALFPRI can then be defined without reference to the current value of HPRNUM, and yield different results when evaluated with HPRNUM set to different values. The control resides not with the defined function, but with the environment it is evaluated within.

The chief advantage of environmental as opposed to definitional control for certain proceeses is the ease with which the environment can be changed. Suppose, for example, that you have defined a function called MYPRINT which uses HALFPRI at several points in its execution. In order to change the number of atoms printed by HALFPRI at each of these points, it is only necessary to change HPRNUM once. If you do not wish to destroy the original value of HPRNUM, the function MYPRINT can be embedded in a lambda-expression which has HPRNUM as one of its variables. When the lambda-expression is entered, HPRNUM is set to the desired value; when it is exited, the old value of HPRNUM is restored, and the environment has been preserved in a very handy way.

Switches are indexed along with pre-defined functions in Appendix C.

I Chapter 7

Functionals

Functionals are functions which take other functions as arguments. A function used as an argument will be called a functional argument (meaning either that it is an argument to a functional or that it is a function which is also an argument, take your pick). Because of the nature of the modified ALISP EVAL, functional arguments are not passed in quite the same way as with most other EVAL's. The following sections explain differences and describe the pre-defined functionals available in the ALISP system.

7.1 Passing Functional Arguments

The easiest way to understand the workings of functional arguments is to go thru an example. Start with the form:

```
(FN X Y)
```

In order for this form to evaluate correctly, the atom FN must have a valid function definition, either a PNUM or lambda-expression, in its value cell (see section I.6.2). Keeping this in mind, we embed the form in a function, thus:

```
(LAMBDA (FN) (FN X Y))
```

Now FN is also a variable in a LAMBDA form. Since it is a variable, it gets bound to an argument when the LAMBDA form is used as a function; and since binding stuffs the argument into the value cell of FN, the argument must be a valid function type, a PNUM or lambda-expression. This seems easy enough, so a few examples of functional arguments are given in Dialogue 7.1, and a commentary follows here.

In Dialogue 7.1, FOO is initially defined as a lambda-expression. The first time FOO is called, its argument is CONS. Since CONS evaluates to a PNUM (and FOO evaluates its arguments) the variable FN gets bound to a PNUM. When the form (FN X Y) gets evaluated within FOO, FN has a PNUM, a valid function type, in its value cell, and so the form evaluates correctly.

In the second call to FOO, the argument is (QUOTE CONS).

Dialogue 7.1
Functional Arguments

<pre>?(SETQ X 1 Y 2)
2
?(DE FOO (FN) (FN X Y))
FOO
?FOO
(LAMBDA (FN) (FN X Y))</pre>
	Initializations. X is set to 1, Y to 2, and FOO is defined as a function of one argument.

<pre>?(FOO CONS)
(1,2)</pre>
	Correct passing of functional arguments, CONS.

<pre>?(FOO 'CONS)
 	
*** FUN-ERR FROM FN
OFFENDING VAL = CONS</pre>
	Incorrect passing of functional argument.

<pre>?(FOO '(LAMBDA (X Y) (LIST X Y)))
(1 2)
?(FOO (LAMBDA (X Y) (LIST X Y)))
(1 2)</pre>
	Correct passing of lambda-expressions as functional arguments. Either quoted or unquoted forms will do.

<pre>?(DE LIST2 (X Y) (LIST X Y)))
LIST2
?(FOO LIST2)
(1 2)</pre>
	Lambda-expression is passed as the value of LIST2.

<pre>?(FOO 'LIST2)
 	
*** FUN-ERR FROM FN
OFFENDING VAL = LIST2
?</pre>
	Incorrect since LIST2 is not itself a valid function type.

This evaluates to the atom CONS, and FN is bound to it. Now when the form (FN X Y) gets evaluated, it does not have a valid function type in its value cell, but rather the atom CONS. A FUN-ERR results.

The rest of the calls to FOO show examples of lambda-expressions used as functional arguments. In the first one, '(LAMBDA...) evaluates to a lambda-expression, which then gets bound to FN, and (FN X Y) evaluates correctly. In the second, the LAMBDA form itself gets evaluated. This is no problem, however, since LAMBDA and FLAMBDA forms just evaluate to themselves (see section I.6.3.1). The result is the same as the previous expression.

Finally, consider an atom, LIST2, which has a function definition (lambda-expression) in its value cell. This case is entirely analogous to the first case considered, i.e., (FOO CONS), where CONS also have a valid function definition (a PNUM) in its value cell. The value of LIST2 is bound to FN, and FN then has a lambda-expression value, so the form (FN X Y) evaluates properly. Once again, however, the quoted atom LIST2 will not work, since FN gets bound to the atom LIST2 rather than its lambda-expression value.

There is a golden rule for passing functional arguments in ALISP:

Never quote a function name used as a functional argument.

The above rule will never lead you astray when functional arguments are called for.

The LISP hacker may have noticed a problem with the passing of functional arguments in the above example. The function FOO was defined as a LAMBDA (using DE), so that it evaluated its arguments. It was this evaluation which enabled it to get the values of functional arguments such as CONS and LIST2, and correctly apply them. If FOO were defined as a FLAMBDA (using DF), then no such evaluation of arguments would take place, and all of the examples in Dialogue 7.1 would fail, except for the one where an unquoted LAMBDA-list was used as an argument. Well, it is obvious that FLAMBDA functions which use functional arguments need some method for getting the function definition of an atom passed as a functional argument. The simplest solution is to use the function GETFUN (section I.6.4.1). In Dialogue 7.2, for example, FOO is defined as an FLAMBDA equivalent of its definition in Dialogue 7.1. On the first call to the function FOO, FN is bound to its unevaluated argument, CONS. The SETQ call sets FN to the function definition of CONS, a PNUM. Then the form (FN X Y) evaluates correctly. Thus, passing functional arguments to FLAMBDA functions is no problem, as long as the FLAMBDA variable is reset to the function definition of the argument, with GETFUN.

7.2 Pre-defined Functionals

These functions are identified by the letters "MAP" appearing in their pnames. They take a function and apply it to successive CDR's or elements of a list. All are SUBR's of two arguments, the first argument being a list, the second a function

Dialogue 7.2
Functional Arguments to FLAMBDA Functions

```
 ?(SETQ X 1 Y 2)
2
 ?(DE FOO (FN) (FN X Y)))
FOO
 ?FOO
(FLAMBDA (FN) (SETQ FN (GETFUN FN)) (FN X Y))
```

Initialization. FOO is set to the FLAMBDA equivalent of its definition in Dialogue 7.1.

```
 ?(FOO CONS)
(1,2)
```

This succeeds because GETFUN resets FN from CONS to the PNUM value of CONS.

---

to apply to parts of the list. Note that this order of arguments for the MAP functions is reversed from that of some LISP implementations. The result returned by a particular MAP function depends upon the nature of the function.

7.2.1 MAPC and MAPCAR

These two functions apply their second argument to successive elements of the first. MAPC returns the result of the last application, while MAPCAR returns a list of the results of all applications. The equivalent LISP definitions of MAPC and MAPCAR are given at the end of this chapter in Table 7.1. Examples of the MAPC and MAPCAR functions are given below in Dialogue 7.3.

7.2.2 MAPL and MAPLIST

These functions are just like MAPC and MAPCAR, except they apply their second arguments to successive CDR's of the first argument. MAPL returns the result of the last application, while MAPLIST returns a list of the results of all applications. The MAPL and MAPLIST functions are defined as LISP lambda-expressions in Table 7.1. Examples of these two functions are given below in Dialogue 7.4

7.2.3 MAPCON and MAPCONC

Note that MAPCAR and MAPLIST always return a list with the same number of elements as their first argument. It is often desirable to delete certain of the elements returned in the final list. MAPCON and MAPCONC, by using NONC rather than CONS to

Dialogue 7.3
The Functions MAPC and MAPCAR

```
?(MAPCAR '(FOO BAR) PRINT)
FOO
BAR
(FOO BAR)
```

MAPCAR applies the function PRINT to successive elements of the list (FOO BAR). Note that the unquoted atom PRINT is used as an argument. The alue of MAPCAR is a list of the values of the PRINT function.

```
?(MAPC '(FOO BAR) PRINT)
FOO
BAR
BAR
```

MAPC is like unto MAPCAR, but returns as its result only the last result of the application of PRINT. MAPC is used when the effect of a function, rather than its result, is desired.

```
?(MAPCAR '(FOO BAR MOO)
?  (LAMBDA (X) (EQ X 'BAR))))))
(NIL T NIL)
```

An example of a lambda-expression as an argument to MAPCAR. The lambda-expression was used without quoting, since it evaluates to itself.

---

string together the results of application of the second argument, allow a variable number of elements to be returned.

The difference between MAPCON and MAPCONC is the same as the difference between MAPLIST and MAPCAR; MAPCON applies the second argument to successive CDR's of the first argument; MAPCONC to successive elements.

In this example, MAPCONC is used to delete all non-atomic elements from a list:

```
?(MAPCONC '(FOO (NIL T) BAR (MOO))
          (LAMBDA (X) (IF (ATOM X)(LIST X))))
(FOO BAR)
?
```

This can be understood as follows: apply the lambda-expression to each element of the first argument. The four results are:

```
FOO     gives (FOO)
(NIL T) gives NIL
BAR     gives (BAR)
(MOO)   gives NIL
```

Dialogue 7.4
The Functions MAPL and MAPLIST

?(MAPLIST '(FOO BAR) PRINT) (FOO BAR) (BAR) ((FOO BAR) (BAR))	MAPLIST applies PRINT first to the list (FOO BAR), then to its CDR, (BAR). The result of the MAPLIST function is a list of the results of the PRINT function.
?(MAPLIST '(FOO BAR MOO) CAR) (FOO BAR MOO)	Here, MAPLIST reconstructs its first argument by applying CAR to successive CDR's of (FOO BAR MOO).
?(MAPL '(FOO BAR) PRINT) (FOO BAR) (BAR) (BAR)	MAPL only returns the value of the last application of PRINT.

These four results are now strung together using NONC. It is easy to see that the result of these NONC's is the list (FOO BAR), which is exactly the result returned by the MAPCONC call.

Table 7.1
LISP Definitions of the MAP Functions

```
(DE MAPC (X FN)
     (COND ((ATOM X) NIL)
           ((ATOM (CDR X)) (FN (CAR X)))
           (T (FN (CAR X)) (MAPC (CDR X) FN)) ))

(DE MAPCAR (X FN)
     (COND ((ATOM X) NIL)
           (T (CONS (FN (CAR X)) (MAPCAR (CDR X) FN))) ))

(DE MAPL (X FN)
     (COND ((ATOM X) NIL)
           ((ATOM (CDR X)) (FN X))
           (T (FN X) (MAPL (CDR X) FN)) ))

(DE MAPLIST (X FN)
     (COND ((ATOM X) NIL)
           (T (NCONC (FN X) (MAPLIST (CDR X) FN))) ))

(DE MAPCON (X FN)
     (COND ((ATOM X) NIL)
           (T (CONS (FN X) (MAPLIST (CDR X) FN))) ))

(DE MAPCONC (X FN)
     (COND ((ATOM X) NIL)
           (T (NCONC (FN (CAR X)) (MAPCONC (CDR X) FN))) ))
```

I Chapter 8

Erogram Elow

The functions described in this section are those used to control program flow -- COND, IF, PROG, etc. In most respects they act like the standard functions described in Weissman's Erimer.

8.1 Conditionals

Four functions are described in this section: COND, IF, AND, and OR. They are all LSUBR's; COND and IF must have at least one argument, AND and OR can take none.

In general, when a conditional tests a value, it looks for either a NIL or a non-NIL value, i.e., everything which is not NIL is considered to be true in a conditional test. Logical truth in ALISP is represented by any non-NIL S-expression, logical falsity by NIL.

8.1.1 COND and IF

Format for COND is:

```
(COND  (pred1 exp11 exp12 ... exp1n)
       (pred2 exp21 exp22 ... exp2m)
       .
       .
       .
       (predj expj1 expj2 ... expjq) )
```

The evaluation order for the arguments of COND is as follows: each ered is evaluated in order, starting with ered, until one is found which returns a non-NIL value. store Examples of legal COND forms:

```
(COND  ((ATOM X) (SETQ X NIL)  (CONS Y X))
       ((EQ (CAR X) 'FOO) (CONS (CDR X) Y))
       (T (SETQ Y X) (NCONC Y '(FOO BAR)) (CDR X)))

(COND  (X NIL)
       ((FOO Y)))
```

Note that in the second example, the second COND argument had a ered but no exe. When this occurs, and ered evaluates to a non-NIL expression, that expression is returned as the value of the COOND.

The function IF is a shortened COND with a single predicate. iFormat is:

```
(IF pred exp1 exp2 ... expn)
```

pred is evaluated, and if the result is NIL, the value of the IF function is NIL. If non-NIL, exp thru expn are evaluated in order , and the value of the last is returned. If there are no exp, then the value of the pred expression becomes the value of the IF (but in this case, the IF function is superfluous anyhow).

The SELECTQ function is a specialized COND. Format is:

```
(SELECTQ sexp
          (a1 exp11 exp12 ... exp1n)
        (a2 exp21 exp22 ... exp2n)

            .
            .
            .

        (aJ expJ1 expJ2 ... expJq)
           dexp)
```

Sexp is evaluated (its result should be atomic) and checked against each atom a's for a match. If one is found, the corresponding exps are evaluated, and the value of the last is returned as the value of the SELECTQ. If no a's matches, dexp is evaluated and its result returned.

Note that SELECTQ uses EQ in checking for a match to the a's, so that SNUM's, literal atoms, and gensym atoms are okay (see section I.9.1).

8.1.2 AND and OR

These functions act as continuous conditionals, testing the values of each of their arguments. Format is:

```
AND
( or exp1 exp2 ... expn)

OR
```

Each of exp is evaluated sequentially from left to right.

AND stops at the first NIL value and returns NIL; if no NIL result is encountered, the value of expn, the last expression to be evaluated, is returned as the value of the AND function.

OR stops at the first non-NIL value and returns that; if no non-NIL result is encountered, OR returns NIL.

If there are no exp, AND returns T, OR returns NIL as a result. Examples of AND and OR functions are given below in

8.2 Program Feature

The functions PROG, GO, and RETURN form the program feature. The experienced LISP'er will resort to PROG syntax only when absolutly necessary, since it introduces the FORTRAN-like elements of looping and iteration so foreign to LISP.

8.2.1 PROG

The PROG function acts something like a lambda-expression in that it binds variables and evaluates a sequence of expressions, but it also has the ability to jump program control between expressions within its body. Format is:

(PROG varlist exp1 exp2 ... expn)

where varlist is a list (perhaps empty) of variables to be used within the PROG, and exp thru expn are arbitrary S-expressions composing the body of the PROG. If any of exp are atomic, they are treated not as expressions to be evaluated but as labels for control of program flow. SNUM's and litats (including Gensym's and NIL) are all valid labels which will work with GO.

When the PROG is entered, all variables on varlist are bound to NIL. Each exp is then evaluated sequentially from left to right, with labels (atomic exp) being skipped. Unless a GO or a RETURN statement is encountered somewhere within an exp, PROG exits with value NIL after expn has been evaluated. Program flow is diverted from this order with the GO and RETURN functions.

The function GO is an FSUBR of one argument. If its argument is non-atomic, it keeps evaluating it until it is atomic. It then uses this atom to match a label in a PROG body. If no match is found, the PROG is exited with value NIL. If a match is found, execution of exp within the PROG body starts again from the matched label. Looping and branching in general within a PROG are accomplished with the GO statement.

RETURN is a SUBR of one argument. It causes an immediate exit from the PROG, and the PROG returns as its value the argument of RETURN. Note that the only way to exit a PROG with a value other than NIL is with the RETURN function.

Both RETURN and GO can be used at any level within a PROG. They need not even be explicitly present in the body of the PROG, for example, an exp could call a function which has a GO or RETURN call, and they will work correctly. If, however, a RETURN or GO is executed outside the scope of a PROG, an error will be

Dialogue 8.1
The Functions AND and OR

```
?(AND (SETQ FOO 'BAR)
?      (CAR '(NIL))
?      (SETQ FOO 'MOO))
NIL
?FOO
BAR
```

The AND functions first evaluated the SETQ, setting the value of FOO to BAR. Next, the CAR function was evaluated yielding NIL; the AND function stopped at this point and returned NIL. The last SETQ never got evaluated, so the value of FOO is BAR.

```
?(AND (PRINT 'FOO) (PRINT 'BAR)
FOO
BAR
BAR
```

Here AND evaluates the first PRINT function, which prints FOO and returns the non-NIL value FOO. Then the second PRINT function is evaluated, printing BAR and returning a non-NIL result, BAR. The result of the whole AND expression is the result of the last PRINT evaluation, namely, BAR.

```
?(OR (SETQ FOO 'MAR)
?      (PRINT 'BAR)
?    (SETQ FOO 'BAR))
MAR
?FOO
MAR
```

The OR function stops at the first non-NIL result it encounters, and returns that. In this case, the first SETQ expression returned the atom MAR, Px so OR stopped there and returned the atom MAR as its value. Now FOO is set to MAR.

```
(OR (PROGN (PRINT 'FOO) NIL)
?      (PROGN (PRINT 'BAR) NIL))
FOO
BAR
NIL
```

Here OR evaluates the first PROGN expression. PRINT prints the atom FOO, but the result of the whole PROGN expression is its last evaluation, NIL. Thus OR goes on to the second PROGN expression, which likewise print the atom BAR, but returns NIL as its value. The result of the entire OR expression is NIL.

issued with the message, "NO PROG EXECUTING".

The action of RETURN and GO is immediate. If, for example, a PROG has the following form for one of exp:

```
(SETQ A (RETURN NIL) B 'FOO)
```

then not only would B never be set to FOO, but A would never be set to the value of the RETURN statement, since upon execution, RETURN immediately returns control to the PROG function and causes it to exit. This is true no matter what the calling level at which the RETURN or GO occurs within a PROG.

When the PROG exits, all PROG variables are reset to their values just prior to the PROG call; see section I.6.3.1 for more information about variable findings. An example of the PROG function is given in Dialogue 8.2 below.

8.2.2 PROGN

This function is a castrated PROG; no variables and no labels. Its sole purpose is to allow execution of a number of expressions. It is an LSUBR, with calling format:

```
(PROGN exp1 exp2 ... expn)
```

Each exp is evaluated in sequence from left to right, and the value of the last is returned as the value of PROGN.

8.3 Iteration

It is unfortunately often convenient in LISP to iterate a program sequence a number of times. The function DO, an LSUBR, supplies a simple iteration facility. Format is:

```
(DO n exp1 exp2 ... expn)
```

The first argument n is evaluated; it must evaluate to a positive SNUM. This is the number of times the iteration will proceed. If n is zero or negative, no iterations of the loop will be performed, but DO will simply exit. On each iteration, exp thru expn are evaluated sequentially from left to right. The value of DO is NIL. Examples of the DO function will be found in

Dialogue 8.2
The Function PROG

```
?(SETQ FOO '(A B C D E F G))
(A B C D E F G)

?(PROG (X PRED RESULT)
?    (SETQ X FOO)
?    TAG                          ;THIS IS A LABEL FOR THE PROG LOOP
?    (COND ((NULL X) (RETURN RESULT))   ;EXIT WITH RESULT
?                                       ;IF X IS EMPTY
?          (T (SETQ PRED (CAR X))))     ;ELSE GET FIRST ELEMENT
?                                       ;OF X
?    (SETQ RESULT (CONS PRED RESULT)    ;ADD ELEMENT TO RESULT
?          X (CDR X))                   ;POP ELEMENT OFF OF X
?    (IF (NULL (EQ PRED 'D)) (GO TAG))  ;LOOP TO TAG IF ELEMENT
?                                       ; IS NOT D
?    (RETURN RESULT))                   ;ELSE RETURN THE RESULT
(C B A)
```

The PROG function first bound its variables, X, PRED, and RESULT, to NIL. Then X was set to the value of FOO, or the list (A B C D E F G). The PROG loop was then entered. The first element of X was CONS'ed onto RESULT, unless X was empty or the element was the atom "D". In this case, the atom "D" appeared first, and the result of the PROG was the reversed list (C B A).

Dialogue 8.3
The Function DO

```
?(DO 3(PRINT 'FOO))
FOO
FOO
FOO
NIL
```

The PRINT expression is evaluated three times. The value of the DO function is NIL.

```
?(SETQ N 1 COUNT 4)
?(DO COUNT (PRINT N)
?          (SETQ N (ADD1 n)) )
```

```
?
 1
2
3
4
NIL
 ?N
5
```

Here DO evaluates its first argument, COUNT, yielding an iteration count of 4. First the PRINT expression is evaluated, then the SETQ function. The value of the whole DO expression is again NIL. Note that the effects of the SETQ on N are retained outside of the DO expression; DO does not bind any variables.

A non-NIL result from iteration is returned by the function DOCONS. DOCONS is like DO, except that the values of expn are concatened into a result list. For example:

```
(DOCONS 8 NIL)
```

returns a list of 8 NIL's.

More structured iteration is available with the REPEAT function. The form of REPEAT is like that of PROG, with the addition of an automatic loop and exit flags. Format is:

```
(REPEAT     varlist
            lexp1
            lexp2
             .
             .
             .
            lexpn
        BEGIN
            exp1
             .
             .
             .
        WHILE
            expw
        UNTIL
            expu
             .
             .
             .
            expn)
```

varlist is a list of variables bound initially to NIL, as in PROG. All S-expressions before the BEGIN atom are evaluated once, in order to set up initial values of variables or perform other actions before the main repeat loop (if there is no setup to be done, BEGIN may be omitted, and the repeat loop starts with the first S-expression after varlist). exe through exen ,are evaluated in order, and then control loops back to exe, and the process is repeated. The loop exits when an S-expression after WHILE evaluates to NIL (exew), or an S-expression after UNTIL evaluates non-NIL (exeu). Value of the REPEAT function is that of the S-expression which caused the exit. More than one WHILE or UNTIL may be present.

The REPEAT can also be exited at any time using the RETURN function, in the same manner as PROG. There are no labels in REPEAT, however, so GO will not work.

I Chapter 9

Equality

The concept of equality is a key one for many ALISP functions. It is easy to define equality for litats, since they are all stored uniquely. In general, however, different ALISP data types have different meanings for equality, and different ALISP functions test for different types of equality among data types. The purpose of this section is to define carefully the various types of equality present in the ALISP system, and the functions which call on them.

9.1 Pointer Equality

This is the simplest type of equality. Two ALISP pointers (see section I.2) are equal if they have exactly the same bit pattern. The function which indicates pointer equality is EQ, a SUBR of two arguments. EQ returns T if both its arguments are exactly the same ALISP pointer, NIL if not.

This type of equality is most useful for litats and SNUM's. SNUM's and litats which print the same are always EQ to each other (except, of course, if a litat has been WIPE'd; see section I.5.1.2). Note that LNUM's and BNUM's, even if they have the same numeric value, will in general not be EQ; and that list structures, even if they look the same at read or print time, are not EQ unless they are exactly the same list in core. A few examples of the EQ function are given in Dialogue 9.1 below.

Pointer equality is used by almost every ALISP pre-defined function which must check for equality of two expressions. The GO function uses it when searching for a label in a PROG body, so that both litats and SNUM's are valid PROG labels. The plist functions use it when searching for labels on property lists, so that SNUM's and litats are valid property labels. Two functions which check for inclusion of atoms in a list structure use pointer equality: MEMBER and MEMB.

MEMBER is a SUBR of two arguments. The first is an S-expression to be searched for, the second is a list to search. MEMBER checks the first argument against successive top-level elements of the second. If one is found which is EQ, the list starting from that element is returned; else NIL is returned.

Dialogue 9.1
The Function EQ

?(EQ NIL NIL) T	Both NIL and T evaluate to themselves.
?(EQ T T) T ?(EQ 'FOO 'FOO) T ?(EQ 0 0) T ?(EQ -123 -123) T	SNUM's can be compared with EQ.
?(EQ 0 NIL) NIL	Zero and NIL are not EQ, even though they have the same address pointer of zero (see section I.2).
?(EQ 1.0 1) NIL	BNUM's and SNUM's are never EQ to each other.
?(EQ '(FOO) '(FOO)) NIL	These are two different list structures internally, even though they print the same.
?(SETQ X '(FOO)) (FOO) ?(EQ X X) T	The value of X is EQ to itself, since it is the exact same internal list structure.

MEMB is also a SUBR of two arguments. It searches every level of its second argument for an S-expression EQ to its first argument; if successful, MEMB returns T, else it returns NIL. Note that both MEMB and MEMBER, if given an atomic second argument, return NIL. Examples of these two functions are found below in Dialogue 9.2.

9.2 Numeric Equality

This equality is useful when comparing the values of the various ALISP number types. The function EQP, a SUBR of two

Dialogue 9.2
The Functions MEMB and MEMBER

```
?(MEMBER 'FOO '(FOO BAR MOO))
(FOO BAR MOO)
```

MEMBER found FOO as the first element of its second argument, and so returned the list starting from FOO.

```
?(MEMBER 3 '(FOO 3 BAR))
(3 BAR)
```

SNUM's are valid labels to MEMBER.

```
?(SETQ X '(FOO (BAR) MOO)
?      Y (CADR X))
(BAR)
?(MEMBER Y X)
((BAR) MOO)
```

```
?(MEMBER '(BAR) '(FOO (BAR) MOO))NIL
```

Since MEMBER uses EQ in searching a list, it found the tag (BAR) which Y was set to. Note that the next example does not succeed, because the list (BAR) is a different internal structure in the separate arguments to MEMBER.

```
?(MEMBER 'FOO '(BAR (FOO MAR) MOO))NIL
```

MEMBER searches only the top level of a list.

```
?(MEMB 'FOO
?    '(BAR (MOO(FOO))NIL MAR))))
T
```

MEMB searches all levels of a list, and returns only T or NIL.

```
?(MEMB 'FOO 'FOO)
NIL
```

The second argument to MEMB and MEMBER must be non-atomic to succeed.

---

arguments, does numeric equality testing. It works with any mixture of LNUM, BNUM, and SNUM arguments. The algorithm used is:

let d = abs (arg1*fuzz)

then arg1-d≤arg2≤ arg1+d

where abs is the absolute value function, and fuzz is a comparison tolerance. The value used for fuzz is a BNUM contained in the value cell of the atom FUZZ; initially, it is 2E-5. The user can reset FUZZ to any comparison tolerance desired, but it must be a BNUM, or an ARG-ERR will be issued at the next EQP call. This algorithm works pretty well, and assures that zero is never equal to anything but zero.

EQP will complain if given anything but LNUM, SNUM, or BNUM arguments. Some examples of the EQP functions are given below in Dialogue 9.3.

Dialogue 9.3
The Function EQP

```
?(EQP 0 0.0)
T                                        Zero is only EQP to zero.
?(EQP 0 .0000000000000001)
NIL

?(EQP #12 10.0)
T
?(EQP #-77 -63)
T
?(EQP 24E3 24E4)
NIL                                      Different number types can be
                                         compared.     LNUM's   are
                                         considered as 16-digit octal
                                         integers, with sign.

?FUZZ
.2E-5
?(EQP 1 1.0000000001)
T                                        The comparison tolerance of
                                         FUZZ is used by EQP.

?(SETQ FUZZ 2.0)
.2E1
?(EQP 2 3)
T
?(EQ 2 3)
NIL                                      If FUZZ is reset to a large
                                         enough    BNUM,    ridiculous
                                         comparisons can be made. Note
                                         that EQ does not use FUZZ in
                                         comparing SNUM's.
```

Three functions compare numbers to zero. They are all SUBR's of one argument, and take any of SNUM, BNUM, or LNUM types.

ZEROP returns T if its argument is zero, NIL if not. Note that (ZEROP x) is different from (EQ x 0), sine ZEROP will work with LNUM's and BNUM's as well as SNUM's. Negative zero can exist for LNUM's, and ZEROP returns T in this case.

PLUSP returns T if its argument is positive (zero included). LNUM's are considered negative if their high-order bit is set; thus an LNUM negative zero yields NIL from PLUSP.

MINUSP returns T if its argument is negative.

All three of these functions will complain with a NUM-ERR if given anything but an SNUM, BNUM, or LNUM argument.

Finally, the function ODDP, a SUBR of one argument, can be used to determine if a number is odd or even. ODDP takes either SNUM, BNUM, or LNUM arguments, but it truncates BNUM's to their integer part (if the BNUM is larger than 2B47 - 1, it is always considered to be even). LNUM's are treated as 48-bit signed integers. ODDP returns T if its argument is odd, NIL if not.

9.2.1 Numberic Inequality

While EQP can tell if two numbers are equal to within a certain tolerance, it is often useful to know which of two numbers is larger or smaller than the other. The functions GREATERP and LESSP, both SUBR's of two arguments, provide this facility.

GREATERP returns T if its first argument is numerically greater than its second; returns NIL if not. LESSP returns T if its first argument is numerically less than its second; returns NIL if not. Both functions accept any combination of SNUM, BNUM, or LNUM arguments. Because of the comparison tolerance FUZZ used in EQP, two numbers may be both EQP to each other and LESSP or GREATERP than the other. Examples of these two functionss may be found below in Dialogue 9.4.

Also useful for numeric comparison are the functions MAX and MIN, both SUBR*'s, which take an arbitrary number of numeric arguments and return the numeric maximum and minimum, respectively. At least two arguments must be given to these functions, and all arguments should be SNUM's, BNUM's, or LNUM's. Again, LNUM's are considered to be 47-bit integers with sign. The functions MAX and MIN always return their results as the same data type as given; see the examples in Dialogue 9.5 below.

9.3 List Structure Equality

Dialogue 9.4
The Functions GREATERP and LESSP

```
 ?(SETQ FOO 1.0000000001)
.10000000001E1
 ?(EQP FOO 1)
T
 ?(LESSP 1 FOO)
T
 ?(GREATERP FOO 1)
T
```

With FUZZ set to .2E-5, these two numbers are equal. Still, LESSP finds that 1.0 is less than 1.0000000001.

```
 ?(LESSP 1 1)
NIL
 ?(GREATERP #12 10)
NIL
```

LESSP and GREATERP tend to be better behaved with fixed numbers. Note the mixed modes.

Dialogue 9.5
The Functions MAXand MIN

```
 ?(MIN 1 1.001 -3 #12)
-3
 ?(MIN 1 1.001 #12)
1
 ?(MIN #12 61E3 126)
#12
 ?(MAX 3 4 5 6 7 #-66)
7
 ?(MAX 3 4 26E2)
.26E4
```

Note the use of mixed modes in these examples.

The function EQUAL is used to test equality of list structures. Two list structures are EQUAL if they have the same form and the same (EQ or EQP) atoms at the same points in their structure. Numeric atoms are testedwith EQP, litats with EQ. Because EQUAL checks each node of two list structures, it is much slower than EQ, but it is also the only way to check for equivalent list structures in a system where they are not uniquely stored.

EQUAL is also useful in testing for atomic equality, where it is unknown beforehand whether the arguments are numeric or not. (EQ x y), for example, will not work if x and y are LNUM's and BNUM's, while (EQP x y) will complain if x or y are non-numeric. (EQUAL x y), on the other hand, will use EQP if x or y is a number, EQ if they are litats. Examples of the EQUAL function will be found in Dialogue 9.6 below.

Dialogue 9.6
The Function EQUAL

```
 ?(EQ '(FOO BAR) '(FOO BAR))
NIL
 ?(EQUAL '(FOO BAR) '(FOO BAR))
T
```

This example points up the difference between EQ and EQUAL. EQ returned NIL because its two arguments were not the same list internally, while EQUAL returned T because they were the same list structurally.

```
 ?(EQUAL E E.0)
T
 ?(EQUAL 'A 1.0)
NIL
```

EQUAL acts like EQP on numbers, except it will not complain if given a non-numeric argument.

```
 ?(EQUAL '(1.0 FOO) '(1 FOO))
T
 ?(EQUAL '(FOO (BAR #12) MOO)
 ?       '(FOO (BAR 10) MOO))
T
```

EQUAL, like EQP, will correctly compare mixed number types.

```
 ?(EQUAL '(FOO (BAR)) '(FOO BAR))
NIL
```

An example of non-EQUAL lists. The two arguments to EQUAL do not have the same structure, since in tthe first BAR is on the second level, while on the top level of the list.

9.4 Address Equality

For some purposes, e.g., in forming ordered binary trees, it is convenient to establish an ordering of ALISP data. The three functions EQ, ADDLT (ADDress Less Than), and ADDGT (ADDress Greater Than) provide this facility. Every ALISP data structure can be compared with these three functions; every ALISP data structure is either EQ, ADDLT, or ADDGT every other. The ordering relationship is transitive, and exclusive. The comparison uses the internal addresses of the data to establish the ordering. Since different data are stored at different addresses in free space (except for SNUM's' see section I.2), the oordering automatically has the two properties mentioned. SNUM's are given addresses higher than any free-space address, so that they are always ADDGT than any other ALISP data type.

EQ returns T if its two arguments are the same ALISP pointer. ADDLT and ADDGT are both SUBR's of two arguments. ADDLT returns T if its first argument is less than its second in the ordering described above; else it returns NIL. ADDGT returns T if its first argument is greater than its second in the ordering described above; else it returns NIL. The exclusivity property of the ordering means that only one of EQ, ADDLT, and ADDGT will return T for the same two arguments.

Finally, if you wish to use the internal address of an ALISP data structure for your own devious purposes, the function INTADD (INTernal ADDress), a SUBR of one argument, is available. INTADD returns the internal address portion of its argument as an SNUM. Note that if INTADD is given an SNUM for an argument, it is an identify function; thus INTADD does not quite correspond to the ordering used by the function ADDLT and ADDGT. In particular, an SNUM and some other data type could have the same INTADD.

Examples of these address functions are given below in Dialogue 9.7.

```
 ?(SETQ FOO '(A B))
(A B)
 ?(SETQ BAR '(C D))
(C D)
 ?(INTADD FOO)
1067
 ?(INTADD BAR)
2913
```

These are the addresses of the two lists (A B) and (C D) in core.

```
 ?(EQ FOO FOO)
T
 ?(ADDLT FOO BAR)
T
 ?(ADDGT FOO BAR)
NIL
```

The value of FOO is of course EQ to itself. Note that the value of FOO is ADDLT the value of BAR, since its internal address is lower.

```
 ?(EQ 1067 FOO)
NIL
 ?(ADDGT 1067 FOO)
T
```

SNUM's are given a higher ordering value than any other ALISP data type.

```
 ?(INTADD 'FOO)
1392
 ?(ADDGT 'FOO FOO)
T
```

The address functions will compare all data types. Here an atom, FOO, with internal address of 1392, is compared with the value of FOO, the list(A B), with internal address of 1067.

I Chapter 10

List Manipulation

This section documents thos functions which operate on the plist, and accomplish destructive and non-destructive changes on non-atomic s-expressions in general.

10.1 Property List Functions

Property lists are not used by the ALISP system for holding atom values or function definitions, as they are in some systems. Instead, the interpreter relies on the value cells of litats; the property lists are the complete concern of the user.

There are two main reasons why property lists are useful to the LISP programmer. In the first place, property list values are much less volatile than litat value cells. The property list is not affected by lambda-expression or PROG bindings; it can be reached solely through the functions described below. Thus it is useful for holding things which remain relatively constant through the life of an ALISP run -- for example, the grammer rules used by MILISY (MIni-LInguistic SYstem) are stored on plists.

In the second place, plists offer a greater variety of indexing than value cells, and an easy means of storing and retrieving values through this index. Every value stored on a plist has actually two indices: the litat on whose plist the value resides, and the indicator label under which it is called. This double indexing scheme proves handy where the programmer must keep track of any similar items under different keys. For example, suppose that y9u wish to mark all litats that have been processed in a certain way. A simple, efficient solution would be to put the value T under the indicator PROCESSED on each litat's property list. Then, to check whether a particular litat X had been processed, it is only necessary to evaluate (GET 'X 'PROCESSED).

ALISP supports the standard functions for adding, removing, and fetching values from property lists. The format for property lists is:

(lab1 prop1 lab2 prop2 ... labn propn)

where the lab are labels (either litats or SNUM's) and the prop are any S-expressions. Every nlitat can have a property list associated with it.

Entries are added to the plist with the functions PUT and DEFPROP. PUT is a SUBR of three arguments. The first argument is an nlitat whose property list will be used, the second is a label, and the third is its associated value:

(PUT lit lab prop)

If lab is already on the plist of lit, then prop destructively replaces the property associated with it on the plist. If Lab is not on the plist, then a new entry of lab followed by prop is added to the front of the plist. Note that PUT and DEFPROP use EQ in searching for lab on the plist, and test only every other element of the plist. Thus atomic prop will not cause false matches on plist label searches.

DEFPROP, an FSUBR of three arguments, acts exactly the same as PUT, except it does not evaluate its arguments. Both functions return lab.

Properties are fetched from plists using GET and PROP. They are both SUBR's of two arguments, with format:

GET
(or lit lab)
PROP

They search the plist of lit (using EQ) for a match to lab; if found, GET returns the prop associated with it, while PROP returns the rest of the plist following lab. If lab is not found, both functions return NIL. IT is impossible to distinguish between GET returning a prop of NIL, and not finding lab at all, an ambiguity which is often useful. If it is nf it is necessary to distinguish the two cases, PROP can be used.

Properties are removed from the plist with the function REMPROP, a SUBR of two arguments. Format is the same as that for GET or PROP above. REMPROP searches the plist of lit (using EQ) for a match to lab; if one is found, it and its associated prop are destructively deleted from the plist. If no match is found, no action is taken. The value of REMPROP is NIL.

Finally the whole plist can be accessed with the function PLIST, a SUBR* of one or two arguments. With one argument, it returns the complete plist of that argument. With two arguments, it sets the plist of its first argument to the second argument.

Examples of all these plist functions will be found below in Dialogue 10.1.

10.2 Non-destructive List Manipulation

Dialogue 10.1
The Plist Functions

```
?(PLIST 'FOO)
NIL
```

The plist of FOO is initially NIL.

```
?(PUT 'FOO 'BAR 26)
BAR
?(PLIST 'FOO)
(BAR 26)
?(GET 'FOO 'BAR)
26
?(PROP 'FOO 'BAR)
(26)
```

The plist of FOO now contains the indicator BAR and the value 26. Note that GET returns the value, while PROP returns the rest of the plist, starting just after the indicator.

```
?(DEFPROP FOO MOO NIL)
MOO
?(GET 'FOO 'MOO)
NIL
?(PROP 'FOO 'MOO)
(NIL BAR 26)
?(PLIST 'FOO)
(MOO NIL BAR 26)
```

Now the plist of FOO has the indicator MOO and associated value NIL on its plist. Note that GET returns NIL, just as if the indicator MOO were not on the plist; but PROP will distinguish this case.

```
?(REMPROP 'FOO 'MOO)
NIL
?(PROP 'FOO 'MOO)
NIL
```

REMPROP removes the indicator MOO and its value NIL.

```
?(DEFPROP FOO 26 BAR)
26
?(PLIST 'FOO)
(26 BAR BAR 26)
?(GET 'FOO 26)
BAR
```

SNUM's are valid indicators on property lists, since the search functions use EQ.

```
?(PLIST 'FOO '(MOO MAR BOO BAR))
FOO
?(PLIST 'FOO)
(MOO MAR BOO BAR)
```

PLIST can change the whole plist at once.

These functions form results from their arguments without changing the original arguments."

10.2.1 Of CAR's and CDR's

These standard functions are SUBR's of one argument. Neither will work on atomic arguments, except for NIL. The CAR and CDR of NIL both return NIL.

Combinations of CAR's and CDR's can be performed with the functions CAAAR through CDDDR.

Multiple CDR's can be performed with the function CDRS, a SUBR of two arguments. The first argument is a list (or NIL) on which to apply the CDR's, the second is an SNUM specifying the number of CDR's to be taken. If zero or negative, no CDR's are taken, and the original first argument is returned. Excessive CDR's past the end of the first argument just return NIL.

The first several elements of a list can be fetched with the functio CARS, a SUBR of two arguments. The first argument is a list whose elements are to be extracted, then second is an SNUM specifying the number of elements to be taken; if zero or negative, NIL is returned. If the second argument specifies more elements than the first has, a top-level copy of the first argument is returned. Note that CARS creates a new list structure, calling CONS implcitly.

The LAST function, a SUBR of no arguments, returns the last element of a list. If given an atomic argument, LAST returns it.

The function ARGN can be used to return a specific element of a list. It is a SUBR of two arguments; the first is a list, the second an SNUM specifying the element of the list to be returned. If the second argument is less than or equal to zero, or larger than the length of the first argument, NIL is returned. ARGN is chiefly useful in lambda-expressions of an indefinite number of arguments (see section I.6.3.1).

The function LENGTH, a SUBR of one argument, returns the number of elements in that argument. If its argument is atomic, it returns zero.

Examples of these functions will be found in Dialogue 10.2

below.

10.2.2 List Construction

These functions construct new lists from their arguments, using the single primitive CONS implicitly (CARS above is also one of this group). No worries about destroying the original list structures with these functions; however, they have the disadvantage of using up free storage.

CONS is the standard function, a SUBR of two arguments. Its result is the dotted pair:

    (arg1, arg2)*

CONCONS takes a variable number of arguments, being a SUBR*, and strings them together using CONS. Its result is the S-expression:

    (arg1 arg2 ... argn,argm)

CONCONS must have at least two arguments. It is equivalent to LIST if its last argument is NIL.

LIST is a SUBR* of at least one argument. It forms a true list of its arguments:

    (arg1 arg2 ... argn)

APPEND is a SUBR of two arguments, usually lists, which forms a result by merging its first argument with its scond. Consider the form:

    (APPEND arg1 arg2)

where arg and arg are both non-atomic S-expressions. APPEND first makes a top-level copy of arg, then stuffs arg into the last CDR of this copy. Suppose, for example, that arg = (FOO BAR MOO), and arg = ((NIL) MAR), then the result would b

    (FOO BAR MOO (NIL) MAR)

You can think of APPEND as forming a single list whose elements are the elements of arg and arg.

APPEND also works nicely for the special cases where arg and arg are atomic. If arg is atomic, APPEND simply returns arg. If arg is atomic, it gets sstuffed into the last CDR of a copy of

---

*In ALISP, a dotted pair is represented as (A,B) rather than (A.B), in order to prevent confusion with floating-point number syntax.

Dialogue 10.2
CAR, CDR, and Derivative Functions

```
 ?(SETQ FOO '(A B (C) D))
(A B (C) D)
 ?(CDR FOO)
(B (C) D)
 ?(CADDR FOO)
C
 ?(CAR NIL)
NIL
 ?(CDDR NIL)
NIL
```

CAR and CDR of NIL return NIL.

```
 ?(CDRS FOO 0)
(A B (C) D)
 ?(CDRS FOO 2)
((C) D)
 ?(CDRS, FOO 10)
NIL
```

CDRS does multiple CDR's on its first argument.

```
 ?(CARS FOO 0)
NIL
 ?(CARS FOO 2)
(A B)
 ?(CAR FOO 10)
(A B (C) D)
 ?(EQ FOO (CARS FOO 10))
NIL
```

CARS extracts elements from the beginning of a list, and uses CONS to create a new list with these elements. Note that this new list is not EQ to the first argument to CARS.

```
 ?(ARGN FOO 0)
NIL
 ?(ARGN FOO2)
B
 ?(ARGN FOO 10)
NIL
```

ARGN takes the nth element of a list. Note that if asked for an element not in the list, it returns NIL.

```
 ?(LENGTH FOO)
4
 ?(LENGTH NIL)
```

```
0
 ?(LENGTH 'FOO)
0
```

LENGTH returns the number of top-level elements in a list as an SNUM. Atomic arguments to FOO have zero length.

```
 ?FOO
(A B (C) D)
```

None of the above functions changed the original list.

---

arg; so that, if arg is a list and arg is NIL, APPEND just returns a copy of arg. (APPEND x y)is entirely equivalent to (NCONC (COPY x) y). Examples of the APPEND function are given in Dialogue 10.3 below.

Dialogue 10.3
The Function APPEND

```
     ?(SETQ A '(FOO BAR))
(FOO BAR)
 ?(APPEND A '(MOO (MAR)))
(FOO BAR MOO (MAR))
 ?A
(FOO BAR)
```

APPEND strings its arguments together at the top level. Note that the original list remains unchanged.

```
 ?(APPEND 'FOO '(MOO (MAR)))
(MOO (MAR))
```

If the first argument to APPEND is atomic, the second argument is returned.

```
 ?(APPEND '(FOO BAR) NIL)
(FOO BAR)
 ?(APPEND '(FOO BAR) 'MOO)
(FOO BAR,MOO)
```

If the second argument to APPEND is atomic, the first argument has it stuffed into its CDR.

```
 ?(APPEND '(FOO BAR,MOO) '(A B))
(FOO BAR A B)
```

Note that the final CDR of the first argument is always lost.

Two functions, a COPY and DCOPY, are provided for copying list structure. Both are SUBR's of one argument. COPY forms a top-level copy of its arguments by applying CONS to each element in its argument. DCOPY forms an in-depth copy of its argument, entirely re-creating its list structure down to atomic level. Their LISP equivalents are:

```
          (DE COPY (X)
               (COND ((ATOM X) X)
                    (T (CONS (CAR X) (COPY (CDR X)))) ))

          (DE DCOPY (X)
               (COND ((ATOM X) X)
                    (T (CONS  (DCOPY (CAR X)) (DCOPY  (CDR X))))
))
```

If they are given atomic arguments, COPY and DCOPY simply return them.

The function REVERSE, a SUBR of one argument, reverses the order of the top-level elements in that argument. If its argument is atomic, it is simply returned. A non-atomic argument to REVERSE should be a true list; if it is not, the last CDR in the argument is lost when the reversal is performed. Examples of the REVERSE function will be found below in Dialogue 10.4.

Dialogue 10.4
The Function REVERSE

?(REVERSE 'FOO) FOO	REVERSE just returns its argument if atomic.
?(SETQ A '(FOO BAR)) (FOO BAR) ?(REVERSE A) (BAR FOO) ?A (FOO BAR)	The top-level elements of the argument to REVERSE where reversed. Note that the original list was not changed.
?(REVERSE '(FOO (BAR (MAR MOO)) NU)) (NU (BAR (MAR MOO)) FOO)	Only the top-level elements of a list are reversed.
?(REVERSE '(FOO BAR,MOO)) (BAR FOO)	The final CDR of a reversed list is always lost.

10.3 Destructive List Manipulation

Unlike the non-destructive list functions, the functions described in this section actually change already existing structures, rather than creating new ones. Among other advantages, these functions are faster and use less free space than their non-destructive counterparts. However, they can also screw up existing list structures if used incautiously, creating such usually undesirable structures as circular lists.

10.3.1 RPLACA, RPLACD, NONC

RPLACA and RPLACD are the standard functions, both SUBR's of two arguments. RPLACA replaces the CAR of its first argument with the second argument, RPLACD replaces the CDR of its first argument with the second. Both return the altered first argument as a result.

Both these functions will given an error if called with an atomic first argument. Note that their effects are permanent, as the examples in Dialogue 10.5 below indicate.

Dialogue 10.5
The Functions RPLACA and RPLACD

```
 ?(SETQ FOO '(A B C))
(A B C)
 ?(RPLACA FOO 'BAR)
(BAR B C)
 ?FOO
(BAR B C)
```

RPLACA replaced the CAR of the list (A B C) with BAR. Note that its effects are reflected in all pointers to the list it changed, i.e., it alters extant list structure.

```
 ?(RPLACD (CDR FOO) 'MOO)
(B,MOO)
 ?FOO
(BAR B,MOO)
 ?(RPLACD (CDR FOO) '(MOO MAR))
(B MOO MAR)
 ?FOO
(BAR B MOO MAR)
```

RPLACD replaces the CDR of its first argument. It thus has the power to change the length of a list.

NONC, a SUBR of two arguments, acts just like APPEND except that it does not copy its first argument. It permanently changes list structure by making the last CDR of its first argument point to its second argument. If its first argument is atomic, the second argument is returned. Examples of this function may be found in Dialogue 10.6 below; compare to the examples of APPEND in Dialogue 10.3 above.

Dialogue 10.6
The Function NONC

```
?(SETQ FOO '(A B COO
(A B C)
?(NONC FOO '(D E))
(A B C D E)
?FOO
(A B C D E)
```

NONC changes internal list structure; therefore the value of FOO was implicitly changed by the NONC call.

```
?(NONC FOO 'BAR)
(A B C D E,BAR)
?FOO
(A B C D E,BAR)
```

An atomic second argument is stuffed into the last CDR of the first argument.

```
?(NONC 'FOO '(MOO MAR))
(MOO MAR)
```

Atomic first arguments are ignored, NONC returns the second argument.

```
?(SETQ MOO '(F G H I))
(F G H I)
?(NONC FOO MOO)
(A B C D E F G H I)
?MOO
(F G H I)
```

Only the first argument to NONC has its list structure altered. Note that the first argument to NONC always loses its last CDR.

CONC is like NCONC except that it is a SUBR* and can thus take a variable number of arguments (but always at least two). The following two expressions are equivalent:

```
(CONC arg1 arg2 ... argm argn)
```

(NONC arg1 (NONC arg2 ... (NONC argm argn)))

10.3.2 Element Functions

A common operation in LISP is the addition or deletion of an element from a list, using the element position as an argument. In this form lists are treated as variablesize vectors of elements, the first (leftmost) element being numbered by 1, the second by 2, etc. The total number of elements in the list is given by the LENGTH function.

The functions ADDEL and DELETEL allow elements to be added and deleted from a list by specifying an element position. The format for ADDEL, a SUBR of three arguments, is:

(ADDEL new lis pos)

where new is the element to be added, lis is the list to add it to, and pos is an SNUM specifying the element after which new will be inserted. If pos is zero, new is added as the first element of lis. If pos is negative or greater than the number of top-level elements in lis, an ARG-ERR is issued.

Note that ADDEL actually changes the internal structure of lis, so that all pointers to it will point to the altered structure. If lis is atomic, there is no structure to alter, and ADDEL simply returns a list of one element, new.

DELETEL, a SUBR of two arguments, deletes elements from a list. Its format is:

(DELETEL lis pos)

where lis is a non-atomic list and pos is an SNUM specifying element in lis to be deleted. pos must be greater than zero and less than or equal to the number of the elements in lis, or an ARG-ERR is issued. DELETEL returns the altered list as its value.

Note that DELETEL actually changes the internal structure of lis, so that all pointers to it will point to the altered structure. It is, however, impossible to delete the last element from a oneelement list by altering its structure. In this case, DELETEL returns the expected value NIL (an empty list), but does not change the structure of lis. Examples of the ADDEL and DELETEL functions are given below in Dialogue 10.7.

The function EFFACE is usedto remove an element of a list by name. EFFACE, a SUBR of two arguments, searches its first argument for a top-level element EQ to its first argument. If none is found, EFFACE returns its second argument unchanged. If an occurrence is found, EFFACE deletes the first such occurrent

Dialogue 10.7
The Functions ADDEL and DELETEL

```
?(SETQ FOO '(A B C))
(A B C)
?(ADDEL 'D FOO 2)
(A B D C)
?FOO
(A B D C)
```

ADDEL added the atom D after the second element of the list. The list was permanently altered.

```
?(SETQ BAR (CDR FOO))
(B D C)
?(ADDEL 'F FOO 2) (A B F D C)
?BAR
(B F D C)
```

This illustrates the effect of the list-altering functions on all pointers to a list. BAR has as its value the CDR of the value of FOO, i.e., the list (B D C). When ADDEL changed the structure of the list which was the value of FOO, it also changed the value of BAR, since the value of BAR was part of the same list structure.

```
?(SETQ FOO NIL)
NIL
?(SETQ FOO (ADDEL 'A FOO 0))
(A)
?FOO
(A)
```

Here is the correct way to use ADDEL with empty lists. Since the value of FOO was NIL, ADDEL could not really alter any list structure. It simply returned a list of the single element A. Now using SETQ causes the value of FOO to be set to the list returned by ADDEL, namely, (A). Note that the SETQ expression will work even when the value of FOO is non-NIL, since ADDEL returns the altered list as its result.

```
?(DELETEL BAR 4)
(B F D)
?BAR
(B F D)_
?FOO
```

DELETEL removes an element of a list, permanently altering that list's internal structure. Note that the value of FOO is affected.

```
?(SETQ FOO '(A))
(A)
?(DELETEL FOO 1)
NIL
?FOO
(A)
```

WHen the value of FOO is a one-element list, DELETEL cannot remove that final element, even though it returns NIL as its result.

```
?(SETQ FOO (DELETEL FOO 1))
NIL
?FOO
NIL
```

This is the correct way to use DELETEL on one-element lists. Note that it will also work correctly when the value of FOO is a long list.

---

from the list, and rturns the altered list as its result. If the second argument to EFFACE is atomic, EFFACE returns NIL. If the second argument is a one-element list, EFFACE does not alter its list structure but still returns NIL. Examples of the EFFACE function are given below in Dialogue 10.8.

Dialogue 10.8
The Function EFFACE

```
?(SETQ FOO '(A B (FOO BAR) 4))
(A B (FOO BAR) 4)

?(EFFACE 'A FOO)
(B (FOO BAR) 4)
?FOO
(B (FOO BAR) 4)
```

EFFACE rubbed out the first occurrence of the atom A. Note that the value of FOO was changed: EFFACE alters internal list structure.

```
?(EFFACE 4 FOO)
(B (FOO BAR))
?(EFFACE '(FOO BAR) FOO)
(B (FOO BAR))
```

Since EFFACE uses EQ in searching, SNUM's and litats are found, but not list structures in general.

```
?(SETQ FOO '(A))
(A)
?(EFFACE 'A FOO)

NIL
?FOO
(A)
```

EFFACE cannot delete the last element of a list.

```
?(SETQ FOO (EFFACE 'A FOO))
NIL
?FOO
NIL
```

This is the correct way to use EFFACE with one-element lists.

I Chapter 11

Arithmetic

This section discusses the various functions available in ALISP to perform arithmetic operations. The three numeric types (LNUM's, BNUM's, and SNUM's) have already been discussed in sections I.2 and I.3; predicates for numeric comparisons were discussed in section I.9. PNUM's are not allowed as arguments to arithmetic functions.

11.1 Mixed Modes

Most of the ALISP arithmetic functions, both dyadic and monadic, can be used with all three number types (SNUM's, BNUM's, and LNUM's; PNUM's and ANUM's are not valid arguments to the arithmetic functions). The type of number they return as a result depends upon the types of their arguments and the function involved. In general (except for the logical and bit functions), they return an SNUM if all their arguments were SNUM's and the result is in SNUM range; otherwise they return BNUM's.

11.1.1 Number Type Predicates

Several predicates are provided to differentiate between the various number types. They are all SUBR's of one argument; they return T if their argument is a particular number type, NIL if not. Note that their arguments do not have to be number types; if they are not, these functions simply return NIL.

FIXP returns T if its argument is an SNUM, NIL if not.

FLOATP returns T if its argument is a BNUM, NIL if not.

LOGP returns T if its argument is an LNUM, NIL if not.

NUMBERP returns T if its argument is a number type (including PNUM and ANUM, NIL if not.

11.1.2 Number Type Conversion

To convert between modes, three functions are available, all SUBR's of one argument.

FIX converts to SNUM's. If its argument is out of SNUM range, a NUM-ERR is issued. If its argument is an SNUM, FIX simply returns it.

FLOAT converts to BNUM's. If its argument is a BNUM, FLOAT creates and returns a new BNUM having the same value.

LOGICAL converts to LNUM's. If its argument is out of LNUM range, a NUM-ERR is issued. If its argument is a LNUM, LOGICAL creates and returns a new LNUM having the same alue.

The three functions above can always be used if a result from an arithmetic operation must be a definite ALISP number type.

11.2 Dyadic Eunctions

The dyadic arithmetic functions are all SUBR*'s, except for REMAINDER, which is a SUBR of two arguments. The format is:

(fn arg1 arg2 ... argm argn)

where at least two arg's are present. The dyadicfunction is applied to the arguments from right to left, so that the result is:

(fn arg1 (fn arg2 ... (fn argm argn)))

11.2.1 PLUS, TIMES, DIFF

These functions return SNUM's if all their arguments are SNUM's, and the result is in SNUM range. If these two conditions are not met, they return BNUM's. Dyadic DIFF subtracts arg from arg. Examples of thse functions may be found below in Dialogue 11.1.

11.2.2 Division

Division offers special problems when dealing with different number types. Two functions are provided, DIVIDE and QUOTIENT, which always return BNUM's and SNUM's, respectively, no matter what the types of their arguments. QUOTIENT always truncates the result of each dyadic division, and retains only the integral part; DIVIDE returns the full floating-point result.

Dyadic QUOTIENT and DIVIDE divide arg by arg. If arg is zero, a NUM-ERR is issued. A NUM-ERR is also issued if the result of a divide operation is out of SNUM range for QUOTIENT, found below in Dialogue 11.2.

The function REMAINDER takes only two arguments. It does a floatingpoint divide of arg by arg, and returns the non-integral

Dialogue 11.1
Dyadic Arithmetic Functions

```
?(PLUS 1 3 -2)
2
```

Equivalent to 1+(3-2). Note that the result is an SNUM, since all three arguments were SNUM's and the result was within SNUM range.

```
?(PLUS 1 3.0 -2)
.2E1
?(PLUS #10 #12)
.18E2
?(TIMES 3000 16000)
.48E8
```

Mixed modes. Note that the result is always a BNUM if the arguments were not all SNUM's, or the result was out of SNUM range.

```
?(DIFF 6 3 4 1)
6
```

Equivalent to 6-(3-(4-1)).

---

of the operation. If both arguments were SNUM's, the result will be an SNUM; else it is a BNUM. Note that the concept of remainder is not well-defined for a floating-point division result whose absolute value exceeds 2B47; in this case, the remainder will be close to zero. arg cannot be zero.

11.3 Monadic Functions

Monadic arithmetic functions are all SUBR's of one argument. They can take all three numeric data types (FNUM's and ANUM's not included).

11.3.1 Trivial Monadic Functions

These four functions return SNUM's if their arguments are LNUM's or SNUM's and the results are not out of SNUM range. Else, they return BNUM's.

ADD1 and SUB1 add and subtract one from their respective arguments.

MINUS changes the sign of its argument.

ABSVAL returns the absolute value of its argument.

Dialogue 11.2
The Divide Functions

```
?(DIVIDE 6 4 2)
.3E1
```

Equivalent to 6/(4/2). Note that DIVIDE Always returns a floating-point result.

```
?(QUOTIENT 3.2 .6)
5
?(QUOTIENT 12 6.2 3.3)
12
```

QUOTIENT performs a floating-point divide of its last two arguments, then uses only the integer portion of the result in sucessive perations. Thus it divides 6.2 by 3.3 and truncates the result to 1, then divides 12 by 1 to return 12.

```
?(REMAINDER 3 2)
1

?(REMAINDER 3.2 .6)
.2E0
?(REMAINDER #10 6)
.4E1
```

REMAINDER performs a floating point divide of its arguments, but returns the non-integral portion of the result. If all of its arguments were SNUM's, the esult is an SNUM; else it is a BNUM.

11.3.2 Non-trivial Monadic Functions and RANDY

These functions all return BNUM's, no matter what the numeric type of their arguments. All except RANDY are SUBR's of one argument.

SIN and COS return the sine and cosine functions of their arguments. Arguments are in radians.

SQRT returns the square root of the absolute value of its argument.

EXP returns the exponential function of its argument.

LOG returns the natrual logarithm of its argument, which must be grater than zero.

RANDY, a SUBR of one or no arguments, will return a pseudo-random BNUM in the open interval (0,1) if called with no arguments. If called with one argument, a BNUM, the pseudo-random generator seed is reset using that number.

11.4 Logical Functions

These functions provide logical and shifting operations on LNUM data. They take only LNUM's as arguments, and always return LNUM results.

11.4.1 Boolean Functions

There are four boolean functions which perform bit-by-bit boolean operations on LNUM data. These functions are all SUBR's of two arguments (except for LOGNOT, a SUBR of one argument). The arguments must be LNUM's or a NUM-ERR will be issued. The boolean functions create a new LNUM as a result of their operation, and return it as a result. The original arguments remain unaltered.

LOGAND performs a logical and function.

LOGOR performs a logical inclusive or function.

LOGXOR performs a logical exclusive or function.

LOGNOT performs a logical complement function. It is a SUBR of only one argument; it performs the complement function on that one argument and returns a new LNUM result.

11.4.2 Shifting

Shifting of LNUM's is done by the functions CSHIFT (Circular SHIFTing) and ESHIFT (End-off SHIFTing), both SUBR's of two arguments. The first argument is an LNUM to be shifted, the second is an SNUM giving the shift count. The second argument must be in the range from -48 to 48.

Both these functions create new LNUM's for their results, so calling them uses one word of free storage. The original LNUM argument remains unaltered.

4 CSHIFT does circular shifting. If the second argument is positive, shifting is done right circular. If the second argument is negative, the shifting is done left circular, and the absolute value of the second argument is used as a shift count.

ESHIFT does end-off shifting If the second argument is positive, the shifting is done right end-off, and the sign bit is extended. If the second argument is negative, shifting is done left end-off, and the absolute value of the second argument is used as a shift count.

11.5 Bit Functions

The bit functions all operate on LNUM's. They provide the facility for changing and testing individual bits within an LNUM. All are SUBR's of two arguments, which the following format:

(bitfn lnum pos)

where lnum is the LNUM to be operated on, and pos is an SNUM giving the bit position within the LNUM. Bits are numbered from right to left, the lowest order (rightmost) bit being bit 1, the highest order (leftmost) bit being bit 48.

The bit functions do not create new LNUM's as results, but rather permanently change the lnum they are given as an argument (except for TSTBIT, which does not alter its argument). They thus do not use free storage at all.

With the bitfns, LNUM data can be accessed and set at the bit level. Using LNUM's and the bit functions, the user can store and access large numbers of binary values very cheaply (for LISP, that is).

TSTBIT tests individual LNUm bits. It returns T if bit pos of lnum is set, NIL if not.

CLRBIT clears the pos bit of lnum.

SETBIT sets the pos bit of lnum.

TOGBIT toggles (i.e., complements) the pos bit of lnum.

CLRBIT, SETBIT and TOGBIT return the altered lnum as their result. Examples of the bit functions are given below in Dialogue 11.3.

Dialogue 11.3
The Bit Functions

```
?(SETQ FOO #12)
#12
```

To the bit functions, the LNUM #12 looks like:

```
bit #:   ...7 6 5 4 3 2 1
value:   ...0 0 0 1 0 1 0
```

```
?(TSTBIT FOO 2)
T
?(TSTBIT FOO 6)
NIL
?FOO
#12
```

TSTBIT returns T if the gos bit of its first argument is set. Note that its first argument is not changed.

```
?(SETBIT FOO 1)
#13
?FOO
#13
```

SETBIT sets the gos bit of its first argument. Note that it changes the value of its first argument.

```
?(CLRBIT FOO 1)
#12
?(TOGBIT FOO 6)
#52
?FOO
#52
```

CLRBIT clears, and TOGBIT complements, the gos bit of the first argument. The value of the first argument is permanently altered.

I Chapter 12

Arrays and Strings

This chapter deals with two non-standard LISP data types, arrays and strings. Because they are non-standard, the user should read carefully the descriptions in this chapter before using them. They can offer significant advantages in storage and execution times for the right applications.

12.1 Strings

Strings are a compx,ŷy of storing text information. 7-bit ASCII characters are stored at most five per word, with a pointer to the next string word (see I.2.2). This represents a compromise between fully compact storage and the ability to point to different places in the same text.

12.1.1 String Manipulating Operations

There are two string functions, STRCARS and STRCDRS, which will return substrings from a given string.

STRCDRS is a SUBR of two arguments:

(STRCDRS str n)

where str is a string, and n is a positive SNUM. STRCDRS returns a string formed from str by deleting the first n characters. If n is zero, the original string is returned. If n is greater than the number of characters in the string, a null string is returned. Note that STRCDRS returns a pointer into str, rather than creating new string structures. Operations on the substring will affect the original string, since they are part of the same string data structure. STRCDRS can cause the original string to be spread out in free storage, to a minimum of one character to a string word.

STRCARDS is a SUBR of two arguments, like STRCDRS:

(STRCARDS str n)

where str is a string, and n is a positive SNUM. STRCARS returns a new string composed of the first n characters of str. If n is zero, or greater than the length of str, a complete copy of str is returned. Note that STRCARDS creates new string structure, and so uses free storage. STRCARDS can be used to copy and compact (to five characters per word) a string that has been spread out by the action of STRCDRS.

STRCONC is a SUBR* of two or more arguments:

(STRCONC str1 str2 ... strn)

where str through strn are strings. STRCONC concatenates these strings together, in order, to form a new string. No free storage is used; the old string structures are altered in place.

12.1.2 String Matching Functions

String matching involves checking whether one strin is a substring of another. There are two functions: STRTEST and STRFIND. The format for these is:

STRTEST
(or str1 str2)
STRFIND

These functions search str for a substring which matches str. STRTEST searches only the beginning of str, i.e., str must be an initial substring of str. STRTEST returns T or NIL, depending on the success of the match.

STRFIND will search str to find a substring which matches str at any position. If it finds a match, the first character position of the match in str will be returned, as an SNUM.

If str is empty, both functions find a match, STRFIND returning 0. Neither function will find a match if the lngth of str is greater than str; matches are only found if all of str is contained in str.

12.1.3 Comparing and Converting Strings

Strings can be compared with the functions EQS, LTS, and GTS (Section I.9). Strings are equal (EQS) if they match in all character positions and ar the same lenth. One string is less than another if it would appear before it in the dictionary (for characters which are not in the dictionary, order is defined by the ASCII codes in Appendix A).

The number of characters in a string can be found with ATLENGTH, a SUBR of one argument. It returns 0 for the null string. Strings are much like pnames, but are stored internally in different ways. To convert from one to another, the function INTERN is provided. A SUBR of one argument, INTERN will return a literal atom if given a string, and a string if given a literal atom. INTERN will issue an error if given a string argument whose length is greater than 322 characters, the pname length limit.

Reading and printing functions for strings are described in Chapters I.3 and I.4.

Arrays

ALISP arrays can have any number of dimensions, and each dimension can have length from 1 to 2 -1 (subject to core space limitations, of course). Arrays are kept in a special storage area called array space. Since arrays can be moved within this space in order to compact it, access to arrays is always through the array list, which is akin to an atom table bucket list (see ALISP internal specifications manual). Array pointers, also called ANUM's, are an ALISP 30-bit data type (see section), and point to the array list. They can be passed like any other ALISP data type, i.e., bound to variables, inserted into lists, etc. Array pointers always print as A#nnnn, where nnnn is the octal address of an array list word. Array pointers can not be read back in with READ or any other read function.

Arrays are useful for two reasons. First, the array index allows random access of any array element in constant time. Second, arrays are more compact than list structures (twice as compact, if the array header is not counted). They can thus save time and space if used correctly.

Array Types

There are currently three types of arrays: half-word (HW), floating-point numreric (BNUM), and logical numeric (LNUM).

1. HW arrays have elements which are ALISP S-expressions. The S-expression pointers are 30 bits, and thus packed two per word in the array. HW arrays are useful when a large number of S-expressions need to be stored using a numeric index.

2. BNUM arrays store floating-point numbers one per word. Array elements must be BNUM's; the array insertion functions will complain if given any other type. When an element is fetched from a BNUM array, a new floating-point number is created in free storage, and receives the array element. Thus successive accesses of a BNUM array will use up free storage. Also, if the same array element is accessed at two different times, the results of these accesses will not be EQ.

3. LNUM arrays store 16-digit signed octal numbers one per

word. The same remarks apply as for BNUM arrays.

Defining Arrays

Arrays are defined with either the function ARRAY or ARRAYQ. The function ARRAY has the format:

```
(ARRAY name type dim1 ... dimn)
```

All arguments are evaluated. name is a literal atom whose value cell will held the array pointer; type is the type of the array, as HW, LNUM, or BNUM; and dim through dimn are the SNUM array dimensions. ARRAY defines a new array of type type, and places an array pointer to the array in the value cell of name. If name is NIL, then ARRAY does not put the array pointer into a value cell, but simply returns it. The user must then save the array pointer so that he can reference it in the future.

ARRAYQ is the same as ARRAY, except all its arguments are unevaluated.

A newly-defined array is initially empty. For HW arrays all elements are NIL; for BNUM arrays all elements are 0.0; and for LNUM arrays all elements are #0.

Accessing Arrays

In the ALISP system, array pointers, or ANUM's, can be used as functions to retrieve or set elements of an array. In this respect they are similar to FNUM's, the machine language subroutine pointers which define system functions like CONS, CAR, and CDR.

Elements of an array are retrieved by using the array ANUM as a function of n arguments, where n is the number of array dimensions. For example, define array FOO by:

```
(ARRAYQ FOO HW 5 8)
```

so that FOO is a 5 × 8 array of ALISP S-expressions (actually, the value of FOO holds an ANUM pointer to the array). To set the 3,4 element of this array, evaluate:

```
(FOO 3 4)
```

EVAL checks specially for ANUM's, and interprets their arguments as indices to the array, returning the correct array element. Note that all arguments to an ANUM are evaluated, i.e., an ANUM acts like a SUBR.

Array elements can be set by using the array ANUM as a function of n + 1 arguments. The first argument to the ANUM is the new value for the array element, while the rest specify an element index. For example, to set the 3,4 element of FOO to the list (LIKES FIGS), use:

```
(FOO '(LIKES FIGS) 3 4)
```

Again, all arguments are evaluated.

If ANUM's are given too few or too many arguments, they will complain with an ARG-ERR. Also, indices other than positive SNUM's, or indices out of the array bounds, will also generate an error.

Auxiliary Array Functions

There are several helpful functions for finding out about arrays.

DIMS is a SUBR of one argument. If its argument is an ANUM, it returns a list of the dimension lengths, in the correct order. If not, returns NIL.

ARRTYPE, like DIMS, is a SUBR whose single argument should be an ANUM. It returns the array type as HW, LNUM, or BNUM.

ARRAYP can be used to tell if an S-expression is an ANUM or not. Returns T or NIL.

Reading and Printing Arrays

Special functions have been written to print out a complete array definition (including its contents), and to read it back in.

PRINARRAY is a SUBR of one argument, an ANUM. It prints the array defined by the ANUM on the current output unit. The format for the printed array is:

```
(NIL type (dim1 ... dimn) e1 e2 e3 ...)
```

where type is the array type, dim1 through dimn are dimension lengths, and the e's are the array elements, in row-major order. If the array is a large one, this could cause quite a large print-out.

READARRAY, a SUBR of no arguments, will read the next S-expression from the input buffer, and try to form it into an array. The S-expression should be in PRINARRAY format, except that the CAR of the S-expression may be a literal atom instead of NIL. READARRAY will create a new array having the dimensions, type, and elements indicated, and return an ANUM pointer to it. If the CAR of the S-expression is a literal atom, it will also place the ANUM in its value cell.

The @ character has been defined as a macro read character for arrays. It does an immediate call to READARRAY to form an array from the next S-expression in the input buffer. For example, a 3 x 4 array called FOO which looks like:

6.0	1.0	95.2	.06
4.2	1.0	100.6	.05
5.6	1.0	300.5	.04

could be defined by typing:

```
@(FOO BNUM (3 4) 6.0 1.0 95.2 .06 4.2 1.0 100.6 .05
     5.6 1.0 300.5 .04)
```

The @ macro returns an ANUM pointer to the array.

The filing functions know about arrays and how to correctly read and write them to files. However, there are restrictions on this ability, so it would be best to read Chapter II.1 if you intend to input and output arrays to files.

I Chapter 13

External Program Control

This section details three parts of the ALISP system which monitor ALISP programs: error control, interrupts, and tracing.

13.1 Error Control

References have already been made throughout this manual to certain conditions which cause the ALISP system to issue a program error. The general procedure followed on error detection, and methods for user control over error calls and traps, is the subject of this section.

13.1.1 Error Recovery Procedure and Backtracing

All ALISP errors are non-recoverable, that is, the program cannot be started again at the point at which the error occurred. An error causes complete abortion of the currently executing ALISP program (but see ERRSET for trapping), and eventually returns control to the top-level supervisor.

Most of the time, an error message is not sufficient to determine where an error occurred, especially if a complicated set of programs is being executed. A backtracing of function calls and variable bindings pendant at the time the error occurred will be printed if the atom BACKTRK is set to T (it is initially NIL). Variable bindings are printed, deepest bindings first; then the pendant functions, again deepest function calls first.

BACKTRK can take values other than T. In general, the value of BACKTRK is evaluated when an error occurs, before any other error processing takes place. At this point the user can do any processing he chooses, by calling an arbitrary ALISP function. The most useful probably BREAK (section I.13.2.2), which calls the break supervisor and allows the user to examine the environment at the point of the error. Use:

```
(SETQ BACKTRK '(BREAK 'ERROR T))
```

The BREAK is exited with (RETURN T) to print a backtrace, and (RETURN NIL) to get back immediately to top level. Examples of the BACKTRK switch will be found in Dialogue 13.1.

Dialogue 13.1
The Switch BACKTRK

```
?(QSETQ BACKTRK
?    (PROGN (PRINT 'YOU/ LOSE) T))))
(PROGN (PRINT 'YOU LOSE) T)            This sets the BACKTRK switch
                                       to a non-NIL value.

?(DE FLAMER (X Y) (CONS (CDR X Y))))))
FLAMER
?(FLAMER '(A B C D E F) 'BAR)

*** ARG-ERR FROM CDR
WRONG NO. OF ARGS                      The function FLAMER bombs
^                                      because the call to CDR was
                                       incorrect.
YOU LOSE                               Now the BACKTRK expression is
                                       evaluated, printing "YOU LOSE"
                                       and returning the value T.

BACKTRK
Y' BAR
X' (A B C D ... )

BACKTRK
CDR
CONS
FLAMER
?                                      Because the BACKTRK switch
                                       evaluated non-NIL, a backtrace
                                       is printed. First the
                                       variable bindings in effect
                                       when the error occurred are
                                       printed, then the functions
                                       whose execution was
                                       interrupted. Note the HALFPRI
                                       format used to print the
                                       variable bindings.
```

A backtracing of variables is normally printed with the function HALFPRI, so that a less wordy output is produced. The user can effect this in two ways: by setting HPRNUM so that HALFPRI prints more structure, or by using the switch BACKPRN.

If this atom is set to NIL, as it is initially, then the normal backtracing printout will occur. If, however, the atom BACKPRN is non-NIL, then special backtracing occurs. BACKPRN should be defined as a lambda-expression of one argument. When a backtrace is called, BACKPRN has its variable bound to the function name or bound variable currently being popped from the stack. BACKPRN can then print this variable, or perform any ALISP operation in general. When BACKPRN exits, then the next value is popped from the stack, and BACKPRN is called again with this new value bound to its variable. This process continues until the stack is emptied. For example, suppose that you only want to know if the functions CONS and COND are on the stack of function calls. Then simply do:

```
(DE BACKPRN (FN)
     (COND  ((EQ EN 'CONS) (PRINT FN))
            ((EQ FN 'COND) (PRINT FN)) ))
```

Examples of the use of the BACKPRN switch will be found in Dialogue 13.2 below.

The error recovery mechanism automatically cleans up the environment by popping and variable bindings of pendant lambda-expressions, PROG's, and REPEAT's. Note that any plist changes, or changes to list structures, or changes to values of literal atoms which are not variables; any of these changes are not undone after an error, and the user must provide his own functions for resetting these changes (see Dialogue 13.3).

For some applications, it is desirable to suppress any error printing that does occur, even the error message. This becomes especially important for a production system where the programmer does his own error control (with ERRSET), and wishes to shield the end user from even knowing he is in ALISP. If the switch ERRPRIN is NIL, no error message will be printed (although backtracing will occur if BACKTRK evaluates non-NIL). Initial value for ERRPRIN is T.

Because it is difficult to interpret what happens when ERRPRIN is NIL, this should only be done in a stable, well-debugged set of programs.

Of course, it could happen that an error is issued during error processing, for example, during the evaluation of the BACKPRN switch. If this happens, an unbreakable error loop could be established: evaluation of BACKPRN causes an error, which causes BACKPRN to be evaluated, which causes an error, etc. To prevent precisely this occurrence of events, the error processor will abort user control if an error is encountered during error processing. This means, essentially, that step (d) in the error recovery procedure is skipped; no backtracing control is done.

13.1.2 ERRSET Control

The function ERRSET, an FSUBR of two arguments, is used to provide error recovery or trapping within an ALISP program. Errors, no matter what kind, will not propagate beyond an ERRSET call.

Dialogue 13.2
The Switch BACKPRN

```
 ?(SETQ BACKTRK T)
T
```

BACKTRK must evaluate non-NIL for BACKPRN to be called on an error.

```
 ?(DE BACKPRN (X) (COND
 ?     ((EQ X 'CONS) (PRINT X))
 ?     ((EQ X 'COND) (PRINT X)))))))
BACKPRN
```

BACKPRN will now print the function names CONS and COND when they appear on the function all stack.

```
 ?(COND ((CONS (SETQ FOO (CONS NIL)) NIL) T)
 ?        (T T))))

*** ARG-ERR FROM CONS
WRONG NO. of ARGS

BACKTRK

BACKTRK
CONS
CONS
COND
```

No variable bindings are on the stack, but the function COND called CONS which called SETQ which called CONS again, so these function calls were printed by BACKPRN. Note that the function SETQ, which was also pendant, was not printed.

```
 ?(DE BACKPRN (X) (IF (EQ X 'VAR)
 ?      (PRINT (GETVAL VAR))))))
BACKPRN
```

This will print the value of the variable VAR if it appears on the stack.

```
 ?((LAMBDA (VAR)
 ?     ((LAMBDA (VAR) (CONS)) 1) 2))))

*** ARG-ERR FROM CONS
WRONG NO. OF ARGS

BACKTRK
1
2
```

```
BACKTRK
 ?
```

At the time of the error, VAR was bound by two lambda-expressions; in the first, to the value 1; in the second, to the value 2. BACKPRN printed these values when they were popped from the stack.

Dialogue 13.3
Variable Bindings Reset After An Error

```
 ?(SETQ FOO 'MOO BOO 'BAR)
BAR
 ?((LAMBDA(FOO) (PLIST 'FOO '(A B))
 ?          (SETQ FOO NIL BOO NIL)(CONS 'A)) 'MAR)

*** ARG-ERR FROM CONS
WRONG NO. OF ARGS
 ?FOO
MOO
 ?BOO
NIL
 ?(PLIST "FOO)
(A B)
```

The atom FOO, which was used by the lambda-expression as a variable, had its value correctly restored from the stack. The atom BAR, however, retained its value of NIL from within the lambda-expression, since it was not a lambda-variable. The plist of FOO also stayed at its setting within the lambda-expression; plists are never saved on the stack.

The ERRSET format is:

(ERRSET evalform errform)

where evalform and errform are any valid ALISP expressions. When ERRSET is called, it evaluates evalform using the EVAL function. If no error occurs during this evaluation, ERRSET returns a list of the result and exits. If an error does occur, the error recovery procedure (section 13.1.1) takes effect. Instead of popping all variable bindings to top-level, the error recovery procedure only backs bindings up to the level of the ERRSET, so that only variables found in *evalform* are restored. The error is affectively trapped within the ERRSET form. After bindings are restored, *errform* is evaluated (with EVAL) to perform any error processing the user may desire; and ERRSET exits with the value NIL. It is thus always possible to tell if an error occurred during an ERRSET evaluation: if ERRSET returns NIL, there was an error; if ERRSET returns a list, there was no error.

Since ERRSET traps all errors, it is possible to program loops that cannot be exited even with the interrupt facility. The following is the simplest example:

```
(PROG ( ) a (errset (PRINT 'EXECUTING) NIL)
            (GO A))
```

Unless the interrupt catches the evaluation outside of the ERRSET fom, this expression will just keep printiig the atom EXECUTING until the terminal phone is hun up o the CP time limit is reached. Interrupts will be trapped by the ERRSET.

12.1.3 User-defined Errors

The knowledgeable user may initiate his own errors with the function ERR, a SUBR of three arguments. ERR causes an immediate USER-ER type of error, and calls the error recovery procedure (13.1.1). Everything is the same as for a normal ALISP error, except the arguments of ERR specify the error message to be printed. The format for the ERR call is:

```
(ERR x message (y))
```

The error message format is:

```
USER-ER   FROM x
message
OFFENDING VAL = y
```

The first argument of ERR is printed as x if non-NIL. If NIL, neither the characters "FROM" nor x is printed.

The second argument of ERR is printed as message, if it is an nlitat or string. If it is not an nlitat or string, no message is printed.

If the third argument of ERR is non-atomic, then its CAR is printed as y. If it is atomic, no "OFFENDING VAL" message is printed.

ERR uses the normal error recovery procedure, so the ERRPRIN switch is in effect (section 13.1.1). If set to NIL, no message will be printed no matter what the arguments to ERR. Examples of the ERR function in action will be found below in Dialogue 13.4.

Dialogue 13.4
The Function ERR

?(ERR NIL NIL NIL) *** USER-ER	No values are printed, only the user error message.
?(ERR NIL 'FAILURE NIL) *** USER-ER FAILURE	A message (nlitat) was used as the second argument to ERR.
?(ERR 'FOO NIL '(BAR)) *** USER-ER FROM FOO OFFENDING VAL = BAR	Both x and y were specified. Note that the CAR of the last argument was used.

13.1.4 Time Limit and Timing Functions

The KRONOS and NOS operating systems maintain CPU and resource accumulators for a terminal session. If these accumulators reach a certain point, the message:

```
*TIME LIMIT*
or
*SRU LIMIT*
```

will be printed on the terminal. The user should respond either "T,nnn" or "S,nnn", respectively, where nnn is the number of units (seconds or SRU's) which will elapse before the next accumulator message. There is an absolute resource limit which the user cannot exceed, however; when this limit is exceeded, the user is ungraciously excluded from any further processing.

The function PARAMTL is available from ALISP to forestall the time limit error and to return the amount of CP time already spent in a terminal session. PARAMTL is a SUBR* of one or no arguments. With no arguments, it returns a two-element list specifying the current timing status of the ALISP job. The first element of the list is an SNUM giving the number of seconds of CP time used so far by the user in a terminal session; the second element is an SNUM giving the number of seconds in the time limit. The difference between the first and second elements is the number of CP seconds to go before a time limit will be issued.

With one argument, an SNUM, PARAMTL resets the value of the time limit to the SNUM. The SNUM should be less than or equal to the user's validation time limit; if it is not, the ALISP job will be summarily aborted by KRONOS.

There is no function for accessing or changing the SRU limit, a new addition to NOS. If it is necessary that ALISP not be interrupted by an SRU LIMIT message, appropriate NOS control cards can be issued from the batch subsystem before entering ALISP.

A millisecond timing clock local to the ALISP job is provided via the function RUNTIME. The RUNTIME clock can be set and fetched during the course of an ALISP job, or can be used to time the evaluation of an S-expression.

The function RUNTIME is an FSUBR* of one or no arguments. With no arguments, it simply returns the current value of the RUNTIME clock as a BNUM. The RUNTIME clock is tied to the executing ALISP job, i.e., every time the ALISP system uses CP time, the RUNTIME clock is updated. On entering the ALISP system, the runtime clock is initially set to zero.

If RUNTIME is given a BNUM argument, it resets the value of the RUNTIME clock to that argument. The value of the RUNTIME function is its argument. Thus, calling RUNTIME with the argument 0.0 will completely reset the RUNTIME clock.

If RUNTIME is given anything but a BNUM for an argument, it evaluates that argument and returns the evaluation time (in milliseconds) on the current output device, and prints the result of the evaluation. The RUNTIME clock is not reset.

Some examples of the RUNTIME function are given in Dialogue 13.5 below.

13.1.5 ALISP System Errors

There is a chance that at some point you will receive the following error message:

```
*** HALT FROM nnnn
OFFENDING VAL = m
```

where the n and m are digits. If so, this indicates an ALISP system error, a bug in ALISP, and it's my fault, not yours. Please save as much of your output as possible, including the error message; or write down the procedure which led to the error, and give it to me at one of the places listed at the end of the introduction. Prompt redress will be attempted.

The HALT or system error in itself did not harm the executing programs, and the normal error recovery procedure

Dialogue 13.5
The Function RUNTIME

```
?(RUNTIME)
.45E2
```

The RUNTIME clock gives the amount of CP time spent in the ALISP system, until it is reset. The value of the RUNTIME clock is in milliseconds; here, the ALISP job has aged 45 milliseconds of CP time.

```
?(RUNTIME 0.0)
.0
?(RUNTIME)
.2E1
```

Calling RUNTIME with a BNUM argument resets the RUNTIME clock to that argument. This is a handy feature if it is necessary to time some sequence of ALISP commands.

```
?(RUNTIME (CONS 'FOO 'BAR))
*RUNTIME=.1E1
(FOO,BAR)
```

RUNTIME with a non-BNUM argument evaluates that argument, then prints the CP evaluatiom time in milliseconds, and returns the result of theevaluation.

```
?  (RUNTIME)
.4E1
```

The RUNTIME clock accumulates CP time since the last reset.

---

should have unbound all bound variables, so that execution could proceed again from the top level. However, it is wise to save everything again, rather than continuing with a system which went down in a HALT. The reason for this is the system error may indicate that something is wrong internally with that particular ALISP run, and continuing to execute in it may hang the ALISP system.

13.2 Interrupts and Breaks

There are two basic types of interrupts in the ALISP system, both useful only under time-sharing, and so absent from a batch-run ALISP job. They are terminal interrupt and BREAK.

13.2.1 Terminal Interrupt

There is a single program interrupt available from the terminal. It is control-C on ASCII-type terminals and ATTN-S-ATTN on correspondence terminals. The results of the interrupt depend on the state of the ALISP system when the interrupt occurs.

If ALISP is executing a program, the interrupt causes a recoverable break in execution. The output and input buffers are emptied, and the message "BREAK FROM INTRFLG" is printed on the terminal. ALISP is now in a BREAK supervisor loop. Within this loop, the user can execute any ALISP function, examine and change the environment, etc.; see section 12.2.2 below.

When the user is through processing in the BREAK, he can either return control to the executing program at the point where it left off, or cancel execution of the program and return to the top level of ALISP (or to the nearest ERRSET trap, if one exists). To exit the BREAK and continue program execution, use:

```
(RETURN T)
```

To exit the BREAK and return to top level, use:

```
(RETURN NIL)
```

The effect of (RETURN NIL) is actually to cause an error, so that the message,

```
*** HALT FROM INTRFLG
```

will be printed on the SYSOUT device; and all the error processing detailed in section 12.1.1 above will take place.

Within a BREAK, the interrupt is still valid, so it is possible to have nested interrupts and BREAK's. A RETURN will then exit from the current BREAK to the previous one.

There are certain points during execution when an interrupt may not be honored, or behave in a strange way. Most ALISP user programs such as EDIT, INPUT, OUTPUT, etc., are protected from interrupts. A control-C typed during these programs will be ignored.

A control-C will act as an escape character (delete line) if it is typed after typing some input characters, but before a carriage-return.

If an interrupt occurs during the printing of an S-expression, the printing is aborted. This is useful for stopping long undesired printouts such as occur with circular lists.

If an interrupt occurs when ALISP is about to issue a read request, the read will sometimes be issued first, so that the user sees the "?" prompt. The return key should be pressed at this point, and the interrupt will proceed.

Above all, patience should be exercised when dealing with interrupts. The operating system will issue two line-feeds to let you know that your interrupt was accepted; it may take some time after that for ALISP to get around to processing it.

Within the BREAK, one of the most handy functions is the STACK function, a SUBR of no arguments. Evaluating (STACK) returns a list of those function calls pending during the BREAK, i.e., those functions which were executing when the interrupt was given. The list is in stack order, which means that in a nested series of function calls, the innermost ones are first on the list. All entries after the atom BREAK are those of the user's program. An example of the interrupt facility is given in Dialogue 13.6 below.

There are times when the user wishes not to cause an interrupt even when it is requested from the terminal. For instance, there may be sensitive portions of a program that if interrupted and fooled with in a BREAK will wreck the rest of the execution. To prevent unwanted interrupts, the switch INTRFLG is provided to disable the interrupt facility. If INTRFLG has the value NIL, an interrupt requests from the terminal will not be honored. Note that, if the request is not honored, it is thrown away completely. An interrupt request must occur when INTRFLG is set non-NIL; interrupts are not saved to cause delayed interrupts. The value of INTRFLG is initially T.

An interrupt BREAK uses the input and output buffers to communicate with the terminal, and so flushes them before entering the BREAK loop. This causes no harm unless you are doing input and output to files with printing or reading functions other than PRINT or READENT. In this case, some output or input (but never more than one line of each) may be lost when an interrupt occurs.

Dialogue 13.6
Terminal Interrupt

(control-C is represented by the character Ȼ)

?(PROG (FOO) (SETQ FOO '(A B)) ? A (CONS (CAR FOO) (CADR FOO)) ? (GO A)))))	`This expression has an infinite loop inside the PROG.
Ȼ	User hits interrupt here causing a BREAK.
BREAK FROM INTRFLG	Now in the top level of the BREAK, a READENT-EVAL-PRINT loop. An asterisk is the BREAK prompt character.
*FOO (A B)	The value of FOO is its binding in the PROG form.
*(STACK) (CAR CONS PROG)	The STACK function returns a list of pendant functions (excuse the APL terminology). Note the order: CAR, then CONS, then PROG. This is the reverse order from the way the function were entered on execution. The interrupt occurred while the CAR form was being evaluated.
*(RETURN T)	Program continues executing the infinite loop.
Ȼ	Another interrupt. This user must finally realize it's an infinite loop.
BREAK FROM INTRFLG *(RETURN NIL)	This time the RETURN gets back to the top level of ALISP, exiting the infinite loop.
*** HALT FROM INTRFLG ?	

13.2.2 BREAK

The BREAK function is an ALISP program. It starts a READENT-EVAL-PRINT loop at any point in an executing ALISP program, in which the user can check and reset values, examine list structures, or execute any ALISP function; then resume the program at the point where the BREAK was called.

Format for the BREAK function, a LAMBDA of two arguments, is:

(BREAK message pred)

where message is a litat whose pname will be printed when the BREAK is entered, and pred is a condition for the BREAK: if pred is NIL, BREAK exits with value NIL without entering the EVAL loop; if pred is non-NIL, the EVAL loop is entered.

When BREAK is called (pred is non-NIL), it first prints the following:

BREAK FROM message

Then it enters the READENT-EVAL-PRINT loop. At this point, the ALISP system acts just like the top level EVAL supervisor, except that the prompt for input is an asterisk rather than a question mark. Expressions typed at the BREAK supervisor are evaluated with EVAL (the SYS switch for different supervisors does not work in BREAK) and the results printed. Both SYSPRIN and * work in the BREAK evaluation loop, as well as SYSIN and SYSOUT (see section I.6).

All ALISP errors are trapped by the BREAK supervisor, and do not cause the BREAK to exit. If an error occurs, the error recovery procedure is invoked (section 12.1.1) and BREAK re-enters its EVAL loop, printing the BREAK message again. Interrupts within a BREAK cause another BREAK supervisor to be established; exiting this new BREAK causes the previous BREAK to be resumed. SYSIN and SYSOUT are initially set within the BREAK to 0, so that the BREAK supervisor addresses the terminal.

To exit from a BREAK, the form:

(RETURN x)

should be typed at the top level of the BREAK supervisor. The BREAK will exit with value x. For the special case of a control-C interrupt, x should be T to continue execution from the interrupt, and NIL to halt and return to ALISP top level.

Certain variables are set by BREAK and restored to their original values when the BREAK exits. If any of the values of these litats are changed during a BREAK call, they will be restored on exit from BREAK. These variables and their values within the BREAK are given in Table 13.1 below.

Table 13.1
BREAK Local Variable Values

variable	initial value
SYSIN	0 (i.e., the terminal)
SYSOUT	0
INUNIT	0
OUTUNIT	0
PROMPT	54B (i.e., asterisk)
SYSPRIN	T
*	NIL
TRACFLG	NIL
INTRFLG	T
BACKTRK	NIL
TTYCHAR	T
EOLR	T
EOLW	T
ERRPRIN	T

BREAK can be inserted into functions being debugged by use of the editing package (section II.3). It can be inserted at function definition time also. In all respects BREAK is treated as a normal function call within an ALISP program, that is, it takes its two arguments and returns a result. Examples of a BREAK call within an executing function are given in Dialogue 13.7 below.

Inserting a BREAK call into a function being debugged is only one possible use of the BREAK. It has the advantage that the user knows exactly where in his program the BREAK occurred.

There are two other uses of BREAK that are particularly handy. If a BREAK call is stuffed onto the value of BACKTRK, the BREAK will occur just after an error, so that the environment at the time of the error can be examined (see section 13.1.1 above for the use of BACKTRK). Then, when the user exits from the BREAK, he has the option of getting a backtrace printing. (RETURN T) will print it, (RETURN NIL) will exit without printing it.

Secondly, a BREAK call can be used as part of a function trace (see below, section 13.3). In this way, a BREAK can be called on a particular argument to a function, or a particular value returned by a function, or indeed any condition definable

Dialogue 13.7
The BREAK Function

```
?(BREAK 'FOO NIL)
NIL
```

If the second argument to BREAK evaluates to NIL, BREAK exits with value NIL.

```
?(BREAK 'FOO T)
```

The second argument to BREAK evaluated non-NIL, so the BREAK loop is entered.

```
BREAK FROM FOO

*(CONS 'A 'B)
(A,B)
*(PROG () (RETURN 'BAR))
BAR
*(RETURN '(MOO MAR))
(MOO MAR)
 ?
```

The BREAK supervisor is an EVAL supervisor. Note that it uses the asterisk prompt, so that the BREAK supervisor can always be distinguished from ALISP top level.

The first call to RETURN took place inside a PROG, so that the BREAK was still executing. The second call to return took place when no PROG was running; the BREAK exited.

```
?(DE FACT (X)
?  (COND ((ZEROP X)
?         (BREAK 'FACT T) X)
?   (T (TIMES X (FACT (SUB1 X))))))
FACT
```

This defines the recursive factorial function, with a BREAK call inserted when X is zero.

```
?(FACT 4)
```

The BREAK loop is entered while FACT is executing.

```
BREAK FROM FACT

*X
0
*FOO
```

Within the BREAK, the values of litats can be examined. Note that X is set to zero, as

```
*** VAL-ERR FROM FOO
BREAK FROM FACT
*(SETQ X 10)
```

it should be.

FOO is undefined, giving a VAL-ERR; but the BREAK traps errors and does not exit.

```
10
*(RETURN NIL)
240
```

The SETQ resets the value of X to 10 instead of 0. Since the value of X is returned as the value of FACT (see the definition above) the value returned by FACT is actually 10 times the normal factorial function. Note that the value of the BREAK function was not really used; (RETURN T) would have worked as well.

---

by an ALISP expression, just before or after a function is executed. The advantage to using BREAK with the trace facility lies in the flexibility and ease of calling the BREAK on a certain condition. Also, the BREAK call is never actually inserted into the traced function, so that there is no need to use the editor to restore the function to its original form once it has been debugged.

13.3 Tracing

"A good LISP system has a good tracking package", said a famous Chinese philosopher. In keeping with this tautology, a tracing facility of great power, flexibility, generality, and simplicity is available on the current ALISP system.

13.3.1 Simple Tracing

Tracing is the ability to observe programs in execution. In ALISP, the tracing facility is a superstructure on the executing programs; it does not change the form or manner of their execution, but simply observes what they do and reports back to the user.

Tracing is done only on nlitats (non-NIL literal atoms). A traced nlitat causes a tracing printout if it is used as the name of a function. This occurs when expressions of the form:

```
(FOO A B)
```

are evaluated. Here the nlitat FOO, if traced, would cause tracing messages to be printed, because it is being used as a function name.

Two messages are printed on every traced function. The first, when the function is called, gives the recursion level of the function call (zero is the top level of first call), the function name, and the arguments to the function. If the function evaluates its arguments, the evaluated arguments are printed (SUBR, SUBR*, and LAMBDA functions); if not, the unevaluated arguments are printed (FSUBR, FSUBR*, LSUBR, and FLAMBDA functions).

After the function is evaluated, the value it returns is printed, along with the function name and the recursion level. The recursion level of the value message matches that of the argument message, thus enabling the keen-eyed user to match up argument message with value message in a recursive function; see the example in Dialogue 13.8 below.

All tracing messages are printed out on the SYSOUT device. Since they use the output buffer to print their message, the buffer contents are changed by tracing; this could cause programs writing to files with functions other than PRINT to lose lines of their output.

To initiate tracing of a litat, simply use the function TRACE, an FSUBR*. With no arguments, TRACE returns a list of all nlitats currently being traced. With nlitat arguments, TRACE will turn on the tracing status of each of these arguments, and return NIL as its result. The tracing status of an atom can be turned on either before or after the atom is defined as a function; changes in the atom's value do not affect tracing status. If TRACE is given anything but nlitat arguments, it complains with an ARG-ERR (but see the exceptions noted in 13.3.2 below).

UNTRACE can be used to turn off tracing. It, like TRACE, is an FSUBR*. With no arguments, it turns off the tracing status of all nlitats. With nlitat arguments, it turns off the tracing status of each of those arguments only. UNTRACE always returns a NIL result.

At no time should the plist of TRACE be tampered with, as it is used for bookkeeping on the tracing status and recursion levels of nlitats.

An example of simple tracing is given in Dialogue 13.8 above.

All tracing can be temporarily turned off through use of the switch TRACFLG. The value of TRACFLG is tested before each tracing call; if NIL, the trace is not performed. TRACFLG does not affect the tracing status of any nlitat; if NIL, it simply

Dialogue 13.8
Simple Tracing

```
?(DE FACT (X) (COND ((ZEROP X) 1)
?       (TIMES X (FACT (SUB1 X)))))
FACT
```

This is the recursive factorial function.

```
?(TRACE)
NIL
```

No nlitats are currently being traced.

```
?(TRACE FACT)
NIL
```

This sets the tracing status of the nlitat FACT. Note that TRACE returns NIL as its value.

```
?(TRACE)
(FACT (0))
```

Now FACT is currently traced. The list (0) is used to hold the recursion level when a tracing printout is performed.

```
?(FACT 3)

0 ARGS of FACT
   3

1 ARGS of FACT
   2

2 ARGS of FACT
   1

3 ARGS of FACT
   0
```

The recursion level is printed first, then the ARGS OF message, and finally the function name and its arguments.

```
3 VAL OF FACT 1

2 VAL OF FACT 1

1 VAL OF FACT 2

0 VAL OF FACT 6
6
```

The recursion level is printed first, then the VAL OF fn

	message, then the value of the evaluation of fn. The recursion level is useful for matching the correct ARGS OF and VAL OF messages. The final 6 is the value of the (FACT 4) expression.
?(UNTRACE) NIL ?(TRACE) NIL	UNTRACE removes all tracing statuses.

---

prevents all tracing until it is set non-NIL.

The TRACFLG switch is especially handy when used in the ALISP editor, filing, and pretty-print functions. By binding TRACFLG to NIL, these programs temporarily halt all tracing without ruining the tracing status of functions the user wants traced when he runs his own programs. Thus, even if a user is tracing such a ubiquitous function as CONS, editor and filing functions will run without causing any tracing printout.

13.3.2 Conditional Tracing

This section describes the promised flexibility of the ALISP tracing facility. Simple tracing is fine for simple programs which do not recurse too deeply; but for more complex or lengthy programs reams of useless output can be generated--imagine the output for a simple trace of the factorial function of Dialogue 13.8 used with an argument of 1000.

The answer to tracing wordiness is conditional tracing: give control of the trace back to the user.

Conditional tracing is a simple extension of the simple tracing described in 13.3.1 above; there are only two modifications. When a traced function is entered, instead of printing its arguments, an entry function is evaluated. This entry function can call any ALISP function, for printing, reading, etc. If the evaluated entry function returns NIL, then no argument tracing occurs. If it returns a non-NIL result, then the argument message is printed as in simple tracing, on the SYSOUT device.

After the traced function has been evaluated (and despite the results of the evaluation of the entry function), an exit function is evaluated in the same manner as the entry function. Again, the result returned by the exit function signals the printing (non-NIL result) or non-printing (NIL result) of the value tracing message.

The entry-exit functions can be associated with an nlitat via the function TRACE. Instead of giving TRACE an nlitat argument, one gives it a list of the form:

```
(fn ENTRY entryfn EXIT exitfn)
```

where fn is the nlitat name of the function to be traced. Both the ENTRY and EXIT parts of the list are optional, or can occur in reversed order; the trace routines just look for the tag ENTRY and consider the S-expression immediately following it to be the entry function, and the S-expression after EXIT to be the exit function. In this form, a call to TRACE both sets the conditional entry and exit functions, and sets the tracing status of fn.

TRACE evaluated with no arguments will return the tracing conditions of all traced atoms, as a list of atoms followed by their tracing conditions. The initial zero in the tracing conditional list is used as the recursion level marker.

UNTRACE will clear tracing conditions in addition to the tracing status of an nlitat.

Within the conditional trace, several litats have values which could be useful.

ARGS holds a list of arguments to the traced function. These arguments are evaluated if the function is a LAMBDA, SUBR, or SUBR type. If there are no arguments to the function, *ARGS is NIL.

*VAL holds the result of the evaluation of the traced function. Before the function is evaluated, *VAL is set to the atom NOVAL.

*LEVEL holds the recursion level of the traced function, as an SNUM.

*TRACE is a tracing switch. If entryfn sets *TRACE to NIL, all tracing will be turned off during the evaluation of the traced function.

Conditional tracing expressions are limited only by the ingenuity of the user. A few examples of the capabilities of the conditional trace follow:

1. To trace only the first n recursion levels of a function FOO, use:

```
(TRACE (FOO
       (ENTRY (IF (EQ *LEVEL n)(SETQ *TRACE
        NIL))))
```

2. To cause a BREAK when the first argument of FOO is BAR, but to cause no tracing printout:

```
(TRACE (FOO
       ENTRY (BREAK 'FOO (EQ (CAR *ARGS) 'BAR))
       EXIT NIL))
```

3. To print the result of evaluating FOO only when it is non-atomic:

```
(TRACE (FOO
       ENTRY NIL
       EXIT (IF (LISTP *VAL)(PRINT *VAL) NIL)))
```

I Chapter 14

Allocations and Garbage Collections

Information on the various ALISP storage areas, and the routines used to maintain them, is contained in this section. For most ALISP programs, there is little need to worry about storage problems; but those users with large programs or heavy storage requirments should go over this section carefully in order to optimize their execution speed.

14.1 ALISP Storage Areas

There are four ALISP storage areas, called spaces. Descriptions of these spaces, and their initial storage allocations, are given below in Table 14.1.

Table 14.1
Initial Storage Allocations

Name	Initial Available Allocation (in decimal words)	Description
FREE	8000	Holds all ALISP data types except arrays.
PROG	1000	Storage for arrays and binary programs.
JPDL	500	Jump Push-Down List -- holds return addresses for recursive routines.
APDL/ SPDL	500	Argument/Special Push-Down List -- holds variable bindings and arguments to functions.

The allocations for these areas are not fixed. As a space becomes filled up, the ALISP system requests more storage for it, and expands the initial available space allocation. This expansion is automatic for all four spaces, up to the field length limit (see section 14.2 below). If a space has excessive storage assigned to it when it is not needed, that space is contracted. Contraction is automatic for all storage areas except FREE, which cannot be contracted at all. Contraction enables the ALISP system to have a smaller execution field length, thus increasing the ratio of CP time to rollout time for an ALISP job. The automatic expansion and contraction of space means that the execution field length of any particular ALISP job changes dynamically in response to program needs for storage, giving more efficient use of storage.

The algorithm used for deciding when a storage area size should be changed is fairly simple. There are three parameters for each storage area: a minimum size for unused space, a maximum size, and an increment size. If the unused space in a given storage area is below the minimum size parameter, then that storage area is expanded by the increment size. If unused space is above the maximum, it is contracted so that the amount of unused space is equal to the increment size. This algorithm works pretty well in the storage-eating programs run so far in testing the ALISP system. The parameters are not currently accessible by the user.

The function PARAMGC, a SUBR of no arguments, provides information about unused storage available in each of the four spaces. Evaluation (PARAMGC) returns a list of six SNUM's. The first four are the number of free words of core left in the four ALISP spaces (FREE, PROG, JPDL, and APDL, in that order); the last two are the number of garbage collects done since the last PARAMGC call, and the total number of garbage collects since the ALISP system was initiated. Note that the PARAMGC value for FREE space will actually give less than the total amount of FREE space left to the user, unless a GC has just been called. This is a consequence of ALISP's garbage collect mechanism, which does not reclaim previously used but inactive FREE storage until there is no more unused storage left.

14.2 Field Length Limit

The ALISP system has a maximum field length limit beyond which it will not expand. This limit is set when the ALISP system is initiated, by the FL parameter on the ALISP control card (see Appendix B); the default value is 64000B. From ALISP, the field length limit can be accessed and changed with the function PARAMFL, a SUBR* of one or no arguments. With no arguments, PARAMFL returns a list of three SNUM's, the first of which is the current execution field length, the second the ALISP field length limit and the last the absolute KRONOS FL limit for the user. With one argument, an SNUM, PARAMFL sets the ALISP field length limit to that argument, if it is larger than the current execution field length; if it is smaller, PARAMFL issues an ARG-ERR.

It does no harm to use a very large field length limit if you feel you might need it, since ALISP does its own dynamic storage allocation and will not use the excess unless and until it is necessary. On the other hand, if you are sure that you will not need that large an allocation of storage, it is reasonable to set a low field length limit, since this traps pathological errors such as infinitely recursive functions all the sooner.

When the user's execution field length approaches the field length limit, the storage re-allocation algorithm described in 14.1 is modified somewhat to try to squeeze every last word of the limit into the execution length. However, there comes a point where the ALISP storage spaces are too clogged, and garbage collections begin to take up too great a portion of execution time. When this happens, the system complains with a GC-ERROR, and prints the message "SYSTEM TOO FULL". At this point, the frustrated user can either use PARAMFL to change his field length limit, if it isn't already at his KRONOS maximum, and try re-execution. If the user is at his KRONOS maximum, he can try freeing up space by eliminating unneeded programs and data. He must pare down his programs, use program segmentation, and try calling in segments only when they are needed; or find more efficient ways of storing his data. The address capabiity of ALISP is limited to 17 bits (see section I.2), and there are no immediate plans for increasing it.

14.3 Garbage Collection

When all unused FREE storage is gone, the ALISP system does a garbage collect to reclaim all FREE storage which is not actively accessed by any ALISP data structure. For instance, suppose the following dialogue took place:

```
 ?'(A B C D)
(A B C D)
 ?(SETQ FOO '(E F G H))
(E F G H)
 ?
```

At this point, the lists (A B C D), although still present in core, is unaccessed by any structure in the ALISP system, and hence just using up valuable FREE storage. On the other hand, the list (E F G H) is accessed by the atom FOO, and it must remain in FREE storage. A garbage collection frees up all unaccessed structures like the list (A B C D), as well as unaccessed litats (TWA's, section I.2) and numbers. This freed-up space is linked together and becomes the new unused storage for FREE space. If the unused storage after a garbage collect is too small, a re-allocation is done, and FREE space is expanded (see section 14.1).

The garbage collect routine also checks the unused space remaining in all the other ALISP spaces, and does a re-allocation on them if necessary.

The FREE space garbage collect uses a recursive algorithm that is fast and efficient; typical GC times are on the order of a few tenths of a second. The dynamic storage capabilities of ALISP assure that a jammed system with frequent GC's will not occur as long as there is space left before field length limit is reached (see section 14.2). If this limit is reached with a crowded system, and the GC routine cannot free up a reasonable amount of space, a GC-ERROR is issued. The amount of unused storage remaining in all four ALISP storage areas, as well as the number of GC's performed, can be obtained from the PARAMGC function.

If you wish for some odd reason to have a garbage collect performed at a specific time, the function GC, a SUBR of no arguments, is available. Evaluating (GC) will cause an immediate garbage collection; the result is NIL.

Interrupts from the terminal are recognized during a GC, but not performed until the GC is exited. Patience as always.

I Chapter 15

FILES

The ALISP system has the ability to access and maintain KRONOS permanent files. With this ability, the user can fetch and store large masses of S-expressions or character data on the KRONOS mass-storage device. For the user interested in maintaining large ALISP programs, a filing system has been written using the filing primitives described in this section; refer to section II.1. The filing system described there is adequate for most user needs, and is in a very convenient form. This present section describes the workings of the ALISP file primitives, and is useful for the user who wishes to do his own file handling.

15.1 Permanent and Local Files

The KRONOS operating system allows the user to store permanent files in his catalog. These files can be accessed from ALISP for reading and writing. Normally, a copy of a permanent file is attached to ALISP and made available for file operations as a local file. After file operations are performed, the local file can be detached from ALISP and optionally put back in the permanent catalog, either replacing the old permanent file, or creating a new permanent file alongside the old.

15.1.1 Opening a Permanent File

A permanent file can be opened for use by ALISP with the function OPEN. Format is:

```
(OPEN fname unit)
```

where fname, a literal atom, is a permanent file name, and unit is a logical unit number from 1 to 16 (see below, 15.1.2). Both arguments are evaluated. OPEN searches the user's catalog for a permanent file with the name fname, and attaches it as local file unit. If fname is NIL, then an empty file is attached. This is useful for creating new files from ALISP.

15.1.2 Local Files

Local files are accessible to the ALISP reading and printing functions. A local file in ALISP has a unit number from 0 to 16. Unit number 0 is reserved for terminal I/O, while 1 through 16 are used for communication with disk files.

All currently open local file units can be found with the function UNITNOS, a SUBR of no arguments. The result is an ordered list of SNUM's of all currently open local file units. An opened local file is also called an active local file. There can be a maximum of 16 active local files at any time.

Often it is desirable to find a unit number that is not currently in use, so that OPEN will not release a currently active local file. The function FRUNNO, a SUBR of no arguments, will return an inactive unit number if one exists, or NIL if there is none. All ALISP support packages (INPUT, EDITFILE, etc.) use FRUNNO so that they will not destroy a user's currently active local files.

Local file units are assigned KRONOS local file names which the user need not ordinarily worry about, except if he has KRONOS local files he doesn't want destroyed across an ALISP run. The reserved file names are TAPE01 through TAPE20.

For those users worried about storage requirements, each active local file uses about 200 words of binary program space for a buffer. This space is released when the local file is closed. Keeping many local files active can clog up a loaded ALISP system; it is wise to close local files as soon as possible.

The status of a local file unit can be accessed with the function FILESTAT, a SUBR of one argument. Format is:

(FILESTAT unit)

where unit is a local file unit from 1-16. If the unit is not currently open, FILESTAT returns NIL. If it is open, FILESTAT returns a list of four elements giving the status of the unit:

(access eofstat perm lastop)

where:

access is IA for indirect access,
DA for direct access

eofstat is NIL if the unit pointer is not at the end of the unit

1, 2 or 3 if it is (see EOFSTAT, section 15.3.1).

perm is R if read only
W if read and write

lastop is NIL if no file operations have taken place, or a REWIND was just performed.

R if the last operation was a read
W if the last operation was a write

15.1.3 Closing a Local File

An active local file can be detached from ALISP with the function CLOSE. Format is:

(CLOSE fname unit)

where fname is a (literal atom) permanent file name, and unit is an active local file unit. The local file will be detached from ALISP and saved as the permanent file fname, replacing any other permanent file of that name. Note that a local file need not be replaced as the same permanent file from which it was opened.

If fname is NIL, the local file will be detached without being replaced in the permanent catalog. Also, there will be no error if unit is not an active local file. Thus a user can always detach a local unit without worrying whether it is active or not.

CLOSE always returns fname as its result.

15.1.4 Alternate Catalogs and Passwords

Permanent files from catalogs other than the user's can be accessed using a slightly different form of OPEN and CLOSE. Instead of a single literal atom file name, a list specifying file name, user number, and (optionally) a password can be used:

(fname usernum password)

usernum should be a valid KRONOS user number, and password should be the file's password, if it exists. If the permanent file is not in the alternate catalog, or if the file is not public or semi-private, then OPEN will issue an error. A permanent file can be made public or semi-private with the KRONOS CHANGE command.

Alternate catalogs cannot be used with the CLOSE function.

Certain values for usernum cause special actions for OPEN and CLOSE.

1. LOCAL

Opens or closes KRONOS local file. For OPEN, instead of checking the user's permanent catalog, a KRONOS local file is attached to ALISP; the original local file is destroyed. For CLOSE, the ALISP local file is detached and left as a KRONOS local file.

LOCAL is useful where ALISP must pass large files to other programs.

2. DISPOSE

Disposes a local file to the printer. An active file can be printed on the batch printer by using CLOSE with DISPOSE as the usernum. At the end of an ALISP run, all such closed files are dumped to the printer. Remember that the first character of each line is used for printer carriage control.

15.1.5 Direct Access Files

Normally, a user will deal only with indirect access KRONOS files. When an OPEN operation is performed on an indirect access permanent file, a copy of that file is attached to ALISP as a local file. Changes can be made to the local file without affecting the permanent file; when all changes are completed, the permanent file can be replaced by the local file with the function CLOSE.

Direct access files, by contrast, are attached directly to ALISP. Any changes made to the local file are directly reflected in the permanent file. The advantage in using a direct access file is that the overhead involved in making a copy of the file is eliminated; this overhead can be significant for large files. The disadvantage is that write operations on the local file are directly reflected in the permanent file, and may leave the file in an undesirable state if a program error occurs before all processing on the file is completed.

Typically, large files that are available on a read-only basis to many users are made direct-access. Thus the main ALISP SAVE file, which is read automatically when ALISP is started, is direct-access.

OPEN will find both direct and indirect access files; it checks for indirect access first. If it is known that a file is one or the other, OPEN can be made to look for only that type. An optional argument is included:

```
(OPEN fname unit access)
```

where access is evaluated, and should be either IA (indirect access) or DA (direct access).

A local file opened from a direct access permanent file can be closed using:

    (CLOSE NIL unit)

since all changes to the local file are also made to the permanent file.

A new direct access permanent file can be created from a local file by using an optional argument:

    (CLOSE fname unit 'DA)

which will create the permanent file fname from local file unit as a direct access file. The default for CLOSE without the optional argument is to create an indirect access file.

15.1.6 Permission Modes

A local file which is attached by OPEN is normally available for both reading and writing. For direct access files this may be a problem, since changes to the local file are reflected immediately in the permanent file. A user may thus wish to attach a file in a read-only mode, in order to prevent accidental damage to the permanent file. A local file can be made available for read-only operations by an extra argument to OPEN:

    (OPEN fname unit 'R)

All attempted write operations on the local file will cause a FIL-ERR.

15.2 Sequential File Operations

The normal mode for performing file I/O is by sequential operations. This section describes sequential file formats and operations.

15.2.1 Sequential File Format

Sequential files are composed of lines. Like terminal I/O lines, a sequential file line can be up to 150 characters long. No line editing is done on file lines, however; thus a control-H in a file line will be read as a control-H, rather than causing the previous character to be deleted.

The character set used by sequential files is the KRONOS 6-bit or 6/12-bit set, so that sequential files can be created by the KRONOS TEXT command. Also, any sequential files created or modified by ALISP are readable by other programs as text files.

The choice of 6-bit or 6/12-bit character sets is controlled by the switch ASCII. Initially ASCII is set to T, so that sequential files are read using the extended 6/12-bit convention, corresponding to ASCII terminal mode. If a sequential file is to be read or written using 6-bit conventions (NORMAL terminal mode), then ASCII should be set to NIL.

15.2.2 Sequential File Pointer

A local file unit has an associated unit pointer that tells what the current position of the unit is. When a request for an input line is made to a unit, the line is taken from the position of the unit pointer, and the unit pointer is advanced to the next line. There is no way to skip forward or backward within the file, since the unit pointer is not directly accessible to the user (sequential files may be rewound, however).

Each local file unit has its own pointer, so that input and output to different units can be intermixed without losing track of the position of any given unit.

When a line is written to a unit, it is written at the position of the unit pointer, and the unit pointer is incremented past the line just written. Successive lines will thus be written one after the other on the unit. A side effect of writing a line to a unit is to cause an EOI (end-of-information) to be placed at the end of the written line. Thus any lines after the unit pointer are automatically lost when a sequential write is performed. The last line written will always be the last line of the file.

15.2.3 Reading Sequential Files

On input, whenever the input buffer must be filled to satisfy a read request (from READ, TEREAD, etc.) the value of INUNIT is checked. If it is not an SNUM in the range 0-16, a NUM-ERR is issued and INUNIT is reset to SYSIN. If INUNIT is in the correct range, then the input buffer is filled from the corresponding local unit, if it is opened. For INUNIT = 0, input is taken from the terminal. An example of reading from a local file unit is given in Dialogue 15.1 below. The function EOFSTAT checks for the end of the unit (section 15.2.5 below).

The SYSIN unit is special, since INUNIT is set to this unit whenever an error occurs, or the top-level loop of the ALISP supervisor is entered. The supervisor can thus be made to read and evaluate S-expressions from a file unit other than the terminal. For example, the user can type S-expressions he wishes to have evaluated into a text file outside of ALISP, then enter ALISP, open the text file, and evaluate the S-expressions by setting SYSIN to the local file unit. (See Dialogue 15.2.) The control card parameter SI can be used to open and read a file of

Dialogue 15.1
Reading a Local File

Let unit 1 have the following three lines in it:

```
(FOO BAR)
(MABEL HATES FIGS)
MOMANDDAD
```

```
?(PROG (INUNIT)
?     (SETQ INUNIT 1)                The PROG has set the litat
                                     INUNIT to local file unit 1.

? TAG (IF (EOFSTAT 1)(RETURN 'DONE))
                                     If the end of unit 1 is
                                     reached, return.

?     (PRINT (READENT))
?     (GO TAG)))))                   If there is stuff in the file,
                                     read and print it.

(FOO BAR)
(MABEL HATES FIGS)
MOMANDDAD
DONE                                 The three S-expressions are
                                     read from the file, and the
                                     PROG exits with value DONE.

?(EOFSTAT 1)
T
?(REWIND 1)
1
?(EOFSTAT 1)
NIL                                  Local unit 1 is at its EOI. A
                                     call to REWIND resets it to
                                     the beginning again.
```

S-expressions automatically when ALISP is entered (Appendix B).

The ECHO switch can be used to automatically echo lines read from a local file unit onto the current output unit. ECHO should be set to the local file unit being read from. Echoed lines are printed exactly as they appear on the input unit. The ALISP output buffer is not used, so its contents are undisturbed. The initial value for ECHO is NIL, i.e., no echoing takes place.

Echoing lines from one local file unit to another is much faster and less wasteful of storage than READing and PRINTing S-expressions. It is thus a useful way of copying portions of one file to another.

Dialogue 15.2
SYSIN Set to a Local File

Let the following lines exist on local file unit 1:

```
            (CONS 'FOO 'BAR)
            DE FACT (X) (COND ((XEROP X) 1) (T (TIMES X (FACT (SUB1
X))))))))
            (FACT 4))
            (PPRINT 'FACT)
            (LIST 'FOO 'BAR)
            (SETQ SYSIN 0)
```

```
 ?(SETQ SYSIN 1)
1
```

This sets the top-level input file to unit 1.

```
(FOO,BAR)
FACT
24
(LAMBDA (X)
   (COND
   ((ZEROP X) 1)
       (T (TIMES X (FACT (SUB1 X))))))
(FOO BAR)
0
 ?
```

With SYSIN set to 1, the S-expressions in local input file unit 1 are evaluated by the EVAL supervisor. Since the final expression in the file was (SETQ SYSIN 0), supervisor continues to take input from the terminal. If no such statement had been included in the local file, the ALISP system would have exited after encountering the EOI on the SYSIN unit.

15.2.4 Writing Sequential Files

On output, whenever the output buffer must be dumped to satisfy a write request (from PRINT, TERPRI, etc.) the value of OUTUNIT is checked. If it is not an SNUM in the range 0-16, a NUM-ERR is issued and OUTUNIT is reset to SYSOUT. If OUTUNIT is in the correct range, then the output buffer is dumped to the corresponding local unit, if it is opened. For OUTUNIT=0, output is dumped to the terminal.

The SYSOUT unit is the default unit used by the top level supervisor, and by the error processor. It is thus possible to channel output from the interpreter to a local unit, by opening a local unit and setting SYSOUT to it. This can be useful when debugging, if large amounts of output are produced.

The switches SLASHES, ASCII, NORMTAB, and PRINBEG are often useful in writing to files.

15.3 End-of-File Processing

Normally, files processed by ALISP are single-record files from the standpoint of KRONOS. They consist of a single KRONOS record, which may contain an arbitrary number of lines. Files are ended by an EOI (end-of-information) mark. ALISP will not read past this mark; if a READ is incomplete when the EOI mark is encountered, a FIL-ERR is issued. Lines can always be appended to the end of a file; the EOI mark is placed after the last line written.

KRONOS makes a further distinction in end-of-file marks, with EOR (end-of-record), EOF (end-of-file), and EOI (end-of-information). Any one of this will normally be interpreted by ALISP as the end of a file.

15.3.1 EOFSTAT and REWIND

The function EOFSTAT, a SUBR of one argument, will enable the user to tell if a unit pointer is at the end of the unit. Format is:

```
(EOFSTAT unit)
```

where unit is a local file unit number from 1 to 16 (the terminal, unit 0, is never at an end-of-file, since a new line can always be typed). EOFSTAT returns NIL if the unit pointer is not at the end of the unit, and 1, 2, or 3 if it is. The numbers correspond to KRONOS end-of-file marks:

```
1 = EOR
2 = EOF
3 = EOI
```

It is always possible to reset a local file unit pointer to the beginning of the unit, with the function REWIND, a SUBR of one argument. (REWIND unit) will rewind local file unit unit, where 1 ≤ unit ≤ 16.

15.3.2 Multi-record Files

In special cases, an ALISP programmer may need to access or write multi-record or multi-file files. However, the read and print functions in ALISP always work with a single KRONOS file record. In order to skip past a record, two special functions are provided: EOFSKIP and EOFMARK.

EOFSKIP, a SUBR of two arguments, is used when reading a local file unit. It will skip the unit pointer over end-of-file marks, to get to records other than the first on multi-record files. Format is:

(EOFSKIP unit eoftype)

where unit is a local unit number, and eoftype is an SNUM which specifies the type of end-of-file mark to skip, according to the following table:

eoftype	action
0	skip up to the next EOR (but do not cross it to the next record).
1	skip past the next EOR
2	skip past the next EOF
3	skip to the EOI.
-1	rewind to the beginning of the current record.
-2	rewind to the beginning of the current KRONOS file (just after the last EOF read).
-3	rewind to the beginning of the file (equivalent to REWIND).

EOFSKIP will normally return its second argument as a result. There is one special case: if eoftype is 2 and there is no EOF before the EOI, NIL is returned, and the unit pointer is positioned at the EOI.

ECHO will work with EOFSKIP for positive eof type. All lines and end-of-file marks skipped over are copied to the echo output unit.

In order to produce multi-record files from ALISP, the ability to write end-of-file marks must be available. The function eofmark, a SUBR of two arguments, will do this. Format is:

(EOFMARK unit eoftype)

where unit is a local file unit number, and eoftype is either 1 or 2. A 1 causes an EOR to be written, and a 2 an EOF. In both cases the unit pointer is advanced past the end-of-file mark just written, and points at the EOI. EOFMARK returns its second argument.

There are some peculiarities to note, for you fans of the

KRONOS file system. First, an EOR is automatically written before any EOF, so that the structure of a multi-file file is always hierarchical:

```
          first     data
          record    EOR
          second    data       first multi-file file
          record    EOR
                    data
                    EOR
                    data
                    .
                    .
                    .
                    data
                    EOR
                    EOF
                    etc.
```

The only exception to this rule is at the end of the file, where an EOR may be followed directly by an EOI.

Second, it is basically impossible to write an empty record (although an empty file is possible). Try to put something in each record in a multi-record file, or an EOFMARK with **eoftype** of 1 will write an EOF rather than EOR (don't ask why).

15.4 Checkpoint Files

A checkpoint file is a snapshot of the ALISP system. A checkpoint file can be re-loaded to restart ALISP at the exact point at which the checkpoint file was made. Typically, checkpoints are used to save the state of an ALISP execution after a large amount of setup has been done. The checkpoint file saves the cost of the execution of the setup each time ALISP is entered.

Because checkpoint files are expensive in terms of disk storage, they should be used with restraint. Production systems which require substantial setup time and are used by a number of users are the best candidates for checkpointing.

The functions LOAD and SAVE, both SUBR's of one argument, are used for creating and loading ALISP checkpoints. Their format is:

```
     LOAD
(or filename)
     SAVE
```

where **filenam** has the same format as the **filenam** parameter in the OPEN command (section 15.2.2 above).

SAVE creates a snapshot binary file of the ALISP system, and saves it as a permanent indirect-access file with name fn. SAVE uses CLOSE to store the permanent file; see the description of the CLOSE command in 15.2.2 above for alternate user access, permission mode, etc.

SAVE can be called at any point in an ALISP program. It is a normal function call, and does not interrupt the program flow. SAVE returns fn as its result.

LOAD loads the checkpoint file fn into the ALISP system, and starts up execution at the point where the SAVE was called. Since LOAD uses the OPEN function to find the permanent load file, the same conditions of alternate user access and permission apply as for the OPEN function; see section 15.2.2 above.

LOAD returns NIL as its result, so that a a program with an embedded SAVE can determine whether the SAVE just created the checkpoint, or the checkpoint was started up by a LOAD.

A simple example of the use of SAVE and LOAD commands is given below in Dialogue 15.3.

There are several parameters and buffers which are not saved in a checkpoint, or restored on a LOAD. Local file units remain unchanged under a SAVE or LOAD. The input and output buffers also remain unchanged, although the buffer pointers (PRINBEG, READBEG, etc.) take on values from the loaded file. The parameters involving CP time, number of GC's, and maximum field length, as well as the control-point parameters, are all unchanged by a LOAD.

Checkpoints are a convenient method for saving an entire ALISP system for later restart. They are inexpensive in terms of CP time, but take a lot of disk space to save. Nevertheless, there are times when it is worth the added disk expense to save a checkpoint: when it is to be used often, or as a safety checkpoint before trying a tricky and bombable ALISP program. For storage of large programs, it is recommended that you use the ALISP filing system (see section II.1) rather than checkpoints.

A checkpoint receives special status when it is called from the ALISP control card with the LD parameter. In this case, none of the other control card parameters are processed; instead, the checkpoint can use the PARAMCP function (see section I.1.1.1) to fetch the other parameters, and perform its own control card processing.

Because they are so wasteful of disk space, checkpoints, when used in excess, have a tendency to overflow a user's catalog

Dialogue 15.3
ALISP Checkpoint Functions

```
?(PROGN (PRINT 'FOO)
?       (PRINT (SAVE 'TEMP))
?       (PRINT 'DONE)))
1
TEMP
DONE
DONE
```

The PROGN function executes a number of functions. The first PRINT call prints the atom FOO. The second print call evaluates the SAVE function, which saves a copy of the ALISP system as the overlay file TEMP, and returns the atom TEMP as its value; PRINT outputs this atom. The final call to PRINT outputs the atom DONE, and PROGN returns the same atom for its result.

```
.
.
.
```

This signifies the user does some processing.

```
?(LOAD 'TEMP)
TEMP
DONE
DONE
```

The LOAD function loads the overlay TEMP and starts executing at the point where the SAVE call was issued. The second two PRINT statements in the PROGN call are executed. In all respects the ALISP system is now at the same point as it was after the PROGN expression was first executed. The processing done after the PROGN call has dissappeared.

```
?(CONS (LOAD 'TEMP) NIL)
TEMP
DONE
DONE
?
```

The LOAD function completely

halts the program being executed when it is called. Here, the call to CONS never completes; instead, the ALISP system restarts in the PROGN function again.

limits. Indeed, the lowest priority user number class (BL) does not have enough permanent file storage space to save an ALISP checkpoint. When there is not enough room to store a checkpoint, a FIL-ERR is issued. The user must either delete unwanted permanent files with PURGE (see above) or request a user number with a larger permanent file storage limit.

I Chapter 16

Batch

ALISP will operate under batch at the University Computer Center. This section describes the method for calling ALISP from batch, and some peculiarities of a batch origin ALISP job.

16.1 Running a Batch Job

The control cards needed to run ALISP are:

```
JOBNAME.
ACCOUNT,...
ATTACH,ALISP/UN=LISP000.
ALISP.
7/8/9 -EOR card
data for the ALISP interpreter
(EXIT)
6/7/8/9 -EOI card
```

The ALISP control card can have parameters attached to it; see Appendix B for the effects of these parameters. There can be other KRONOS control cards before and after the ALISP control card. There is no error exit from the ALISP control card; upon completion of the ALISP program, the next KRONOS control card is always executed, unless the time limit has been reached. The time limit for the whole job can be set using the T parameter on the job card or with the TL parameter on the ALISP control card. Within ALISP, the time limit can be extended to the user's validated maximum with the PARAMTL function (section I.14.2).

The 7/8/9 card (multi-punched in the first column) is an end-of-record, and signals the end of the KRONOS control cards. After it comes the data used by the control cards. ALISP always uses one record of data from the batch deck, even if the user changes SYSIN so that no data is actually read into the ALISP system from the deck (see below, 15.2). Which record in the deck gets used by ALISP depends on where the ALISP control card appears. If the ALISP control card is the first control card to use records from the input deck, then it uses the second record in the deck (the first record after the control cards). If control cards before the ALISP card use records from the deck (FORTRAN, COMPASS, etc.), then ALISP will use the one after theirs. An example of a multiple deck structure might be:

```
JOBNAME,T100,CM50000.
ACCOUNT
FORTRAN.
ATTACH,ALISP/UN=LISP000.
ALISP.
7/8/9
data for FORTRAN compiler
7/8/9
data for ALISP interpreter
(EXIT)
6/7/8/9
```

An (EXIT) statement is normally included as the last statement in an ALISP data record, but it is not strictly necessary. If ALISP hits the end-of-record mark on the data from batch, it automatically terminates as if an EXIT had been called. Also, an EXIT can be called from anywhere within a statement being executed by the ALISP interpreter; it will cause an immediate exit from ALISP.

16.2 File Assignments and Initial Values

Local file unit 0 is the job deck data record on input, the file OUTPUT (i.e., the batch printer) on output. Initially, SYSIN and SYSOUT are both set to 0, unless the I or O options are used on the ALISP control card.

The data record evaluated by ALISP can be printed on the output device along with the results of the evaluations. This is done automatically on entering a batch ALISP job, which sets ECHO to SYSIN; all lines read from the SYSIN unit will be printed directly on SYSOUT before they are evaluated. If SYSIN is set to 0, then the echo is preceeded by prompt characters to distinguish it from the results of its evaluation. The prompt characters are always "$$$$$$$$Ø"; the atom PROMPT has no effect under batch. The echo feature can be turned off by using the E parameter on the ALISP control card.

The atom PRINBEG is initially set to 1, rather than 0 as under timesharing. Since the first character of each line outputted to the printer is interpreted as a carriage-control character (see Appendix A), setting PRINBEG to 1 causes this first character to always be a blank, i.e., skip to the next line. If you reset PRINBEG to 0 under batch, then all printer output will lose the first character to carriage control. On the other hand, it is sometimes desirable to do your own carriage control (skip to the top of the page, etc.), and this can be done by setting PRINBEG to 0 for the carriage control line, resetting it to 1 for printing of S-expressions. The printer carriage-control characters are listed in Appendix A. Be careful to reset PRINBEG to 0 when outputting to local file units other than 0, or they might not read back in properly.

16.3 BATCH, Interrupts, and Overlays

Although batch and timesharing programs run very much alike under ALISP, it is sometimes necessary for a program to know if it is running as a batch job or not. The function BATCH, a SUBR of no arguments, can be used. It returns T if the program is running as a batch job, NIL if it has a time-sharing origin.

Interrupts are of course inactive under batch. The function BREAK and the switch INTRFLG are still available, but have little use in the batch environment, and should not be called.

Checkpoints are a slight problem in batch mode, since they can be created from either batch or time-sharing jobs. If a checkpoint created by a time-sharing ALISP job is loaded into ALISP running under batch, all the batch peculiarities described so far in this section will apply to the checkpoint system. Remember, however, that the checkpoint loads its own values of PRINBEG and ECHO, and the batch job may have to reset these to continue comfortably. The function BATCH described above comes in handy here. The following expression is an example of an overlay creation which will load differently (and correctly) for batch and timesharing jobs:

```
(PROGN (SAVE 'MYLOAD)
       (COND ((BATCH) (SETQ PRINBEG 1 ECHO 0))
             (T (SETQ PRINBEG 0 ECHO NIL))))
```

II Chapter 1

ALISP Filing System

The filing system, written in the ALISP language using the file primitives, provides an effective and painless means for creating, documenting, maintaining, modifying, inputting and outputting large numbers of ALISP S-expressions. It is recommended that both the novice and experienced LISPer use this system for maintaining large ALISP programs consisting of many functions and plist assignments.

1.1 General Description

The filing system uses indirect-access permanent files to store groups of S-expressions. The only limit to the number of separate files that can be maintained is the user's maximum file limit. Each file is given a user-specified name by which it is called from ALISP; this name is also the permanent file name in the user's catalog. Each file can have an unlimited number of S-expressions in it (as long as the user's KRONOS file limits aren't exceeded, of course). In addition, each file can contain documentation for every S-expression in it, as well as information regarding formatting of the file for printing, compiler declarations and other subsidiary niceties.

The filing system operates completely from within ALISP. Do not attempt to create files outside of ALISP to be used by the ALISP filing system, as they will not work. Within ALISP, however, you can, for example, define a function, output it to a file, modify it, update the file, re-input the function, etc. A typical session might look like the one in Dialogue 1.1 below.

In this session, the user defined the function FACT, and initialized an ALISP file called MYFNS using the function INITFILE. Then FACT was output to the file MYFNS, and the LISTFILE function revealed that MYFNS did actually contain FACT. The user then did some editing, and re-output FACT; the new version replaced the old in MYFNS. Then he set FACT to NIL, erasing the function definition; the function INPUT retrieved FACT from the file MYFNS, where it was safely stored. The user then verified that FACT contained its old function definition.

This sample session did not exhaust by any means the abilities of the filing system, but it showed by far the most

Dialogue 1.1
The ALISP Filing System

```
 ?(DE FACT (X) (COND ((ZEROP X)1)
 ?(T(TIMES X(FACT (SUB1 X)))))))
FACT
```

The user defines the recursive factorial function.

```
 ?(INITFILE MYFNS)
MYFNS
```

A new ALISP permanent file is created with the filing system function INITFILE. This file is initially empty.

```
 ?(OUTPUT MYFNS (FACT))
(FACT)
 ?(LISTFILE MYFNS)
(FACT)
```

The user now outputs the function definition to the file MYFNS by using the filing function OUTPUT. The function LISTFILE verifies that the file actually contains the function definition that was output.

```
 ?(EDIT FACT)
  .
  .
  .
END EDIT
 ?(OUTPUT MYFNS (FACT))
```

The user now changes the function definition of FACT with the editor. The new version of FACT is re-stored in MYFNS with the OUTPUT function. The new definition replaces the old one.

```
 ?(SETQ FACT NIL)
NIL
 ?(INPUT MYFNS (FACT))
(FACT)
 ?(FNTYPE FACT)
LAMBDA
```

The user forgets he has defined FACT and sets its value to NIL, erasing the function definition. All is not lost, however; he simply inputs FACT from the file MYFNS with the filing system function INPUT. Note that now FACT has been restored to its lambda-expression value.

---

practical and typical use: defining a function in ALISP and saving it in a permanent file for later use.

By convention, files created using INITFILE for the storage of S-expressions are called ALISP files. Throughout this chapter, "file" will mean an ALISP file so created and used. Although ALISP files are KRONOS text files, they have a very restricted format. Hence it is usually not profitable to list them on the terminal, or use the KRONOS text editor on them (ALISP files may be listed from ALISP with the function GRIND, and edited with EDITFILE or OUTPUT).

1.2 File Format

ALISP files are intended to be convenient depositories for information created during an ALISP run. Since most of the useful information is stored on the value or plist of atoms (e.g., function definitions), this is what ALISP files contain: a set of atoms, along with their values and property lists. Each atom and its associated information is called an *entry* in the file.

Functions exist for transferring entries from ALISP to a file, from a file to ALISP, or from one file to another. Since a file (or ALISP) can contain a great number of atoms, and usually only a subset is to be transferred, an *entrylist* is commonly used in the filing functions. In its simplest form, the entrylist is simply a list of literal atoms:

```
(FOO BAR .....)
```

For example, in Dialogue 1.1, the entrylist (FACT) was used to specify the single atom FACT on input and output.

On rare occasions a more complicated entrylist allows the user to specify only the value or plist of an atom:

```
(FOO (BAR VALUE)(FACT PLIST) ....)
```

Here the value of BAR and plist of FACT are indicated (note that the default is to consider both as for FOO).

Some functions interpret an empty or atomic entrylist as meaning "all entries", e.g., INPUT will input the whole file. Descriptions of individual functions will indicate actions for special values of entrylist.

1.3 Filing Functions

The filing functions are organized in this manner:

Initialization:	INITFILE
Input, output, upkeep:	INPUT OUTPUT OUTPUTA COPYFILE PURGFILE
Printing and listing:	LISTFILE GRIND
Documentation and formatting:	COMMENT PAGEFILE DECFILE

Since all the file functions are defined as FLAMBDA's, all arguments are unevaluated. Use (INPUT MYFNS (FOO BAR)) rather than (INPUT 'MYFNS '(FOO BAR)).

1.3.1 Initialization

Before ALISP atoms can be output to a file, the file must have been created with INITFILE. Once created, an ALISP file exists as a permanent file until it is destroyed (usually by a KRONOS PURGE command).

Initialization creates an empty ALISP file as an indirect-access permanent file, destroying any permanent file with the same name in the user's catalog (or alternate catalog, if one is specified in *filenam*). The name used by the ALISP filing functions and the name in the user's catalog are the same.

The function call format for initialization is:

```
(INITFILE filenam)
```

If *filenam* is atomic, then INITFILE initializes the file with name *filenam* in the user's catalog; this file need not have already been present as a permanent file. If *filenam* is not atomic, then INITFILE will attempt to initialize an ALISP file in an alternate user's catalog. The format and restrictions on *filenam* in this case are the same as those for the CLOSE function (section I.15).

1.3.2 Input, Output and Updating

These are the most-used filing functions. They affect only the value and plist attributes of entries. With them, the user can output S-expressions to an ALISP file, input S-expressions from a file to the ALISP system, purge file entries, and transfer entries from one ALISP file to another.

The standard format for these functions is:

(filefn filenam entrylist)

where entrylist is as defined in section 1.2 above, and filenam is a file name. If an alternate user number is specified in filenam, then it must conform to OPEN specifications for INPUT, and CLOSE specifications for the other functions (see section I.15).

INPUT

This function inputs entries from an ALISP file into the ALISP system. If entrylist is not atomic, then only those entries or parts of entries specified by entrylist are input.

If entrylist is atomic or omitted entirely (only one argument to INPUT), then all entries will be input from the file.

INPUT returns a list of those entries for which it input either a plist or a value.

OUTPUT

This function outputs entry attributes to an ALISP file from the ALISP system. Only those entries specified by entrylist are output; an atomic entrylist does nothing. The entry name must have either a value (not be ILLEGAL) or a non-NIL plist in the ALISP system in order for these attributes to be output.

If an entry in entrylist is in the output file, the new attributes replace the old ones. If it is not, a new entry is created at the end of the file and the attributes placed there. OUTPUT thus does not change the order of already-present entries in filename. OUTPUT returns a list of those entries into which it output either a plist or value attribute.

The two functions INPUT and OUTPUT are the most important and useful members of the filing system. In general, they are very friendly -- they protect the user from his mistakes, and need very little thought to be used correctly. Some examples of these two functions are given in Dialogue 1.2 below.

OUTPUTA

Dialogue 1.2
The INPUT and OUTPUT Functions

```
?(DE FOO (X) X)
FOO
?(PLIST 'BAR '(MOO MAR))
(MOO MAR)
```

The user defines FOO as a function, and puts something on the plist of BAR.

```
?(INITFILE MYFILE)
MYFILE
?(OUTPUT MYFILE (FOO BAR))
(FOO BAR)
```

The user initializes an ALISP file with the name MYFILE, then outputs the attributes of FOO and BAR to this file. Note that the file now contains the value (function definition) of FOO and the plist of BAR, since these were the only attributes defined for these two atoms in ALISP.

```
?(DE BAR (X) (CONS X NIL))
BAR
```

BAR now has a function definition as well as a non-empty plist.

```
?(INPUT MYFILE (BAR))
(BAR)
?BAR
(LAMBDA (X) (CONS X NIL))
?(INPUT MYFILE ((BAR VALUE)))
NIL
```

The user now inputs the entry BAR from the file MYFILE, expecting to replace the value of BAR with its value on the file. However, no value attribute for BAR was output to MYFILE: INPUT finds the plist attribute and inputs that, but the value of BAR remains the same. Note that when the user tries to input the value attribute of BAR from MYFILE, INPUT returns NIL as a result and does nothing.

```
?(REMOB 'FOO)
FOO
```

```
?(OUTPUT MYFILE (FOO))
NIL
```

The user removes the value of FOO with the function REMOB. Since FOO now has neither a value nor a non-empty plist, OUTPUT cannot send anything to MYFILE, and returns a NIL result. The entry FOO of MYFILE remains unaffected.

```
?(INPUT MYFILE (FOO))
(FOO)
?FOO
(LAMBDA (X) X)
?(PLIST 'FOO)
NIL
```

INPUT restores the value of FOO that was originally sent to the file MYFILE. Note that the plist of FOO remains unaffected, because it was not output to MYFILE.

This function is essentially the same as the OUTPUT function, except that it affects the order of entries in a file.

entrylist should be non-atomic. The first entry in _entrylist_ is the entry after which all of the other entries in _entrylist_ will be output; this first entry is not itself output to the file. If it does not exist as an entry in the file, then everything is added at the end of the file.

The rest of the entries on _entrylist_ are output as in the OUTPUT function. If they have either a value or non-NIL plist, then they are output after the first entry on _entrylist_, and any duplicate entries are deleted from the file. If they do not have a value or non-NIL plist, then they are not output; and the entry on the file is not affected.

PURGFILE

This function purges attributes and entries from an ALISP file. If _entrylist_ is atomic, no action is taken; use INITFILE to completely erase all entries in an ALISP file.

If, thru purging, both the value and plist attributes of an entry are deleted, then the whole entry is deleted. An entry must have either a value or plist attribute to remain in the file. PURGFILE returns a list of all entries from which it has deleted at least one attribute.

COPYFILE

This function copies entries and attributes from one ALISP file to another. It has a third argument:

(COPYFILE filenam1 entrylist filenam2)

Entries are copied from filenam1 to filenam2; only complete entries can be copied, so specifying PLIST or VALUE attributes within entrylist will have no effect. An entry in entrylist, if it appears in filenam1, is first deleted from filenam2 if it is present there, and then copied from filenam1 onto the end of filenam2. If an entry is entrylist does not appear in filenam1 no deletion or copying to filenam2 occurs. filenam1 always remains unchanged. COPYFILE returns a list of entries actually copied. If entrylist is atomic, all entries from filenam1 are copied.

1.3.3 Printing and Listing

At some point it is desirable to know what the contents of an ALISP file actually are. LISTFILE retrieves entry names from a file, while GRIND pretty-prints value and plist attributes.

LISTFILE

This function has the standard format:

(LISTFILE filenam entrylist)

filenam can either be atomic, in which case it specifies an ALISP file in the user's catalog; or non-atomic, in which case it has the same format and restrictions as the argument to OPEN in Chapter I.15.

If entrylist is atomic or omitted, all entry names in filenam will be returned. Otherwise, LISTFILE returns only entry names in entrylist which are found in filenam. If a PLIST or VALUE attribute is specified in entrylist, then the corresponding entry in filenam will be returned only if it has that attribute. The value of LISTFILE is a list of entry names found.

Some examples of the function LISTFILE will be found in Dialogue 1.3 below.

GRIND

Dialogue 1.3
The Filing Function LISTFILE

```
 ?(SETQ FOO 'A BAR 'B)
B                             FOO and BAR  are given values,
                              then  outputted as  entries to
                              the file MYFILE.
 ?(INITFILE MYFILE)
MYFILE
 ?(OUTPUT MYFILE (FOO BAR))

 ?(LISTFILE MYFILE)
(FOO)                         LISTFILE  with   no  entrylist
                              returns a list  of all entries
                              in the file.

 ?(LISTFILE MYFILE (FOO))
(FOO)                         If  entrylist  is  non-atomic,
                              LISTFILE returns those entries
                              from entrylist  which it finds
                              inthe  file.   Here  FOO  was
                              found.

 ?(LISTFILE MYFILE ((FOO PLIST)
 ?                          (BAR VALUE)))
(BAR)                         Only  the  value attribute  of
                              FOO and BAR  exists on MYFILE,
                              so  LISTFILE returns  only the
                              BAR entry.
```

This most useful function displays the entries in a file, in a pretty-print format (see section II.2).

The format for the GRIND function is:

(GRIND filenam entrylist -options-)

where filenam is the name of an ALISP file, and entrylist is an entrylist for that file; an atomic entrylist indicates all entries in the file.

GRIND will pretty-print all specified entries in the file on the terminal. Options can specify printing to a file, giving a cross-reference, and several other useful features. Options can appear in any order. Their effect is as follows:

1. SNUM

specifies printing width; should be between 50 and 136. Default is value of PRINEND for printing on the terminal, and 110 for the batch printer.

2. atom PRINTER

   specifies a format suitable for printing from the batch printer, with a blank first column for carriage control. Useful when GRIND'ing to a file for later printing on the batch printer. This option is selected automatically if ALISP is running as a batch job.

3. atom DISPOSE

   specifies a dispose to the batch printer. The output of the GRIND is saved on a special local file which is dumped to the batch printer at the end of the ALISP run (see section I.15.1.4). Automatically selects the PRINTER option.

4. atom XREF

   produces a cross-reference of function calls and variable usage at the end of the listing.

5. atom ALPHA

   the output is alphabetized by entry name. This option can be expensive unless the ALISP file is short or is in nearly alphabetic order.

6. anything else

   specifies a permanent file name to receive the output of the GRIND. This file may then be listed using KRONOS commands, but note that it is not an ALISP file, and cannot be read back in correctly by INPUT.

1.3.4 Documentation and Formatting

Documentation consists of adding comments to entries within a file. Formatting specifies certain types of control to be used when printing an ALISP file with GRIND; at present the only formatting control is paging. Both of these functions only affect the printing of files, when the GRIND function is used. They have the common format:

```
COMMENT
(or          filenam entrylist)
PAGEFILE
```

COMMENT

Comments can be associated with each entry in a file. The comment is for the entry as a whole; GRIND will first print the entry name, then any comment lines associated with it, then the value and plist attributes. COMMENT is the only way to add comments to a file.

Comments reside strictly on the file. INPUT does not input comments. Most other filing functions do what one would expect, e.g. OUTPUT does not destroy comments if it updates an entry; COPYFILE copies comments; etc. The only exception is OUTPUTA, which does destroy the comments of any entries on its entrylist. The only way to print comments is with GRIND.

This function adds comments to entries within an ALISP file. If entrylist is atomic, comments are added to all entries in filenam. Comments must be added by the user from the terminal. COMMENT will print the name of an entry to be commented on the terminal, followed by the prompt "+". The user types in as many lines as desired, at least one character per line, until he wishes to end commenting of the entry; he then types an empty line (CR immediately after the "+" prompt), which ends the comment field. COMMENT then prints the next entry to be commented, followed by the "+" prompt, etc., until all specified entries have been commented.

If an entry specified by entrylist is already commented, an interactive editing mode is entered, where individual lines of the original comment attribute can be deleted or replaced or added to.

When COMMENT encounters an entry which is already commented, it prints (on the terminal) the first line of the comment, and requests input with a colon character prompt. At this point the user has two options. If he presses CR without typing anything, the printed comment line is accepted as part of the new comment attribute, and the next line of the comment is printed. The user can keep hitting CR and accepting comment lines until he finds a line he wishes to edit, or until the comments have been completely printed out.

If the user wishes to edit a line, he can type in an editing command after the line is printed. The command format is:

x n

where x is a one-letter editing command, and n is an optional base 10 positive integer (the spaces between the command and n are also optional). The effect of this command is as follows:

A -- add lines after

If the n parameter is present, this command ignores it. The A command requests lines to be added after the command line just printed; the prompt character + is used. The user can type in as many lines as he desires; when he wishes to stop, he should type a CR without typing any characters on a line (a null line). After the lines are added, the rest of the comment is edited as usual.

B -- add lines before

If the n parameter is present, this command ignores it. Processing is the same as for the A command, except lines are added before the comment line just printed. Note that both the A and B commands accept the printed line into the new comment attribute.

D -- delete lines

If the n parameter is absent or not a positive SNUM, the comment line just printed is deleted from the new comment attribute.

If n is a positive SNUM, then n consecutive lines, starting with the one just printed, are deleted (and printed on the terminal). If n is larger than the number of lines left in the old comment attribute, then all lines starting from the printed line are deleted.

If there are any lines left in the old comment attribute after the delete command is executed, processing continues on the line after the last deleted line.

E -- end editing

If the n parameter is given, it is ignored. This command ends all further editing of the comment attribute. All old comment lines, starting from the one currently being edited, are added to the new comment.

R -- replace lines

If the n parameter is absent or not a positive SNUM, only the line just printed is replaced. Else, n lines starting from the printed lines are replaced; if n is greater than the number of lines in the old comment attribute after the printed line, all these lines are replaced.

The R command is the same as the D command followed by the B command.

S -- skip lines in comment

If the n parameter is absent or not a positive SNUM, then all lines of the old comment attribute, starting with the printed line, are added into the new comment.

If n is a positive SNUM, then n comment lines, including the one just printed, are skipped over and added to the new comment attribute.

If there are any lines left in the old comment, editing continues.

Unless an E command has been given, COMMENT always makes a final input request when editing of the old comment attribute is finished. The input request is made with the prompt character +, and can be ended with a null line. After this final input request, processing of the comment entry is finished.

PAGEFILE

This function causes page markers to be inserted in the file, so that a page eject occurs when using GRIND. A page eject skips to the top of the next page on the line printer, and skips 5 blank lines on the terminal.

The format for PAGEFILE is:

(PAGEFILE filenam entrylist)

A page marker is inserted before each entry on entrylist; if entrylist is atomic, all entries are so marked.

In order to delete page markers, it is necessary to use the function DECFILE, described in the next section.

1.4 Declarations

Besides property list, value, and comment attributes, an entry in a file can have a declarations list. This list holds auxiliary information used by various functions such as GRIND and the compiler. Currently, the chief use of the declarations list is for page markers (see previous section) and compiler declarations for free variables and function linkage.

The declarations list can be manipulated explicitly with the function DECFILE:

(DECFILE filenam entrylist)

The name of each entry in entrylist will be printed, followed by its declarations list (if empty, the declarations list will print as NIL). Next an asterisk prompt will be printed, and the user can type in a new declarations list (including NIL). The declarations list will be replaced on the file. DECFILE then processes the next entry in the same way, until all entries in entrylist have been found. An atomic entrylist causes all entries to be processed.

There are two special atoms which can be typed instead of a new declarations list.

1. EDIT

   The ALISP editor is called on the declarations list. Editor command can be used to alter the list. When the editor is exited, the altered list becomes the declarations list on the file.

2. STOP

   Causes DECFILE to stop processing entries and exit. Useful if an atomic entrylist was used, and the file is large.

In order to use DECFILE, one must know what the format of declarations list elements is. Compiler declarations are described in the chapter on the compiler (II.4). The page marker is the atom PAGE appearing anywhere in the list. Page markers can be deleted by deleting this atom.

II Chapter 2

ALISP PRETTY-PRINT

One of the greatest boons to the LISP user is a good pretty-print program. Pretty-printing means that an S-expression, instead of being printed as a linear, parenthesized structure (as one would input it), has line-feeds and spaces inserted so that it prints as a block structure.

2.1 Description of the Pretty-Print Algorithm

The difference in readability from linear to block structure can be seen in the following example, utilizing the familiar recursive factorial function:

```
(LAMBDA (X) (COND ((ZEROP X) 1) (T (TIMES X (FACT (SUB1
X))))))

(LAMBDA (X)
    (COND ((ZEROP X) 1)
          (T (TIMES X (FACT (SUB1 X))))))
```

The first expression above is typical print-out from PRINT; the second is the pretty-print form of the same expression. Note that it, too, is a valid S-expression, and could be read back in correctly, since only CR's and spaces have been inserted to reformat it.

The task of programming an efficient and readable pretty-printer is not trivial. The algorithm used for the ALISP pretty-printer does a modified, truncated look-ahead down the list structure being pretty-printed at each print decision point. The time taken for pretty-print is less than a quadratic function of the size of the list, but probably greater than linear. A large list which pretty-prints in about 30 lines takes around .5 seconds of CP time.

Pretty-printing is most useful for function definitions (lambda-expressions), but it can be used with any type of S-expression. The basic action of the printer can be explained in terms of three formats: linear, open, and miser.

Linear format is used when a list can fit completely on a line. This is the same format used by the PRINT function:

```
(T (TIMES X (FACT (SUB1 X))))
```

Open format is used when there is not sufficient room left on a line to print the complete list, but the list is not overly long:

```
(COND ((ZEROP X)1)
      (T (TIMES X (FACT (SUB1 X)))))
```

Open format prints the first element of the list followed by the second element on the same line, and succeeding elements indented on successive lines, as above.

Miser format is used when space is very tight, and a long list must be printed. Each element is printed on a new line, elements after the first being indented by one space:

```
(COND
 ((ZEROP X) 1)
 (T (TIMES X (FACT (SUB1 X)))))
```

In the order in which they have been presented, each successive format uses more vertical lines (1, 2, and 3, respectively) and less page width (50, 36, and 31, respectively). In general, pretty-printer tries to print S-expressions in as few vertical lines as possible, so it is biased towards using linear and open format. Overuse of open format will squash a long list against the right hand margin; too much miser format sacrifices readability. I have a truncated look-ahead that makes a decision between these two based on a number of parameters; it works at least acceptably.

In addition, special formats are used for some functions such as LAMBDA, PROG, SETQ, and others. The function QUOTE is converted into the macro character '. Sometime in the future, a macro format facility will be incorporated that will enable the user to define his own formats for special handling of specified list structures.

The pretty-print program is a group of ALISP functions contained on the indirect-access ALISP file PRINFNS in the ALISP library. They are loaded with the initial ALISP system. Both the filing function (in particular, GRIND) and the editing package (PP command) make use of the pretty-print functions.

2.2 Pretty-Print Functions

PPRINT is used for pretty-printing lambda-expressions. It is a FLAMBDA of one argument, the name of the function to be printed. To print the function FACT, for example, use:

```
(PPRINT FACT)
```

PPRINT will return NIL if its argument does not have a valid lambda-expression in its value cell. PPRINT normally returns the name of the function it pretty-printed.

PPRINE is used to pretty-print S-expressions in general. It is a LAMBDA expression of one argument. To pretty-print the value of FOO, for example, use:

```
(PPRINE FOO)
```

II Chapter 3

ALISP EDITING

The ALISP editing package is powerful means for changing the structure of S-expressions in the ALISP system. The editor is written as a group of ALISP functions, which are in the file EDITFNS on the ALISP library. The editor is automatically loaded with the initial ALISP system. One command, PP, uses the pretty-print package (section II.2).

A list editor takes some getting used to; but once learned, the ALISP editor is one of the most valuable tools in the ALISP system for modifying large programs. The following sections explain how to call the editor, key editing concepts, and the command repertoire for the ALISP editor.

3.1 Calling the Editor

There is only one function call for editing: EDIT, a LAMBDA expression of one argument. EDIT will only work on non-atomic arguments.

To edit a function definition, use:

(EDIT FNNAME)

where FNNAME is the name of the function. Since function definitions are contained in the value calls of atoms, EDIT gets the value of FNNAME, that is, its lambda-expression function definition. This is the most common EDIT call.

To edit a plist (in this case, the plist of FOO), use:

(EDIT (PLIST 'FOO))

Once the EDIT function has been called, it responds by printing the editing prompt character ":" and waiting for input from the terminal. The user then types editing commands, as many per line as he wishes (or can fit), which the editor interpretes and applies to its argument. The editing is ended with the command END, at which point the function EDIT exits with value END/ EDIT. All changes made to the edited list are permanent, that is, they retain their effects after the exit from EDIT. A sample session is given below in Dialogue 3.1.

Dialogue 3.1
A Sample Editing Session

?(EDIT FACT)	The recursive factorial function is being edited.
:P 1 LAMBDA 2 (X) 3 (COND ((ZEROP X) 1) ...)	The colon character is the editor prompt. A P command causes the elements of the list being edited to be numbered and printed.
:(1 FLAMBDA) END END EDIT ?	Two commands were given on one line. The first command replaced the first element of the list (which was LAMBDA) with the atom FLAMBDA. The second command caused the editor to exit. The function FACT is now a FLAMBDA rather than a LAMBDA function.

Note that the editor does not print a response to every command it is given. At the end of the edit, the message "END EDIT" is printed.

Errors can occur on certain editing commands. If this happens, an error message is printed, the rest of the input line is cleared, and the editor asks for more input. No cause for alarm.

3.2 Editing Concepts

The editor looks at only one list at a time. At any given moment, the list the editor is looking at is called the current level (CL). When the editor is first entered, the CL is set to its argument. The editor is able to desend from the CL by making non-atomic elements of the CL into a new CL; and ascend from the CL (if it is not already at the top) by reversing this process (see 3.3.2 below).

The elements of the CL are referred to in editing commands by number, i.e., the first element is 1, the second 2, etc. Some commands allow a reverse specification: -1 is the last element, -2 the second from last, etc. If an attempt is made to reference a non-existant element (e.g., 10 on a five element list), then a BOUNDS error will be given. The last CDR element of a list can be accessed by using an asterisk. In figure 3.1 some illustrations of the element numbering system for the CL are given.

The CL must always be a list, with at least one element. An attempt to make an atom the CL will result in an ATOMIC SUBLIST error, since the CL must always be a list. If the CL is a general S-expression rather than a true list, there is no problem, since an asterisk can be used to reference the final CDR. It may be wiped out by certain commands, such as adding elements to the end of the CL.

The CL corresponds to file pointers in a text-oriented editor. Instead of moving back and forth along text in a sequential file, however, the CL, as the center of the editor's attention, moves up and down through levels of a list. A normal sequence of list editing is to find the right CL, replace an element of the CL, and go back to the original level.

The CL is really the only concept needed to start using the editor.

3.2.1 Editor Values

Note - this section is for sophisticated users of the editor; it may be skipped by those just learning to use the editor.

Within the editor, it is convenient to have a means for refering to different parts of the CL, or to other lists defined within an edit. The editor uses a $ convention for this purpose. The $ character, wherever it appears in an editor command, refers to the editor value of the expresison immediately following it. The following are pre-set editor values:

$n -- element of a list
n is an integer. If positive, this is the nth

Figure 3.1
The CL and its Elements

CL	CL Elements
(FOO BAR)	1 FOO -2 2 BAR -1
(FOO (A B) BAR,MOO)	1 FOO -3 2 (A B) -2 3 BAR -1 MOO
((A (B)) (FOO) BAR ((C D) E))	1 (A (B)) -4 2 (FOO) -3 3 BAR -2 4 ((C D) E) -1

element of the list; if negative, the nth element from the bottom of the list. This value is actually a single-level copy of the specified element, so it can be used, for example, in a replace or add command, without fear of creating a circular list structure.

$STR -- extracted list
This is a copy of the list extracted by the extract command. If none has been extracted, $STR has value NIL.

$x -- set value
x must be an nglitat, but not STR. Value is the expression x has been set to using the SET editor command (see below, section 3.3.6).

The editor values are very handy when doing extractions or multiple replacements of the same list, or when used in conjunction with the search commands. Examples of their use will be given with individual editor command descriptions. Note that, within the editor, the $macro has been re-defined to yield editor values, rather than doing an immediate evaluation.

3.2.2 Command Format

There are two basic types of command formats:

1. com
2. (com arg1 arg2 ... argn)

where com is a one-to-four-letter command, and arg thru argn are arguments to the command. The form (com), with no arguments, is equivalent to the first format above. Some commands can be used with either format, while some take just one or the other. Extra arguments to a command are always ignored; too few will cause an error. If the editor cannot recognize a command, it will give an error.

Editor values ($) can be used at any point within a command format, or at any level within an argument. They are translated directly upon input into their actual values (except for the searching commands, which hold back assignment of some editor values until they complete a successful match; see 3.4 below).

3.3 Editor Commands

This section gives a complete description of all the editor commands, together with examples of their use. The recursive factorial function FACT defined earlier in the manual (see section II.2) is the principal list used in these examples.

3.3.1 Printing and Listing

One of the first requirements of editing is that you know what you are editing. Two commands, P and PP, enable all or parts of the CL to be displayed at the terminal.

P -- print elements
The P command prints and numbers elements of the CL. It uses the function HALFPRI for printing, so that only the first four atoms of long elements are printed. Both command formats can be used. The first format causes all elements of the CL to be printed. If the second format is used, arg thru argn must be integers specifying elements of the CL to be printed. Negative integers can be used.

PP -- pretty-print elements
The PP command pretty-prints the CL or elements of the CL. If the first format is used, the entire CL is pretty-printed. If the second format is used, arg thru argn must be integers specifying elements of the CL to be pretty-printed (negative integers are ok); elements are numbered and pretty-printed.

Examples of the printing commands are given in Dialogue 3.2 below.

3.3.2 Traversing List Structures

It is often desirable to change the CL, in order to get the editor closer to a particular structure that must be operated on. The list traversing commands q, GO, RET, U, and TOP provide an easy facility for going down and popping back up thru list

Dialogue 3.2
The Listing Commands P and PP

```
?(EDIT FACT)
:P

1 LAMBDA
2 (X)
3 (COND ((ZEROP X) 1) ... )
```

The P command by itself numbers and prints the whole current level. Note that HALFPRI format is used to reduce the wordiness of the printout.

```
: (P 2 1)

2 (X)
1 (LAMBDA)
```

The second format for P prints only the given elements of the CL. Note that the elements are printed in the order given by the P command.

```
:PP

(LAMBDA (X)
    (COND ((ZEROP X) 1)
          (T (TIMES X (FACT (SUB1 X))))))
```

PP by itself pretty-prints the whole CL.

```
:(PP -2)

2 (X)
:
```

Here the negative element specification was used to pretty-print the second element from last of the CL.

---

levels.

n -- set CL to nth element
    n is an integer, either positive or negative, specifying the element of the CL which is to become the new CL. The element must be non-atomic or an error is issued.

U -- set CL up levels
    The U command backs the CL up to a previous level after the n command has been used. With no arguments, it backs up one level; with one argument, a positive integer, it backs up the number of levels specified by that integer. If you attempt to back up past the top level of the edited list, an error will be issued, and nothing done.

TOP -- set CL to top level
    Sets the CL to the top level first used as CL when the edit was entered. Used with no arguments.

GO -- set level marker

The GO command sets a level marker that can be returned to with a RET command. GO saves the CL so that many list traversal commands can be used without keeping track of them; RET always returns the CL to its value at the last GO. GO commands can be nested. Used with no arguments (first format).

RET -- return to level marker

The RET command returns the CL to the first value when the last GO was called. If no GO had been called, an error message is printed. Doing a RET wipes out the last GO marker. Used with no arguments.

Examples of these commands will be found below in Dialogue 3.3.

3.3.3 Element Manipulation

These commands actually change list structure. Elements of the CL can be replaced or removed, and new elements can be added.

n -- replace element

n must be a positive or negative integer specifying the element to be replaced. Only the second command format may be used. arg1 thru argn are elements which will be substituted for the specified element. At least one argument must be present.

D -- delete elements

Only the second format is allowed for the D command. arg1 thru argn are integers specifying elements to be deleted, in any order (negative integers are ok). At least one argument must be present. Duplicate arguments are eliminated. Note well: the final element of a list cannot be removed, that is, the D command cannot delete all elements from the CL. If this is attempted, an error will be issued.

A -- add elements

Only the second format is allowed for the A command. arg1 must be a positive or negative integer specifying an element after which additional elements are to be added. If arg1 is zero, elements are added before the first element. arg2 thru argn are the elements to be added.

Note that all of these commands can change the number of elements within the CL, and this affects the operation of subsequent commands. If you are not sure of element numbering after one of the above commands, do a P command to print out all the elements of the CL, or a (P -1) to print out the last element and thus find the number of elements in the CL. Examples of the element manipulation commands will be found in Dialogue 3.4

Dialogue 3.3
The List Traversing Commands

```
?(EDIT FACT)
:3 P

1 COND
2 ((ZEROP X) 1)
3 (T (TIMES X (FACT ... )))
```

The command 3 causes the CL to become the third element of the old CL, i.e., the COND form of the FACT function (refer back to Dialogue 3.2, and note that the P command reveals the COND form as the third element). Now the editor is acting on the COND form, so that the P command prints the elements of this form.

```
:-1 P

1 T
2 (TIMES X (FACT (SUB1 ...)) )
```

The command -1 causes the CL to become the first element from the end of the old CL, i.e., element 3, the second argument to the COND function.

```
:U (P 1)

1 COND
```

The U command causes the CL to move back up one level, undoing the effect of the -1 command. Now the CL is the COND form again.

```
:GO 3 2 P

1 TIMES
2 X
3 (FACT (SUB1 X))
```

The GO command saves the CL for later recall by the RET command. The 3 and 2 commands traverse down the COND form, finally ending up by making the CL be the TIMES list.

```
RET (P 1)

1 COND
```

The RET command causes the CL to be reset to the list which was the CL when the last GO command was executed. The CL is back at the COND form again.

```
:TOP (P 1 2)

1 LAMBDA
2 (X)
:END
END EDIT
 ?
```

The TOP command sets the CL back to the original CL when the editor was entered, i.e., the LAMBDA form.

below.

3.3.4 Level Manipulation

At times it is desirable to change the level position of an element, either to make it a sub-element, or to make its sub-elements into CL elements. Take, for example, the list:

    (TIMES (X FACT (SUB1 X))))

There is an extra set of parentheses after the atom TIMES. What should be done is the elements of the list (X (FACT (SUB1 X))) should be made elements of the list (TIMES ...), that is, they sould be moved up one level. This can be accomplished with the O command.

Similarly, it is often necessary to convert elements to sub-elements, for example in the list:

    (FACT SUB1 X)

What should be done here is combine the elements SUB1 and X into a single element, (FACT (SUB1 X)). This can be accomplished with the C command. In combination, the O and C commands can effect any arbitrary level manipulations required.

O -- open elements into CL
The O command takes sub-elements of specified elements of the CL and moves them up as elements of CL. Essentially, it removes a set of parentheses from specified elements. Only the second command format can be used. arg1 thru argn are positive or negative integers specifying elements to be opened, in any order. Duplicate arguments are ignored. All specified elements must be non-atomic.

C -- close elements in CL
The C command combines one or several elements from CL into a single element of CL. It takes two arguments, both positive or negative integers. arg1 specifies the first element to be included, arg2 the last; obviously, arg1 must specify an element before

Dialogue 3.4
Element Manipulation Commands

?(EDIT FACT) :3 3 2 P 1 TIMES 2 X 3 (FACT (SUB1 X))	The three commands 3 3 2 take the CL down to the level of the TIMES function.
:(1 PLUS) (P 1) 1 PLUS	The first command here changes element 1 from TIMES to PLUS.
:(-1 Y (Z)) P 1 PLUS 2 X 3 Y 4 (Z)	The -1 command replaces the last element of the CL with two elements, Y and (Z). Thus the length of the CL has been expanded from three to four.
: (D 3 -1) P 1 PLUS 2 X	The D command deletes the third and fourth elements from the CL.
:(A 2 (FACT (SUB1 X)) Y) P 1 PLUS 2 X 3 (FACT (SUB1 X)) 4 Y	The A command adds, after element 2 of the CL, the two additional S-expressions (FACT (SUB1 X)) and Y. Again, the length of the CL has been changed.
:(1 TIMES) (D -1) PP (TIMES X (FACT (SUB1 X))) :	The 1 command replaces the first element with TIMES, and the D command deletes the last element. Now the original TIMES list has been restored.

arg2 in the CL. All elements from arg1 to arg2 are closed, inclusively. If arg1=arg2, only a single element is closed. Essentially, the C command puts an extra set of parentheses before arg1 and after arg2.

Example of these two level manipulation commands will be found below in Dialogue 3.5.

Dialogue 3.5
Level Manipulation Commands O and C

```
?(EDIT FACT)
:P (O 3) P

1 LAMBDA
2 (X)
3 (COND ((ZEROP X) 1) ... )

1 LAMBDA
2 (X)
3 COND
4 ((ZEROP X) 1)
5 (T (TIMES X (FACT ... )) )
```

The O command was used to open the third element of the CL. This caused the COND list to be opened onto the CL, so that elements of the COND list are elements of the CL. Note that the length of the CL changed, as is usual with the O or C commands.

```
:(C 3 5) P

1 LAMBDA
2 (X)
3 (COND ((ZEROP X) 1) ... )
```

The C command closed the elements 3 thru 5 of the CL as one list. This reversed the effects of the O command; O and C can be used as inverses.

```
:3 2 P

1 (ZEROP X)
2 1
:(C 2 2) (P 2)

2 (1)
```

First the CL is sent down several levels to the first COND clause. Then the C command is applied to the second element of the CL, 1. Note that when the C command has equal arguments, the effect is to make a one-element list.

```
:(O 2 1) P

1 ZEROP
2 X
3 1
```

The O command opens both the first and second elements of the CL, yielding a new CL with 3 elements.

```
:(C 1 2) P END

1 (ZEROP X)
2 1
END EDIT
?
```

The C command closed the first two elements, restoring the COND clause to its original form.

3.3.5 Undoing

Every command in the editor is undoable, which is to say its effects are reversible. Think of commands which are given to the editor as being kept in a stack. At any given moment, the state of an edit is given by this command stack. Undoing causes the last entries to the stack to be popped and their effects reversed. For example, suppose the stack is:

```
P
(D 3 4)
2
(3 (QUOTE FOO))
GO
1
(A 2 X)
```

Most recent entries are at the bottom of this list. If the last three commands, say, are undone, the stack will look like:

```
P
(D 3 4)
2
(3 (QUOTE FOO))
```

and the state of the edited list will be exactly the same as if only the four above commands had been given. Subsequent undo's will pop the stack further; you can't undo an undo. All commands, even those which don't affect list structure such as P or PP, are inlcuded in the command stack. The B command is used for undoing.

B -- back up command stack

If the first format is used, the last command is undone. If the second format is used, arg1 must be a positive integer specifying the number of commands to be undone.

CLR -- clear command stack

The CLR command undoes the effect of every editor command. The CLR command is a last resort if you have totally screwed the function you are editing; it restores the function to its original form. No arguments are used.

Examples of the undo commands will be found in Dialogue 3.6 below.

3.3.6 Setting and Extraction

Setting and extraction commands are used in conjunction with editor values to move lists from one place to another in an

Dialogue 3.6
The Undo Commands B and CLR

```
?(EDIT FACT)
:3 2 P

1 (ZEROP X)
2 1
:(1 NIL) P

1 NIL
2 1
```

The editor here goes down several levels of list structure, then changes the value of the first element of the CL from (ZEROP X) to NIL. The command stack now has five com- i.e., the print and mands on it: two level-changing commands, a print command, a replace command, and another print command.

```
:(B 2) P

1 (ZEROP X)
2 1
```

The B command undoes the effect of the last two editor commands, replace commands. Note that the first element of the CL has been restored to its original value. The command stack now has four commands on it: two level-changing commands, a print command, and the print command just completed.

```
:(B 3) P

1 COND
2 ((ZEROP X) 1)
3 (T (TIMES X (FACT ... )) )
```

This B command undoes the last three commands, so that effectively only one command remains, the 3 command at the beginning of the edit. Note that the B command undoes the effect of level changes in the CL, as well as changes in list structure.

```
:CLR P END

1 LAMBDA
2 (X)
3 (COND ((ZEROP X) 1) ...)
```

The CLR command undoes the effect of every command; the CL is back at the top level of the FACT function.

edited structure. The extract command, STR, sets the extract buffer to the CL or an element of the CL. The set command, SET, gives a name to a list structure so that it can be referenced in subsequent commands as an editor value (see section 3.2 above).

SET -- set editor value
Only the second format can be used with the SET command. arg1 must be an nglitat, but not STR. arg2 can be any S-expression. The effect of SET is to cause arg1 to have an editor value of arg2. This value can be accessed later by using "arg1" in an editor command.

STR -- set extract buffer
The STR command sets the extract buffer to either the CL or an element of the CL. The previous value in the extract buffer is flushed. If the first format is used, the buffer is set to the whole CL. If the second format is used, arg1 must be a positive or negative integer specifying an element of the CL to which the buffer is set. The value in the the extract buffer is accessed with "$STR".

Examples of the setting and extraction commands will be found in Dialogue 3.7 below.

3.3.7 Conditional Editing

There is one conditional command, IF, for use with the editor. It is most valuable when used in conjunction with the search commands (see section 3.4 below). The IF command evaluates an S-expression and, if the result is non-NIL, applies a bunch of editor commands.

IF -- conditional edit
Only the second format can be used with the IF command. arg1 is an S-expression which is evaluated with EVAL. If the result is non-NIL, arg2 thru argn are treated as editor commands. If the result is NIL, no editor commands are called. Editor values ($) can be used in arg1 for testing.

Examples of the conditional editing command will be found below in Dialogue 3.8.

3.4 Search Commands

These are the most powerful editing commands. With them, the user can do context-oriented editing, by looking for list

Dialogue 3.7
The Setting and Extraction Commands SET and STR

```
?(EDIT FACT)
:(SET FOO (TIMES X Y)) 3 3 P

1 T
2 (TIMES X (FACT SUB1 ... )))
```

The SET command sets the editor value of FOO to the list (TIMES X Y). The ALISP value of FOO is unaffected, and will remain the same as it was before the edit.

```
:(STR 2) (2 $FOO) (P 2)

2 (TIMES X Y)
```

This command sets the extract buffer to the second element of the CL. The next command replaces the second element with the editor value of FOO, i.e., the list (TIMES X Y).

```
:(2 $STR) PP

(T (TIMES X (FACT (SUB1 X))))
```

The STR command restores the second element to the value of the extract buffer. the extract buffer is not changed by this operation.

```
:(A 1 $FOO) (P 1 2)

1 T
2 (TIMES X Y)
:
```

The editor value of FOO is used here in the A command to add the list (TIMES X Y) after the first element of the CL.

---

structures which match a pattern. The heart of the search commands is the pattern matcher.

3.4.1 Pattern Matching

The workings of the pattern matcher must be understood before the search commands can be used. Basically, the matcher takes a pattern composed of constants and variables, and tries to match it to structures in the list being edited. The pattern can either be a single atom, or a multi-level list structure. A ? character in the pattern indicates a pattern variable. Examples of patterns:

```
FOO
```

Dialogue 3.8
The Conditional Editing Command IF

```
?(EDIT FACT)
:(P 1) (IF (EQ $1 'LAMBDA)

1 LAMBDA
:  (1 FLAMBDA) (2 (Y)) ) P


2 (Y)
3 (COND ((ZEROP X) 1) ... )
```

The first print command shows that the first element of the CL is LAMBDA. The second command is an IF command, which continues across several input lines. The first argument of the IF command uses the $1 to test if the first element of the CL is the atom LAMBDA. Since this test is successful, the IF proceeds to use the rest of its argument as editor commands, changing the first element of the CL to FLAMBDA, the second element to (Y).

```
:(IF (AND (EQ $1 'FLAMBDA) (ATOM
:            $2)) PP)
:
```

This IF command fails because the second element of the CL is not atomic.

```
:(IF (LISTP $2) (1 LAMBDA)
:    (2 (X)) (P 1 2))

1 LAMBDA
2 (X)
```

This IF command succeeds, replacing the first element of the CL with LAMBDA, the second with (X).

```
(FOO BAR)
(FOO ? (BAR ??))
(?VAR FOO (?VAR2 BAR))
(?(VAR (EQ =VAR 'COND)) BAR FOO)
```

Everything not immediately preceeded by a ? character is a constant, and must match exactly its corresponding part in a list structure in order for the match to succeed.

A ? by itself (third pattern above) matches any element of the data. Thus:

```
(FOO ? BAR)     matches       (FOO NIL BAR)
                              (FOO (QUOTE MOO) BAR)
                              (FOO FOO BAR)
                but not       (FOO BAR)
```

(FOO MOO (MAR) BAR)

A ?? by itself matches any number of elements in the data. Thus:

```
(FOO ?? BAR)   matches      (FOO NIL BAR)
                            (FOO (QUOTE MOO) BAR)
                            (FOO (QUOTE MOO) BAR)

               and also     (FOO BAR)
                            (FOO MOO (MAR) BAR)
```

Along with the variable indicator characters, a variable to be bound to part of the data at match time can be specified. For instance, in the fourth pattern above, ?VAR matches any first element of the data, and binds VAR to that element. The binding is accomplished by setting the editor value of VAR to the element if the match succeeds; it can then be accessed with $VAR. Examples:

pattern	data	VAR set to
(FOO ?VAR)	(FOO BAR)	BAR
(FOO ?VAR)	(FOO (QUOTE BAR))	(QUOTE BAR)
(FOO ??VAR)	(FOO)	NIL
(FOO ??VAR)	(FOO BAR)	(BAR)
(FOO ??VAR)	(FOO BAR (MOO))	(BAR (MOO))

Finally, conditions can be on the type of data which a variable will match. This type of pattern is present in the fifth pattern above. The format is:

```
?(var exp)
or
??(var exp)
```

where *var* is an nglitat, and *exp* is any S-expression. On encountering a form like this in the pattern, the matcher first binds *var* to the corresponding part of the data, then evaluates *exp* (which contain =var to access the binding of *var*) and succeeds if the result is non-NIL. Thus:

```
(FOO ?(VAR (EQ (CAR =VAR) 'COND)) BAR)

        matches  (FOO (COND) BAR)
                 (FOO (COND (X X) (T T)) BAR)
        but not
                 (FOO (X COND) BAR)
                 (FOO BAR)
                 (FOO COND VAR)
```

One must exercise caution when using this format for pattern variables. Note that attempting to match the last data, (FOO COND BAR), will generate an error when the expression (CAR =VAR) is evaluated in the pattern. To be perfectly safe from this kind of error, the following expression should have been used:

```
?(VAR (AND (LISTP =VAR) (EQ (CAR =VAR) 'COND)))
```

When using ? or ?? with variables, it is important that there be no space between the macro character and the variable expression. The following two patterns match wholly different data:

```
(? VAR FOO)
(?VAR FOO)
```

The first pattern matches data lists with three elements; the second matches data lists with two elements and binds VAR to the first element.

Both ?, $, and = are macro characters in the editor, and cannot thus be inputted in pnames unless slashed. Do not use STR as a variable with ? unless you do not intend to use the extract editor values ($STR, see section 3.2.1), with which it will conflict.

3.4.2 Find

The find command, F, searches for a data structure which matches a pattern, then applies the editor to the list of which that data is an element. The find format is:

```
(F pat com1 com2 ... comn)
```

where pat is a valid pattern for the matcher as discussed in 3.4.1 above, and com1 thru comn are optional editor commands to be applied each time a match to pat is found.

The order of events on invocation of the find command is as follows:

1. The CL is searched linearly from left to right for a match to pat. If no match is found, the find ends with the message "END FIND".

2. When a match to pat is found, the editor is called with a CL of the list in which the match was made. All variables bound in the match have their corresponding editor values (see 3.4.1 above).

3. The editor checks to see if there are any com in the find command; if so, they are interpreted as editor commands. If not, the CL is printed.

4. The editor is now operating with the matched list as its top level. All editor commands, even more

F's can be called at this point. Undoing will work, but only with commands given in this particular find call. The U command cannot be used to go back further than the originally matched list.

5. To exit from the find, either END or ENDF can be used. END causes the command to go back to step 1 and find the next occurrence of the pattern. ENDF causes the find to exit without searching for other matches. Both END and ENDF could, of course, have been used as comn in the find command itself. On exit from a find, the CL is the same as before the find was entered.

Basically, there are two modes to the find command. Interactive mode is when one of the com are included, so that the CL is printed each time a match is successful, and the editor waits for input from the user. Automatic mode is when the com are included, and the last one is an END or ENDF command. In automatic mode, nothing is printed out (unless one of com commands is a P or PP) and the find exits automatically, without waiting for user input. A compromise mode is achieved if com are included , but no END or ENDF. Then a few commands are applied automatically on each match, and user input is also accepted.

Some examples of these three modes of the F command are given below in Dialogues 3.9, 3.10, and 3.11.

The B command considers everything done under one F command to be a single command. Thus backing up the command stack after doing an F backs up past everything done in that F. Within the F, a local command stack is used, so that commands within an F command can be undone. Once the F exits, however, all these commands are considered as one by the undoing command.

3.4.3 Replace

The replace command, R, searches for a data structure which matches a pattern, and replaces or deletes that structure. The replace format is:

(R pat dat)

where pat is a valid pattern for the matcher, and dat is omitted, a delete is done.

Within dat, editor values can be specified which depend on matcher variables in pat. A copy of dat is used in replacements, so that there is no need to fear circular list structures.

Finally, it is impossible to delete the last element in a list, and the R command thus cannot delete all elements from the

Dialogue 3.9
The F Command: Interactive Mode

?(EDIT FACT) :(F X)	The F command searches for all occurrences of the atom X.
1 X END	The first X found was the variable list of FACT. Note that F applies the editor the list of which X is an element. The END command causes the F command to search for the next occurrence of X.
1 ZEROP 2 X :(2 Y) END	Here the user replaces X with Y, then uses END to start searching for the next X.
1 TIMES 2 X 3 (FACT (SUB1 X)) : 3 P TOP P 1 FACT 2 (SUB1 X) 1 TIMES 2 X 3 (FACT (SUB1 X))	The user goes down one level to element 3, then goes back to the top level with TOP. Note that the TOP command will only go back as far as the matched list when the F command is in effect.
:ENDF	The ENDF command halts the F command.
:(F (ZEROP ?V)) 1 (ZEROP X) 2 1 :(2 $V) P 1 (ZEROP X) 2 X	The pattern given to the F command matches the list (ZEROP X). The variable V has an editor value of X.

CL. If you attempt to do this, an editor error will be issued.

Be extremely careful with the R command, it is a very powerful list-altering function. Its effects can be undone, though, with the B command.

3.5 Editor Errors

Dialogue 3.10
The F Command: Automatic Mode

```
 ?(EDIT FACT)
:(F X (A -1 FOO) END)
END FIND
:PP

(LAMBDA (X FOO)
  (COND
   ((ZEROP X FOO) 1)
   (T (TIMES X (FACT (SUB1 X)) FOO))))
```

This F command searches for a list which contains the atom X, and adds the atom FOO as the last element to each such list. Note that the END command is given within the F command.

```
:(F X (D -1) END)
END FIND
:
```

This F command reverses the effects of the previous one by deleting every list containing X.

Dialogue 3.11
The F command: Mixed Mode

```
 ?(EDIT FACT)
:(F X
:
  (IF (NULL (MEMBER 'ZEROP $0))END)
:  P)

1 ZEROP
2 X
:ENDF
:
```

Here the F command searches for all lists which contain the atom X. The IF command ignores all those which do not contain the atom ZEROP; when one is found which does, the F command prints it.

The editor traps all errors, The message printed on the terminal should be of the form:

USER-ER FROM EDIT
message

The edit command which called the error is not performed, and any commands after it in the input buffer or in a find or IF command string are aborted.

III CHAPTER 4

Compiler

An overlay compiler is available for use with ALISP functions. Savings in space and execution time of functions results from using the compiler.

4.1 Overlay Compiler-Assembler

This package performs compilation and loading of ALISP functions. Savings in execution time, core storage, and GC time will result from compilation of user functions.

Complete linkage between compiled functions and other functions and global variables is possible. However, tracing and backtracing of functions called from compiled functions will never occur. Debug your programs before you compile them.

User functions to be compiled must reside on an ALISP file. The compiler accepts any number of such files as input, and either loads the compiled code directly into core, or saves it in another file for later loading by the assembler.

Where program segmentation makes it feasible, groups of functions can share the same core space as overlays. The overlay option is described in more detail below.

The compiler is invoked with the command:

(COMPILE flist fn)

Neither flist or fn is evaluated. flist is either a single file name or a list of file names which hold the functions to be compiled. If the first atom of flist is overlay, an overlay will be compiled (see below).

fo directs the output of the compiler. If it is the atom LINK, the compiled functions will be loaded directly into core. If not, fo is the name of the file to which the compiled code will be sent. This code may be loaded at any later time with LAP, the ALISP assembly program.

The compiler will print a list of functions compiled, and the amount of core used if the LINK option was selected. In the latter case, LAP may also print some error messages in loading incorrectly compiled functions. The user should consult the LAP error section below.

In addition to compiling function definitions, the compiler will input all plist definitions (LINK option) or send them to the output file. If an entry value is not a function. or has been declared NOCOMPILE, then the compiler will also input that value as an S-expression (LINK option) or send the value definition to the output file. Thus, the COMPILE function acts just like the INPUT function in loading an ALISP file into the ALISP system; the only difference is that some functions are compiled into machine subroutines.

Before compiling a file, the user should check carefully that all function and variable linkages are correct, as described below.

4.2 Function Linkage

When a function is compiled, the value of the atom which is the function name changes from a lambda-expression (list structure) to a machine language subroutine (PNUM). All uncompiled functions which call the compiled function will simply use this PNUM instead of the lambda-expression. Even functional argument uses of the compiled function will work correctly, thanks to the McCarthy EVAL. No problems here, unless you were explicitly manipulating the list structure of a lambda-expression you compiled.

When a function is compiled, the function calls it makes also get compiled into various linkages. These fall into two main types:

i. Machine lambda subroutine calls.

Calls to pre-defined functions, previously compiled functions, and concurrently compiled functions all use a very fast link. All information as to the name of the function being called is lost; instead a direct jump to the function code is made. If the function being called is later re-defined, this linkage will still be to the old function definition. Tracing and backtracing of this linkage is not possible.

If a function to be compiled is only called by functions which are compiled along with it, then it is no longer necessary to keep the called function name around after compilation. Removing the name relieves conflicting name problems and frees up space on the OBLIST. If a function is declared NONAME, its name will vanish after compilation (see section on declarations below).

ii. Lambda-expression function calls.
Calls to undefined or uncompiled functions use this linkage. The name of the called function is retained, and if the called function is re-defined, the linkage will be to the new function definition. Tracing of this linkage is not possible. If the called function is undefined on execution of the call, a FUN-ERR from APPLY will result.

A function call may be explicitly compiled with this link by using the LAMBDA or FLAMBDA declaration (see below, Declarations).

One further problem exists in linking to functions which are undefined at compile time, or which are used as functional arguments. The compiler must make a decision as to whether an undefined function's arguments are evaluated or not. The default is that they are evaluated. The user can cause them to be unevaluated by declaring the undefined function to be of type FLAMBDA (see below, Declarations).

All functional expressions within a compiled function are also compiled. For example, in the function:

```
(DE FOO (BAR)
     (DREDGE BAR (LAMBDA (C) (CONS C MOO)) (CDR BAR)))
```

the lambda-expression in the call to DREDGE will be compiled into a PNUM. If you wish to pass a lambda-expression as a list, rather than having it compiled, use (QUOTE (LAMBDA ...)) rather than (LAMBDA ...).

4.3 Variable Bindings

The variables of a function can be compiled in either of two ways.

i. If the variables are referenced only within that function, then the variables are compiled to locations on the stack, and their names are lost. They will no longer reference the value cell of the variable atom name. Any functions, compiled or uncompiled, which the compiled function calls, cannot reference these variables.

ii. Variables whose value cells must be referenced outside the compiled function (global varaibles) must be declared SPECIAL (see below, Declarations). A SPECIAL variable in a compiled functions retains its name, and the compiled function will reference the value cell of that atom. Thus compiled SPECIAL variables will perform as normal uncompiled variables, and will show up on backtracking output.

All system switches (OUTUNIT, INUNIT, etc.) are automatically declared SPECIAL.

Be careful when using compiled function variables as free variables in functional expressions. For example, the function:

```
(DE FOO (X)
     (BAR Y (LAMBDA (Q) (LIST Q X))))
```

should have X declared SPECIAL, since it is unbound in the lambda-expression. An exception to this rule is the MAP group of functions. The function:

```
(DE FOO (X)
     (MAPC Y (LAMBDA (Q) (LIST Q X))))
```

need not have X declared SPECIAL when compiled.

Remember to declare SPECIAL all variables in all lambda-expressions of a compiled function, which are to be used globally (freely) in functions called by the compiled function.

4.4 Declarations

There are four types of declarations the compiler listens to. All can be sent to a file using the DECFILE function (see 5.).

i. NOCOMPILE
An entry in a file, if declared NOCOMPILE, will be inputted unchanged by the compiler. The atom NOCOMPILE must be put on the declarations list of the entry.

ii. NONAME
An entry in a file, if declared NONAME, will lose its name when compiled, and cannot be referenced by any other function. All calls to the compiled function by compiled functions in the same file(s) will be correct, however. Note that all references to the atom declared

NONAME, whether to its plist or value, will not find the atom; it will be WIPE'd. The atom NONAME must be put on the declarations list of the entry.

iii. SPECIAL
Variables used globally (freely) outside a compiled function (except in MAP function calls) should be declared SPECIAL. All system switches are automatically SPECIAL.

Variables bound by a function can be made SPECIAL by putting a list of the form:

```
(SPECIAL var1 var2 ... varn)
```

on the declarations list for that function. The SPECIAL status lasts only during the compilation of the function; other functions which have these atoms as variables must also declare them SPECIAL if they are to be used globally. Note that functions which merely refer to the global variables, as opposed to binding them in a LAMBDA or PROG expression, need not declare them SPECIAL.

iv. LAMBDA or FLAMBDA
A compiled function call can be declared either a LAMBDA or FLAMBDA function call (evaluated arguments, respectively) if the function is either undefined, or a defined function type is to be overridden. For example, the form:

```
(FOO X Y)
```

where FOO is an undefined function at compile time, can be compiled either as a LAMBDA or FLAMBDA function by putting the list:

```
LAMBDA
(or  FOO)
FLAMBDA
```

on the declarations list of the function where the form occurs. Default for undefined functions is LAMBDA.

4.5 Restrictions on Compiled Functions

1. All GO's and RETURN's to a compiled PROG must occur explicitly in the body of the PROG, and not in functions called by the PROG.

2. No GO's with evaluated arguments may be used in compiled PROG's, i.e., all GO statements must be of the form:

(GO label)

where label is an atomic PROG label. Any references by GO to undefined labels will be trapped as an error.

3. GO's and RETURN's in compiled functions that are called by uncompiled PROG's will work correctly.

4. LABEL is uncompilable at present.

4.6 Defining Overlays

An overlay is a group of compiled functions that share the same core space. Swapping of overlay functions is automatic and involves a minimum of CP time, but disk time for each swap is appreciable (on the order of 200 milliseconds). Therefore reasonable logical segmentation of overlay function groups is important to reduce the number of swaps required. Each time a function not in fni is called, the correct overlay segment is swapped into core, the function is executed, and fni is swapped back into core at completion of the execution.

In all respects, overlay functions logically act like normal machine subroutines (PNUM's).

4.7 The Assembler (LAP)

IF compiler output has been sent to a file, then the assembler can be used to load that file. Use:

(LAP fn)

where fn is the name of the file that has the compiled code. The LAP program is almost as fast as the KRONOS COMPASS assembler, loading 140 lines of macro instructions per second of CP time.

Routines that must execute in minimum time (those bottleneck inner loops) can be hand-coded using a macro instruction set. Specifications on this set, as well as the internal specs on on ALISP necessary to write correctly linking code, are not yet available (cough, cough).

Appendix B

ALISP Control Card

This Appendix describes the legal parameters and default values for the ALISP control card. Both batch and TELEX origin jobs are considered.

The comma, period, and slash are the only valid ALISP control card separator and terminator characters. After each comma must appear one of the valid control card parameters listed below, unless the LD parameter is used; in this case, the user does his own control card processing.

Parameter	Meaning	Legal Values	Action
AS	Ascii character set	omitted	The value of TTYCHAR is set to T for timesharing origin jobs and NIL for batch origin jobs.
		AS	The value of TTYCHAR is set to NIL.
		AS=0	The value of TTYCHAR is set to T.
E	Echo control	omitted	ECHO is set to NIL for timesharing origin jobs, and set to SYSIN for batch origin jobs.
		E	ECHO is set to SYSIN.
		E=0	ECHO is set to NIL.
FL	Field length	omitted	The maximum field length for the ALISP job is set at 60000B (about 24K decimal).
		FL	Maximum field length is set to 100000B (32K decimal).

		FL=n	n must be an integer. The maximum field length is set to n.
IF	Input an ALISP file	omitted	no action.
		IF	The ALISP file MYFNS is inputted from the user's catalog.
		IF=pfnl=pfn2...	The ALISP files pfnl thru pfni are inputted from the user's catalog.
		IF=pfnl/usernum/passwor=pfn2...	Alternate user number and password are specified for the ALISP file.
LD	ALISP load file (overlay)	omitted	No action.
		LD	ALISP overlay file MYLOAD is loaded from the user's catalog.
		LD=pfn	ALISP overlay file pfn is loaded from the user's catalog.
		LD=pfn/usernum/passwor	Alternate user number and file password specification for the overlay file.
NEWS	ALISP system news	omitted	No action.
		NEWS	Most recent ALISP system news is printed on SYSOUT.
		NEWS=T	All ALISP system news is printed on SYSOUT.

PE	Print line length	omitted	PRINEND is set to 72 for time-sharing orgin jobs, 130 for batch origin jobs.
		PE	PRINEND is set to 100.
		PE=n	n must be an integer from 1 to 150. PRINEND is set to n.
SI	SYSIN unit	omitted	SYSIN is set to 0, i.e., the terminal on timesharing, the card reader on batch origin jobs.
		SI	The permanent file INFILE is opened as unit 1, and SYSIN is set to 1.
		SI=pfn	The permanent text file pfn is opened as unit 1, and SYSIN is set to 1.
		SI=pfn/usernum/passwor	Alternate user number and file password specified on pfn.
		SI=n	n is an integer from 1 to 4. SYSIN is set to local unit n. The local file ITAPEn should exist before ALISP is called.
SO	SYSOUT unit	omitted	SYSOUT is set to 0, i.e., the terminal on timesharing, and the printer on batch origin jobs.
		SO	SYSOUT is set to local output file unit 1. When ALISP exits, local file OTAPE1 will have the output from the supervisor.
		SO=n	n must be an integer from 1 to 4. SYSOUT is set to local output file unit n; when ALISP is exited, OTAPEn will have the output from the supervisor.
TL	Time limit	omitted	Time limit remains unchanged when ALISP is called.
		TL	Time limit is set to 64 CP seconds.

TL=n	_n_ must be an integer. Time limit is set to _n_.

Appendix C

Initially Defined Functions and Switches

This Appendix details the pre-defined function and switch names found in the ALISP system when it is initially loaded.

There are four columns in this table. The first column is the atom name. The second column is the type of its value, as either a function or a switch. Function types are given as SUBR, FSUBR, etc.; an integer immediately following it specifies the number of arguments to the function, if it takes a definite number. The third column is a brief description of the function or switch. The fourth column is a list of references to chapter sections in which the atom is described in more detail.

Within the description given here, arg1, arg2, etc., are used to name the arguments to the function; argn is used as the last argument. When an argument is qualified (such as, "the list arg1"), it indicates that the function normally takes that data type for its argument. These restrictions are not absolute, however; for instance, NCONC will work on atomic as well as non-atomic arguments. See the references for more detail.

Atom Name	Value Type	Description	References
ABSVAL	SUBR,1	Returns the absolute value of number arg1.	11.3.1
ADDEL	SUBR,3	Adds arg1 after SNUM arg3 element of list arg2.	10.3.2
ADDGT	SUBR,2	Address comparison of arg1 and arg2.	9.4
ADDLT	SUBR,2	Address comparison of arg1 and arg2.	9.4
ADD1	SUBR,1	Adds one to number arg1.	11.3.1

AND	LSUBR	Evaluates each argument until one returns the value NIL; else returns the result of the last evaluation.	8.1.2
APPEND	SUBR,2	Non-destructive merging of top levels of list arg1 and list arg2.	10.2.2
APPLY	SUBR,2	Applies function arg1 to argument list arg2.	6.2.4
APPLY*	SUBR*	Applies function arg1 to arg2 thru argn.	6.2.4
ARGN	SUBR,2	Returns the SNUM arg2 element of arg1.	6.3.1 10.2.1
ATLENGTH	SUBR,1	Returns character printing length of atom arg1 as an SNUM.	5.3.1
ATOM	SUBR,1	Returns T if arg1 is atomic, NIL if not.	2.2.3
BACKPRN	switch	Controls the backtracing printout. Initially NIL, i.e., normal backtracing printout.	12.1.1
BACKTRK	switch	Controls backtracing. Initially NIL, thus no backtracing.	12.1.1
BATCH	SUBR,0	Returns T if ALISP job is of batch origin, NIL if not.	15.3
BREAK	LAMBDA,2	BREAK supervisor function.	12.2.2
CAAAR to CDDDR	SUBR,1	Standard CAR and CDR functions; they work with NIL, but no other atoms.	10.2.1
CARS	SUBR,2	Returns a list of the first SNUM arg2 elements of list arg1.	10.2.1 10.2.2
CDRS	SUBR,2	Does SNUM arg2 CDR's of list arg1.	10.2.1
CLOSE	SUBR,2	Closes local output unit number arg2 as permanent file arg1. Returns arg1.	14.2.2
CLRBIT	SUBR,2	Clears the SNUM arg2 bit of LNUM arg1.	11.5

CONC	SUBR*	Strings lists arg1 thru argn together with multiple NCONC's.	10.3.1
CONCONS	SUBR*	Strings arg1 thru argn together with multiple CONS'es.	10.2.2
COND	LSUBR	Conditional function.	8.1.1
CONS	SUBR,2	Standard CONS function.	10.2.2
COPY	SUBR, 1	Returns a top-level copy of arg1.	10.2.2
COS	SUBR,1	Yields the cosine function of number arg1; arg1 is in radians.	11.3.2
CSHIFT	SUBR,2	Does circular shifting of LNUM arg1 by SNUM shift count arg2.	11.4.2
DCOPY	SUBR,1	Returns a complete (all-levels) copy of arg1.	10.2.2
DEFPROP	FSUBR,3	Puts arg3 on the plist of nlitat arg1 under label arg2.	10.1
DELETEL	SUBR,2	Destructively deletes the SNUM arg2 element of list arg1. Cannot remove last element of arg1. Returns altered arg1.	10.3.2
DE	LSUBR	Defines LAMBDA-expressions.	6.4
DF	LSUBR	Defines FLAMBDA-expressions.	6.4
DIFF	SUBR*	Returns arg1-(arg2-(arg3....-argn).	11.2.1
DIGITS	switch	Controls number of significant digits on BNUM printing. Initially 13.	4.2.1
DIVIDE	SUBR*	Returns arg1/(arg2/(arg3.../argn). Result is always a BNUM.	11.2.2
DO	LSUBR	Performs arg1 iterative evaluations of arg2 thru argn; arg1 is evaluated. Returns NIL.	8.3

ECHO	switch	Controls echo of input lines to OUTUNIT device. Initially NIL, i.e., no echo.	3.1.6 14.1.2 15.2
EFFACE	SUBR,2	Dstructively deletes arg1 from list arg2, if found. Returns arg2.	10.3.2
EOFSTAT	SUBR,1	Returns T if local input file unit arg1 is at EOI or empty, NIL if not.	14.1.2
EOLR	switch	Controls appending of CR character at end of input line. Initially T, i.e., CR is appended.	3.1.2 3.2.2 3.3
EOLW	switch	Controls appending of CR-LF at end of output line. Initially T, i.e., CR-LF is appended.	4.1.3
EQ	SUBR,2	Returns T if arg1 is the same ALISP pointer as arg2, NIL if not.	9.1 9.4
EQP	SUBR,2	Numerically compares numbers arg1 and arg2. Uses FUZZ as a comparison tolerance.	9.2
EQUAL	SUBR,2	Compares list structure.	9.3
ERR	SUBR,3	Causes an immediate USER-ER; arg1, arg2, and arg3 are messages.	12.1.3
ERRPRIN	switch	Controls printing of error messages; initially T, i.e., error messages are printed.	12.1.1
ERRSET	FSUBR,2	Evaluates arg1, returns a list of the result if no errors occurred. If an error was issued, arg2 is evaluated, and NIL is returned.	12.1.2
ESHIFT	SUBR,2	Does end-off shifting of LNUM arg1 by SNUM shift count arg2.	11.4.2
EVAL	SUBR,1	Evaluates arg1 using the McCarthy EVAL function.	6.2 6.2.3
EVLIST	SUBR,1	Evaluates each element of arg1, returns the result of evaluating the last element.	6.2.3
EXIT	SUBR,0	Exits from the ALISP system.	1.1 6.1.4 15.1

EXP	SUBR,1	Returns the exponential function of number arg1.	11.3.2
FIX	SUBR,1	Converts number arg1 to an SNUM.	11.1.2
FIXP	SUBR,1	Returns T if arg1 is an SNUM, NIL if not.	11.1.1
FLAMBDA	LSUBR	Identity function. Used as a function definition.	6.2.2 6.3 6.3.1 6.4
FLOAT	SUBR,1	Converts number arg1 to a BNUM.	11.1.2
FLOATP	SUBR,1	Returns T if its argument is a BNUM, NIL if not.	11.1.1
FNTYPE	SUBR,1	Returns the function type of arg1 as SUBR, LAMBDA, etc. arg1 can be either the function definition or its name.	6.4.1
FSUBR	switch	Type of machine-language function.	6.3 6.3.2
FSUBR*	switch	Type of machine-language function.	6.3 6.3.2
FUZZ	switch	Controls the EQP interval for BNUM's. Initially set to .2E-5.	9.2
GC	SUBR,0	Calls an immediate garbage-collect; returns NIL.	13.3
GENCHAR	switch	Controls the GENSYM atom character. Initially NIL, i.e., a default character of "G".	5.2.2
GENSYM	SUBR,0	Generates GENSYM atoms.	5.2.2
GENSYMP	SUBR,1	Predicate for GENSYM atoms; returns T if arg1 is a GENSYM atom, NIL if not.	5.2.2
GET	SUBR,2	Returns the property of indicator arg2 from the plist of nlitat arg1, if found; else returns NIL.	10.1

GETFUN	SUBR,1	Returns the function definition of arg1 if it has one, else NIL. Arg1 can be either a function definition or its name.	6.4.1 7.1
GETVAL	SUBR,1	Returns the value of literal atom arg1.	5.3.2
GO	FSUBR,1	Goes to the tag arg1 within a PROG. Evaluates its argument until it is atomic.	8.2.1 12.2.2
GREATERP	SUBR,2	Returns T if number arg1 is greater than number arg2, NIL if not.	9.2.1
HALFPRI	SUBR,1	Prints the first HPRNUM atoms of arg1. Returns arg1.	4.2.2
HPRNUM	switch	SNUM number of atoms for HALFPRI to print. Default value of NIL is 4 atoms; initially set to NIL.	4.2.2
IF	LSUBR	Conditional function.	8.1.1
ILLEGAL	switch	Value of an atom which has no value; causes a VAL-ERR from EVAL.	5.3.2 6.2.1
INBASE	switch	Holds the input base representation. Initially #12, i.e., base 10.	3.2.2
INTADD	SUBR,1	Returns the internal address of arg1 as an SNUM.	9.4
INTRFLG	switch	Controls terminal interrupt. If non-NIL, interrupts are enabled; if NIL, disabled. Initially T.	12.2.1
INUNIT	switch	Controls the local file unit used on input. Initially 0, i.e., the terminal.	3.1.1 6.1.2 14.1.1 14.1.2
LABEL	LSUBR	Identity function. Also a function definition.	6.2.2 6.3.1

LAMBDA	LSUBR	Identity function. Also a function definition.	6.2.2 6.3 6.3.1 6.4
LAST	SUBR,1	Returns the last element of list arg1.	10.2.1
LENGTH	SUBR,1	Returns the top-level length of list arg1 as an SNUM.	10.2.1
LESSP	SUBR,2	Returns T if number arg1 is less than number arg2, else NIL.	9.2.1
LISTP	SUBR,1	Returns T if arg1 is a list, NIL if not.	2.2.3
LIST	SUBR*	Strings together arg1 thru argn in a list.	10.2.2
LITP	SUBR,1	Returns T if arg1 is a nglitat, else NIL.	5.2.3
LOAD	SUBR,1	Loads the ALISP overlay arg1.	14.3
LOG	SUBR,1	Returns the natural logarithm of number arg1.	11.3.2
LOGAND	SUBR,2	Returns the logical and of arg1 and arg2, both LNUM's.	11.4.1
LOGICAL	SUBR,1	Converts number arg1 to an LNUM.	11.1.2
LOGNOT	SUBR,1	Returns the logical complement of LNUM arg1.	11.4.1
LOGOR	SUBR,2	Returns the logical inclusive or of arg1 and arg2, both LNUM's.	11.4.1
LOGP	SUBR,1	Returns T if arg1 is an LNUM, NIL if not.	11.1.1
LOGXOR	SUBR,2	Returns the logical exclusive or of arg1 and arg2, both LNUM's.	11.4.1
LSUBR	switch	Function type.	6.3 6.3.2
MAPC	SUBR,2	Applies function arg2 to successive CAR's of list arg1; returns last result.	7.2.1

MAPCAR	SUBR,2	Applies function arg2 to successive CAR's of list arg1; returns list of results.	7.2.1
MAPCON	SUBR,2	Applies function arg2 to successive CDR's of list arg1; returns a CONC of the results.	7.2.3
MAPCONC	SUBR,2	Applies function arg2 to successive CAR's of list arg1; returns a CONC of the results.	7.2.3
MAPL	SUBR,2	Applies function arg2 to successive CDR's of list arg1; returns the last result.	7.2.2
MAPLIST	SUBR,2	Applies function arg2 to successive CDR's of list arg1; returns a list of the results.	7.2.2
MAX	SUBR*	Returns the maximum of numbers arg1 thru argn.	9.2.1
MEMB	SUBR,2	Returns T if arg1 is on any level of list arg2; else NIL.	9.1
MEMBER	SUBR,2	Returns rest of list arg2 starting with arg1 if arg1 is on the top level of arg2; else NIL.	9.1
MIN	SUBR*	Returns the minimum of numbers arg1 thru argn.	9.2.1
MINUS	SUBR,1	Returns the negative of number arg1.	11.3.1
MINUSP	SUBR,1	Returns T if number arg1 is negative, NIL if not.	9.1
NIL	switch	Value of NIL is NIL; CAR and CDR of NIL are NIL.	5.2.1
NCONC	SUBR,2	Destructively merges lists arg1 and arg2 on top level.	10.3.1
NOVAL	switch	Value returned by some functions to indicate that no value exists.	5.3.2

NULL	SUBR,1	Returns T if arg1 is NIL, else NIL.	5.2.1
NUMBERP	SUBR,1	Returns T if arg1 is a number token, else NIL.	11.1.1
OBLIST	SUBR,0	Returns a copy of the internal hash buckets holding literal atoms.	3.2.2 5.1 5.1.1
ODDP	SUBR,1	Returns T if integer portion of number arg1 is odd, else NIL.	9.1
OPEN	SUBR,1 2	Opens permanent file arg1 as local input file unit number arg2. Returns arg1.	14.2.2
OR	LSUBR	Evaluates each argument until one returns a non-NIL value; else returns NIL.	8.1.2
OUTBASE	switch	Controls the output representation for numbers. Initially #12, i.e., base 10.	4.1.1 14.1.3
OUTUNIT	switch	Controls the local file unit used on output. Initially 0, i.e., the terminal.	4.1.1 6.1.2 14.1.1 14.1.3
PACK1	SUBR,1	Packs the character represented by SNUM arg1 into the output buffer. Returns arg1.	4.3.1
PACK	SUBR,1	Forms the literal atom specified by the first character of the pnames of atom elements of list arg1.	5.3.1
PARAMCP	SUBR,0	Returns the control card parameters for the ALISP job as a list.	1.1.1
PARAMFL	SUBR*	With no arguments, returns the field length parameters for ALISP a s a two element list of SNUM's; with one argument, sets the maximum field length to that argument.	13.2
PARAMGC	SUBR,0	Returns a list of garbage-collect statistics.	13.1

PARAMTL	SUBR*	With no arguments, returns a list of the time limit statistics. With one argument, sets the time limit to that argument.	12.1.4
PLIST	SUBR*	With one argument, ret urns the plist of nlitat argl. With two arguments, sets the plist of argl to arg2.	5.3.3 10.1
PLUS	SUBR,1	Returns argl+(arg2+(arg3...argn).	11.2.1
PLUSP	SUBR,1	Returns T if argl is positive, NIL if not.	9.1
PRINB	SUBR,1	Packs argl blanks into the output buffer. Returns argl.	4.3.1
PRINBEG	switch	First character position in the output buffer.	4.3 15.2
PRINEND	switch	Last character position in the output buffer.	4.1 4.1.3 4.3
PRINLEN	switch	Current character position in the output buffer.	4.3
PRINT	SUBR,1	Primary print syntaxing function. Outputs argl to the current output device. Returns argl.	4.2.1 4.2.2
PRIN1	SUBR,1	Same as PRINT, except does not do a TERPRI after it prints. Returns argl.	4.2.2
PROG	LSUBR	Program feature.	8.2.1
PROGN	LSUBR	Evaluates argl thru argn, returns the last result.	8 .2.2
PROMPT	switch	Controls the input prompt character. Initially NIL, i.e., a b̸? prompt.	3.1.3 15.2
PROP	SUBR,2	Returns the plist of nlitat argl starting from but not including the indicator arg2, if found; else NIL.	10.1
PURGE	SUBR,1	Purges permanent file argl. Returns argl.	14.2.2

PUT	SUBR,3	Adds arg3 under the indicator arg2 on litat arg1's plist.	10.1
QSETQ	LSUBR	Sets nlitat arg1 to arg2; takes an indefinite number of pairs of arguments.	5.3.2
QUOTE	FSUBR,1	Identity function, returns arg1.	3.2.2
QUOTIENT	SUBR*	Does arg1/(arg2/(arg3.../argn). Truncates the result of each divide operation to an integer.	11.2.2
RANDY	SUBR,0	Returns a pseudo-random BNUM in the open interval (0,1).	11.3.2
READ	SUBR,0	Returns the next S-expression in the input buffer.	3.2 3.2.2
READBEG	switch	First character position to start reading in the input buffer. Initially 0.	3.3
READEND	switch	Last character position in the input buffer.	3.3
READENT	SUBR,0	Does a TEREAD before a READ, returns the result of the READ.	3.2 3.2.4
READLEN	switch	Current character position in the input buffer.	3.3
READCH	SUBR,0	Returns the next character in the input buffer as a single-character atom.	3.3.1
READNB	SUBR,0	Returns the next non-blank (STATUS ≠ 2) character from the input buffer as a single-character atom.	3.3.1
READNM	SUBR,0	Returns the SNUM equivalent of the next character in the input buffer.	3.3.1
READPK	SUBR,0	Same as READCH, except it does not advance READLEN. If READLEN=READEND, returns NIL.	3.3.1
REMAINDER	SUBR,2	Returns the remainder of arg1/arg2.	11.2.2

REMOB	SUBR,1	Removes the value of atom nlitat arg1, by making it ILLEGAL.	6.4.2
REMPROP	SUBR,2	Removes indicator arg2 and associated property from the plist of nlitat arg1. Returns NIL.	10.1
RETURN	SUBR,1	Causes PROG to exit with value arg1.	8.2.1 12.2.2
REVERSE	SUBR,1	Reverses top level of list arg1.	10.2.2
REWIND	SUBR,1	Rewinds input local file unit arg1. Returns arg1.	14.2 .1
RPLACA	SUBR,2	Destructively replaces the CAR of list arg1 with arg2. Returns the altered arg1.	10.3.1
RPLACD	SUBR,2	Destructively replaces the CDR of list arg1 with arg2; returns arg2.	10.3.1
RUNTIME	FSUBR*	With no arguments, returns the RUNTIME clock as a BNUM of milliseconds. If arg1 is a BNUM, sets the RUNTIME clock to arg1. Else, evaluates arg1 and prints the evaluation time in milliseconds on OUTUNIT, and returns the result of the evaluation.	12.1.4
SAVE	SUBR,1	Saves the ALISP system as an overlay file with name arg1. Returns arg1.	14.3
SCRATCH	SUBR,1	Scratches local output file unit arg1. Returns arg1.	14.2.1
SETBIT	SUBR,2	Sets the SNUM arg2 bit of LNUM arg1.	11.5
SET	SUBR,2	Sets the value of nlitat arg1 to arg2.	5.3.2
SETQ	LSUBR	Sets the value of nlitat arg1 to the evaluation of arg2. Takes an indefinite number of pairs of arguments.	5.3.2

SIN	SUBR,1	Returns the sine function of number arg1. Arg1 should be in radians.	11.3.2
SLASHES	switch	Controls the printing of slashes on output of exotic pnames. Initially NIL, i.e., no slashes.	4.2.1
SPECIAL	SUBR*	With one argument, returns the special status of nlitat arg1 as T c NIL. With two arguments, sets tl special status of arg1 to T or NIL.	4.2.1
SQRT	SUBR,1	Returns the square root of the absolute value of number arg1.	11.3.2
STACK	SUBR,0	Returns a list of pendant function calls.	12.2.1
STATUS	SUBR*	With one argument, returns the status of the first character in nglitat arg1's pname as an SNUM from 0 to 7. With two arguments, sets the status of arg1 to SNUM arg2, returns the previous STATUS of arg1.	3.2.1
SUBR	switch	Function type.	6.3 6.3.2
SUBR*	switch	Function type.	6.3 6.3.2
SUB1	SUBR,1	Subtracts one from number arg1.	11.3.1
SYS	switch	Controls the type of supervisor in effect at top level. Initially NIL, i.e., an EVAL supervisor.	6.1.1 12.2.2
SYSIN	switch	Output device for supervisor and error processing. Initially 0, i.e., the terminal.	6.1.2 12.1.1 12.2.2 14.1.2 15.1 15.2
SYSOUT	switch	Output device for the supervisor and error processor. Initially 0, i.e., the terminal.	6.1.2 12.1.1 12.2.2 14.1.2 14.1.3 15.1 15.2

SYSPRIN	switch	Controls supervisor printing. Initially T, i.e., the supervisor prints its output.	6.1.3 12.2.2
TEREAD	SUBR,0	Empties the input buffer and reads in a new line from INUNIT. Returns NIL.	3.2.4
TERPRI	SUBR,0	Terminates the print line and dumps the output buffer to OUTUNIT. Returns NIL.	4.1.3 4.2.2
TIMES	SUBR*	Returns arg1*(arg2*(arg3...*arg n).	11.2.1
TOGBIT	SUBR,2	Complements the SNUM arg2 bit of LNUM arg1.	11.5
TRACE	FSUBR*	With no arguments, returns a list of all atoms currently being traced. With arguments, sets the tracing status of arg1 thru argn.	12.3
TRACFLG	switch	Controls tracing. If NIL, no tracing is done; if non-NIL, tracing is done. Initially T.	12.3.1 12.3.2
TSTBIT	SUBR,2	Returns T if the SNUM arg2 bit of LNUM arg1 is set, else NIL.	11.5
TTYCHAR	switch	Controls translation to upper-case on input. If NIL, no translation is done; if non-NIL, it is. Initially T.	3.1.5
UNPACK	SUBR,1	Returns a list of single-character atoms formed from the pname of non-GENSYM atom arg1.	5.3.1
UNTRACE	FSUBR*	With no arguments, turns off the tracing status of all atoms. With arguments, turns off the tracing status of arg1 thru argn.	12.3
VALUEP	SUBR,1	Returns T if litat arg1 has a value, else NIL.	5.3.2
WIPE	SUBR,1	Puts nglitat arg1 on the WIPELIST.	5.1.2

WIPELIST	SUBR,0	Returns a list of all atoms on the WIPELIST.	5.1.2
ZEROP	SUBR,1	Returns T if number arg1 is zero, else NIL.	9.1
*	switch	Holds the last evaluation printed by the supervisor, either the top-level or BREAK supervisor.	6.1.3 12.2.2
;	macro	Sets off comments on a line.	3.2.2
$	macro	Causes immediate evaluation of the next S-expression in the input stream.	3.2.2
' and "	macro	Returns a quoted list of the next S-expression in the input stream.	3.2. 2

Appendix D

Error Messages

This appendix is a quick reference to most of the errors given by the ALISP system; it enables the keen-eyed user to decipher the error messages issued. More detailed information on the conditions which will cause an error will be found with individual function descriptions in the manual.

An ALISP error will normally print as three asterisks, followed by a group of characters ending in "ERR". Up to four different pieces of information can be included in an error message; these help to pinpoint the source of the error. For intricate programs, a backtracing of function calls and variable bindings (section I.12.1.1) can be a useful supplement to the error message.

The error message format is as follows:

```
*** xxx-ERR FROM y
OFFENDING VAL (or ARG) = z
message
```

For any particular error, some parts of this format may be omitted.

The most important part of the error message is the parameter y. Usually, y is the function name where the error occurred. z and message specify subsidiary conditions or explanations of the error.

A table of errors and error messages follows. They are ordered by type of error (xxx-ERR) and sub-ordered by message.

VAL-ERR Value error.

Only the single parameter y is given. y is an nlitat which did not have a value when EVAL requested one. Frequently occurs when expressions of the form:

(CONS FOO BAR)

are evaluated; here CONS evaluates its arguments, and if FOO and BAR do not have values, the error will be issued.

FUN-ERR Function error.

If y is given, it is the atom name which caused the error. If not, the error was caused by a list as the first element of an evaluated form. In general, a FUN-ERR means that EVAL (or APPLY) failed to find a valid function definition for y. If z is given, this is usually the value of y which caused the problem. Re-read the section on the interpreter (I.6) if you cannot figure out why a FUN-ERR occurred.

SYN-ERR Syntax error.

y is usually READ. This error is issued when READ attmpts to parse an incorrectly syntaxed string. Section I.3.2.2 describes the READ syntax.

BAD NUMBER SYNTAX

An incorrectly formatted numeric string was in the input stream. z is the character position in the input line at which the error occurred.

PNAME TOO LARGE

Pnames of more than 64 characters are illegal.

INITIAL RIGHT PAR

A right parenthesis was the first non-blank character READ found in the input stream when attempting to form an S-expression. z is the character position of the parenthesis in the input line.

MISSING RIGHT PAR

A right parenthesis was missing after a comma when forming a dotted S-expression. For example, the line:

(FOO , BAR MOO)

will cause this error, because READ expects a right parenthesis after BAR. z is the character position in the input line where the parenthesis should have been.

ARG-ERR

Argument error.

This is the most common type of error. Either the interpreter decided that the arguments to a function were not correct, or the function rejected them after they were passed by the interpreter. y is the name of the function, and z is usually the argument which caused the problem.

WRONG NO. OF ARGS

The wrong number of arguments was passed to a function. For lambda-expressions, z is the variable list.

VAR NOT LIT. ATOM

A variable in a lambda-expression was not an nlitat z is the offending variable.

BAD FN. FORM

A function y was used in an incorrectly formatted form, for instance, the form:

(PROG)

would give this error when evaluated, since the variable list and body of the PROG are absent. This error is called by LSUBR type functions.

NO PROG EXECUTING

Called by GO or RETURN when used outside the scope of a PROG evaluation.

CAR (or CDR) OF UNNIL ATOM

CAR or CDR was given an atomic, non-NIL argument. z is the argument.

ARG NOT SNUM

An SNUM was expected as an argument, for example, as the second argument to the DO function. z is the criminal argument. This message is usually given by the functions which consider lists as numbered vectors of elements, e.g., ADDEL, DELETEL, ARGN, etc.

LIST TOO SHORT

A list given as an argument did not contain enough elements. The element manipulation functions such as ADDEL and DELETEL issue this error. z is the faulty list.

ATOMIC ARG
ARG NOT LITAT
ARG NOT ATOM
The wrong type of argument was supplied to a function. z is the offending argument. Valid argument types are detailed in individual function descriptions in this manual.

NUM-ERR Numeric error.
The input and output buffer pointers, as well as the switches INUNIT, OUTUNIT, INBASE, OUTBASE, DIGITS, and Complain when used if they are set to non-numeric or out of range values. y is the mis-set switch; z is the faulty value.
The arithmetic functions also complain with this error. In this case, y is the name of the arithmetic function which issued the error. The following messages apply to arithmetic errors.

TOO FEW ARGS
A dyadic function was given one or no arguments.

NON-NUMERIC ARG
An argument to an arithmetic function was not an ALISP number type. z is the argument at fault.

RESULT OUT OF RANGE
Called by FIX or LOGICAL when attempting to form an SNUM or LNUM from too large a number; here z is the number.
Also called by any arithmetic function which generates a number out of floating-point range, i.e., on division by zero.

GC-ERR Garbage-collect error.
Called when the GC routine fails to free up enough space in one of the storage areas, and the field length maximum has been reached. See section I.13.

FIL-ERR File error.
Called by the read functions on certain errors when reading from a local file; y is then READ. Also called by the permanent file manipulating functions; y is then the function.

LINE TOO LONG
A line longer than 150 characters was inputted

from a local file. z is the local input file unit number.

READ PAST EOI
An attempt was made to read from an empty input local file, or one which had reached the end of information. z is the file unit number.

WRONG VERSION NO.
An attempt was made to load an ALISP overlay file created by an ALISP system with a different version number. z is the overlay name.

FL TOO SMALL
An attempt was made to load an overlay file which required an execution field length larger than the current maximum field length.

BAD KRONOS NAME
A permanent file name, user number, or file password was not in acceptable KRONOS format; see section I.14.2.2. z is the name given.

BAD UNIT NO.
An attempt was made to OPEN or CLOSE an illegal local file unit number; z is the unit number.

BAD ACCESS
The user attempted to access an alternate user number for which he was not properly permitted; see section I.14.2.2. z is the offending access.

PF NOT FOUND
Issued by OPEN or PURGE if a permanent file cannot be found. z is the file name.

FILE TOO BIG
STORAGE FULL
TOO MANY PFS
These errors are issued by CLOSE if the user overflows his catalog limits in attempting to save a file. z is the file name.

BAD PARITY
A file operation could not be verified. Re-try the operation.

USER-ER User definable error.
This error is issued by some of the ALISP filing system functions in section II.1, with an explanatory message.

HALT

ALISP system error.

Save as much of your output as possible, and bring it to the source at the end of the introduction. Note that a "HALT FROM INTRFLG" is <u>not</u> a system error, but an error exit used by the interrupt facility.

D-6