TEK PROGRAMMERS REFERENCE

Part No. 070-5607-00
Product Group 18

4404P30 LISP

Tektronix
COMMITTED TO EXCELLENCE

TEK PROGRAMMERS REFERENCE

Part No. 070-5607-00
Product Group 18

4404P30
LISP

Please Check for
CHANGE INFORMATION
at the Rear of This Manual

First Printing JAN 1985

Tektronix®
COMMITTED TO EXCELLENCE

This revision incorporates minor corrections and changes to the University of California text, and describes the particular version of FRANZ LISP implemented by Tektronix.

MANUAL REVISION STATUS

PRODUCT: 4400P30 Franz Lisp Programming Language

This manual supports the following versions of this product: Version 41

REV DATE	DESCRIPTION
JAN 1985	Original Issue

Contents

Part I

1. FRANZ LISP
   Introduction to FRANZ LISP, details of data types, and description of notation
2. Data Structure Access
   Functions for the creation, destruction and manipulation of lisp data objects.
3. Arithmetic Functions
   Functions to perform arithmetic operations.
4. Special Functions
   Functions for altering flow of control. Functions for mapping other functions over lists.
5. I/O Functions
   Functions for reading and writing from ports. Functions for the modification of the reader's syntax.
6. System Functions
   Functions for storage management, debugging, and for the reading and setting of global Lisp status variables. Functions for doing operating system-specific tasks such as process control.

Part II

7. The Reader
   A description of the syntax codes used by the reader. An explanation of character macros.
8. Functions, Fclosures, and Macros
   A description of various types of functional objects. An example of the use of foreign functions.
9. Arrays and Vectors
   A detailed description of the parts of an array and of Maclisp compatible arrays.
10. Exception Handling
    A description of the error handling sequence and of autoloading.

Part III

11. The Lister Trace Package
A description of a very useful debugging aid.
12. Liszt, the Lisp Compiler
A description of the operation of the compiler and hints for making functions compilable.
13.The Top Level
A description of FRANZ LISP's top level which includes access to debugging tools, a history mechanism, and single stepper.
14 Miscellaneous Topics
Keyword arguments, hash tables, multiple values, the sharp sign macro, and floating point vector functions.
15 The Lisp Stepper and Fixit
Programs which permits you to single-step through a Lisp program, and also examine and modify the evaluation stack: fix bugs on the fly.
16 Lisp Editor
A structure editor for interactive modification of FRANZ LISP code.

Part IV

Appendix A - Function Index
Appendix B - List of Special Symbols
Appendix C - The Garbage Collector

FRANZ LISP INSTALLATION INSTRUCTIONS

INSTALLATION

Perform the following procedure to install and verify your Franz Lisp files.

1. Log in as the system manager by typing:

   login system

   at the system prompt "++".

2. Make sure that you are in the root directory by typing:

   chd /

3. Invoke the *restore* utility to copy the Franz Lisp system from the floppy diskettes to system disk. At the system prompt, type:

   restore +l

4. Insert the distribution floppy diskettes, in sequence, as the *restore* utility prompts you for them.

5. After you have finished with the *restore* utility, you should run *diskrepair* to verify that the system disk structure is correct. At the system prompt, type:

   diskrepair /dev/disk

You are now finished with the installation of Franz Lisp.

FRANZ LISP FILES

The following files are distributed with Franz Lisp version 41.10.

LISP EXECUTABLE FILES

The followings files are found in */bin*:

lisp	Lisp interpreter.
liszt	Lisp compiler.
lxref	Lisp cross reference program.

LISP HELP FILES

Three help files are accessible through the 4404 *help* command. These files are found in */gen/help*:

lisp
liszt
lxref

LISP LIBRARY FILES

Files ending in the extension *.l* are human-readable Lisp source code. Files ending in the extension *.o* are compiled Lisp object code. The following files are found in */lisp/lib*:

System Files

arun	Used to generate autorun files.
as	Assembler for *liszt* files.
array.l, array.o	Array package. Loaded into the standard Lisp interpreter.
autoload.l, autoload.o	Manages autoload of functions. Loaded into the standard Lisp interpreter.
buildlisp.l, buildlisp.o	Used to build the Lisp system from the C kernel.
charmac.l, charmac.o	Backquote and sharp sign macros. Loaded into the standard Lisp interpreter.
cmuedit.l, cmuedit.o	Code for an interactive structure editor. Loaded when edit functions are called.
common0.l, common0.o *common1.l, common1.o* *common2.l, common2.o* *common3.l, common3.o* *common4.l, common4.o*	Most Lisp-coded Lisp functions are in common files. These are loaded into the standard Lisp interpreter.
describe.l, describe.o	Functions to describe any Lisp object, including *flavors*.
filep.l, filep.o	File package that interfaces with the *tpl* top level. Loaded into the standard Lisp interpreter.
fix.l, fix.o	Fix package that is autoloaded when the function debug is invoked.
format.l, format.o	String formatting, compatible with Zetalisp.
machacks.l, machacks.o	Maclisp compatibility package. Autoloaded when the *+m* option is specified for *liszt*.
macros.l, macros.o	Common macros for Franz Lisp. Loaded into the standard Lisp interpreter.
pp.l, pp.o	Pretty printer. Loaded when the function *pp* is invoked.
prof.l, prof.o	Dynamic profiler for Lisp.

record.l, record.o	Record package.
step.l, step.o	Stepping package. Loaded when function step invoked.
syntax.l, syntax.o	Contains setsyntax function. Loaded into the standard Lisp interpreter.
tpl.l, tpl.o	Franz Lisp top level. Loaded into the standard Lisp interpreter.
trace.l, trace.o	Trace package. Loaded when trace function invoked.
vector.l, vector.o	Vector handling functions. Loaded into the standard Lisp interpreter.
version.l, version.o	Franz Lisp version info. Loaded into the standard Lisp interpreter.

Additional Files

These files are distributed as a service to Lisp users. They are *not* supported by Tektronix.

cmuenv.l, cmuenv.o	Loads *cmumacs*, *cmufncs*, *cmutop*, and *cmufile* for a cmu environment.
cmufncs.l, cmufncs.o	Functions required by the cmu macros.
cmumacs.l, cmumacs.o	Macros required for compiling other cmu files, also useful at runtime.
cmutpl.l, cmutpl.o	Cmu top level.
flavorm.l, flavorm.o	Support macros needed by the flavor system. (Copyright 1983 by Massachusetts Institute of Technology.)
flavors.l, flavors.o	Flavor system, object definition and creation. (Copyright 1982 by Massachusetts Institute of Technology.)
lmhacks.l, lmhacks.o	Miscellaneous functions compatible with Zetalisp. (Copyright 1982 by Massachusetts Institute of Technology.)
loop.l, loop.o	Loop macro. (Copyright 1980, 1981 by Massachusetts Institute of Technology.)
struct.l, struct.o	Structure package. (Copyright 1980 by Massachusetts Institute of Technology.)
structini.l	Macros necessary for compiling the structure package.
ucido.l, ucido.o	UCI Lisp do loop.
ucifnc.l, ucifnc.o	UCI Lisp compatibility package.
vanilla.l, vanilla.o	Definition of vanilla flavors and methods. (Copyright 1982 by Massachusetts Institute of Technology.)

CHAPTER 1

FRANZ LISP

1.1. FRANZ LISP was created as a tool to further research in symbolic and algebraic manipulation, artificial intelligence, and programming languages at the University of California at Berkeley. Its roots are in a PDP-11 Lisp system, which originally came from Harvard University. As it grew, it adopted features of Maclisp and Lisp Machine Lisp. Substantial compatibility with other Lisp dialects (Interlisp, UCILisp, CMULisp) is achieved by means of support packages and compiler switches. The heart of FRANZ LISP is written almost entirely in the programming language C. Of course, it has been greatly extended by additions written in Lisp. A small part is written in the assembly language for the various host machines. Because FRANZ LISP is written in C, it is relatively portable and thus is in use on a wide variety of machines.

FRANZ LISP is capable of running large lisp programs in a timesharing environment, has facilities for arrays and user-defined structures, has a user-controlled reader with character and word macro capabilities, and can interact directly with compiled Lisp, C, Fortran, and Pascal code.

This document is a reference manual for the FRANZ LISP system for the Tektronix 4404 implementation. It is not a Lisp primer or introduction to the language. A recommended text for learning Lisp, with specific reference to FRANZ LISP is *Lispcraft* by Robert Wilensky, published by W. W. Norton (1984).

This document is divided into four Movements. The first Movement describes the language of FRANZ LISP precisely and completely. The second Movement describes the reader, function types, arrays, and exception handling. The third Movement describes several large support packages, namely, the trace package, compiler, fixit and stepping package, written to help you use FRANZ LISP Finally the fourth movement contains an index into the other movements. The rest of this chapter examines the data types of FRANZ LISP. The conventions used in the description of the FRANZ LISP functions are given in §1.3 -- it is very important that these conventions are understood.

1.2. Data Types FRANZ LISP has fourteen data types. This section looks in detail at each type, and, if a type is divisible, the insides are examined. There is a Lisp function *type* that returns the type name of a lisp object. This is the official FRANZ LISP name for that type and this name and this name only is used in the manual to avoid confusing you. The types are listed in terms of importance rather than alphabetically.

1.2.0. lispval This is the name used to describe any Lisp object. The function *type* never returns 'lispval'.

1.2.1. symbol This object corresponds to a variable in most other programming languages. It may have a value or may be 'unbound'. A symbol may be *lambda bound* meaning that its current value is stored away somewhere and the symbol is given a new value for the duration of a certain context. When the Lisp processor leaves that context, the symbol's current value is thrown away and its old value is restored.

A symbol may also have a *function binding*. This function binding is static; it cannot be lambda bound. Whenever the symbol is used in the functional position of a Lisp expression the function binding of the symbol is examined. See Chapter 4 for more details on evaluation.

A symbol may also have a *property list*, another static data structure. The property list consists of a list of an even number of elements, considered to be grouped as pairs. The first element of the pair is the *indicator*, the second, the *value* of that indicator.

Each symbol has a print name *(pname)*, which is how this symbol is accessed from input and referred to on (printed) output.

A symbol also has a hashlink used to link symbols together in the oblist. (This field is inaccessible to you.)

Symbols are created by the reader and by the functions *concat*, *maknam*, and their derivatives. Most symbols live on FRANZ LISP's sole *oblist*, and therefore two symbols with the same print name are usually the exact same object; they are *eq*. Symbols that are not on the oblist are said to be *uninterned*. The function *maknam* creates uninterned symbols while *concat* creates *interned* ones.

Subpart name	Get value	Set value	Type
value	eval	set setq	lispval
property list	plist get	setplist putprop defprop	list or nil
function binding	getd	putd def	array, binary, list or nil
print name	get_pname		string
hash link			

1.2.2. list A list cell has two parts, called the car and cdr. List cells are created by the function *cons*.

Subpart name	Get value	Set value	Type
car	car	rplaca	lispval
cdr	cdr	rplacd	lispval

1.2.3. binary This type acts as a function header for machine coded functions. It has two parts: a pointer to the start of the function and a symbol whose print name describes the argument *discipline*. The discipline (if *lambda*, *macro*, or *nlambda*) determines whether the arguments to this function are evaluated by the caller before this function is called. If the discipline is a string (specifically "*subroutine*", "*function*", "*integer-function*", "*real-function*", "*c-function*", "*double-c-function*", or "*vector-c-function*") then this function is a foreign subroutine or function. (See §8.5 for more details on this.) Although the type of the *entry* field of a binary type object is usually **string** or **other**, the object pointed to is actually a sequence of machine instructions.

Objects of type binary are created by *mfunction, cfasl,* and *getaddress.*

Subpart name	Get value	Set value	Type
entry	getentry		string or fixnum
discipline	getdisc	putdisc	symbol or fixnum

1.2.4. fixnum A fixnum is an integer constant in the range -2^{31} to $2^{31}-1$. Small fixnums (-1024 to 1023) are stored in a special table so they needn't be allocated each time one is needed. In principle, the range for fixnums is machine dependent, although all current implementations for FRANZ LISP have this range.

1.2.5. flonum A flonum is a double precision real number.

1.2.6. bignum A bignum is an integer of potentially unbounded size. When integer arithmetic exceeds the limits of fixnums mentioned above, the calculation is automatically done with bignums. If calculation with bignums gives a result that can be represented as a fixnum, then the fixnum representation is used[†]. This contraction is known as *integer normalization*. Many Lisp functions assume that integers are normalized. Bignums are composed of a sequence of **list** cells and a cell known as an **sdot.** You should consider a **bignum** structure indivisible and use functions such as *haipart* and *bignum-leftshift* to extract parts of it.

1.2.7. string A string is a null terminated sequence of characters. Most functions of symbols that operate on the symbol's print name also work on strings. The default reader syntax is set so that a sequence of characters surrounded by double quotes is a string.

[†]The current algorithms for integer arithmetic operations return (in certain cases) a result between $\pm 2^{30}$ and 2^{31} as a bignum although this could be represented as a fixnum.

1.2.8. port A port is a structure that the system I/O routines can reference to transfer data between the Lisp system and external media. Unlike other Lisp objects there are a very limited number of ports (20). Ports are allocated by *infile* and *outfile* and deallocated by *close* and *resetio*. The *print* function prints a port as a percent sign followed by the name of the file it is connected to (if the port was opened by *fileopen, infile, or outfile*). During initialization, FRANZ LISP binds the symbol **piport** to a port attached to the standard input stream. This port prints as %$stdin. There are ports connected to the standard output and error streams, which print as %$stdout and %$stderr. This is discussed in more detail at the beginning of Chapter 5.

1.2.9. vector Vectors are indexed sequences of data. They can be used to implement a notion of user-defined types via their associated property list. They make **hunks** (see below) logically unnecessary, although hunks are very efficiently garbage-collected. There is a second kind of vector, called an immediate-vector, that stores binary data. The name that the function *type* returns for immediate-vectors is **vectori**. For example, immediate-vectors can be used to implement strings and block-flonum arrays. Vectors are discussed in chapter 9. The functions *new-vector* and *vector* can be used to create vectors.

Subpart name	Get value	Set value	Type
datum[*i*]	vref	vset	lispval
property	vprop	vsetprop vputprop	lispval
size	vsize	–	fixnum

1.2.10. array Arrays are rather complicated types and are fully described in Chapter 9. An array consists of a block of contiguous data, a function to access that data, and auxiliary fields for use by the accessing function. Since an array's accessing function is created by you, you can create the array to have any form you choose (e.g. n-dimensional, triangular, or hash table).

Arrays are created by the function *marray*.

Subpart name	Get value	Set value	Type
access function	getaccess	putaccess	binary, list or symbol
auxiliary	getaux	putaux	lispval
data	arrayref	replace set	block of contiguous lispval
length	getlength	putlength	fixnum
delta	getdelta	putdelta	fixnum

1.2.11. value A value cell contains a pointer to a lispval. This type is used mainly by arrays of general lisp objects. Value cells are created with the *ptr* function. A value cell containing a pointer to the symbol 'foo' is printed as '(ptr to)foo'.

1.2.12. hunk A hunk is a vector of from 1 to 128 lispvals. Once a hunk is created (by *hunk* or *makhunk*) it cannot grow or shrink. The access time for an element of a hunk is slower than a list cell element but faster than an array. Hunks are really only allocated in sizes that are powers of two, but can appear to you to be any size in the 1 to 128 range. You must realize that *(not (atom 'lispval))* returns true if *lispval* is a hunk. Most lisp systems do not have a direct test for a list cell, and, instead, use the above test and assume that a true result means *lispval* is a list cell. In FRANZ LISP, you can use *dtpr* to check for a list cell. Although hunks are not list cells, you can still access the first two hunk elements with *cdr* and *car*, and you can access any hunk element with *cxr*†. You can set the value of the first two elements of a hunk with *rplacd* and *rplaca* and you can set the value of any element of the hunk with *rplacx*. A hunk is printed by printing its contents surrounded by { and }. However, a hunk cannot be read in this way in the standard lisp system. It is easy to write a reader macro to do this if desired.

1.2.13. other Occasionally, you can obtain a pointer to storage not allocated by the lisp system. One example of this is the entry field of those FRANZ LISP functions written in C. Such objects are classified as of type **other**. Foreign functions, which call malloc to allocate their own space, may also inadvertently create such objects. The garbage collector ignores such objects.

1.3. Documentation The conventions used in the following chapters are designed to give a great deal of information in a brief space. The first line of a function description contains the function name in **bold face** and then lists the arguments, if there are any. The arguments all have names that begin with a letter or letters and an underscore. The letter or letters give the allowable type or types for that argument according to this table.

†In a hunk, the function *cdr* references the first element and *car* the second.

Letter	Allowable type(s)
g	any type
s	symbol (although nil may not be allowed)
t	string
l	list (although nil may be allowed)
n	number (fixnum, flonum, bignum)
i	integer (fixnum, bignum)
x	fixnum
b	bignum
f	flonum
u	function type (either binary or lambda body)
y	binary
v	vector
V	vectori
a	array
e	value
p	port (or nil)
h	hunk

In the first line of a function description, those arguments preceded by a quote mark are evaluated (usually before the function is called). The quoting convention is used to give a name to the result of evaluating the argument and to describe the allowable types. If an argument is not quoted, it does not mean that that argument is not evaluated, but rather that if it is evaluated, the time at which it is evaluated is specifically mentioned in the function description. Optional arguments are surrounded by square brackets. An ellipsis (...) means zero or more occurrences of an argument of the directly preceding type.

CHAPTER 2

Data Structure Access

The following functions allow you to create and manipulate the various types of lisp data structures. Refer to §1.2 for the details of the data structures known to FRANZ LISP.

2.1. Lists

The following functions exist for the creation and manipulation of lists. Lists are composed of a linked list of objects. Various authors call these either 'list cells', 'cons cells' or 'dtpr cells'. Lists are normally terminated with the special symbol **nil**. **nil** is both a symbol and a representation for the empty list ().

2.1.1. list creation

(cons 'g_arg1 'g_arg2**)**

RETURNS: A new list cell whose car is g_arg1 and whose cdr is g_arg2.

(xcons 'g_arg1 'g_arg2**)**

EQUIVALENT TO: *(cons 'g_arg2 'g_arg1)*

(ncons 'g_arg**)**

EQUIVALENT TO: *(cons 'g_arg nil)*

(list ['g_arg1 ...]**)**

RETURNS: A list whose elements are the g_arg*i*

(append 'l_arg1 'l_arg2**)**

RETURNS: A list containing the elements of l_arg1 followed by l_arg2.

NOTE: To generate the result, the top level list cells of l_arg1 are duplicated and the cdr of the last list cell is set to point to l_arg2. Thus this is an expensive operation if l_arg1 is large. See the descriptions of *nconc* and *tconc* for cheaper ways of doing the *append* if the list l_arg1 can be altered.

(appendl 'l_arg1 'g_arg2)

RETURNS: A list like l_arg1 with g_arg2 as the last element.

NOTE: This is equivalent to (append 'l_arg1 (list 'g_arg2)).

```
; A common mistake is using append to add one element to the end of a list
-> (append '(a b c d) 'e)
(a b c d . e)
; The user intended to say:
-> (append '(a b c d) '(e))
(a b c d e)
; better is appendl
-> (appendl '(a b c d) 'e)
(a b c d e)
```

(quote! [g_qform_i] ...[! 'g_eform_i] ... [!! 'l_form_i] ...)

RETURNS: The list resulting from the splicing and insertion process described below.

NOTE: *quote!* is the complement of the *list* function. *list* forms a list by evaluating each form in the argument list; evaluation is suppressed if the form is *quote*ed. In *quote!*, each form is implicitly *quote*ed. To be evaluated, a form must be preceded by one of the evaluate operations ! or !!. ! g_eform evaluates g_form and the value is inserted in the place of the call; !! l_form evaluates l_form and the value is spliced into the place of the call.

'Splicing in' means that the parentheses surrounding the list are removed as the example below shows. Use of the evaluate operators can occur at any level in a form argument.

Another way to get the effect of the *quote!* function is to use the backquote character macro (see § 8.3.3).

```
(quote! cons ! (cons 1 2) 3) = (cons (1 . 2) 3)
(quote! 1 !! (list 2 3 4) 5) = (1 2 3 4 5)
(setq quoted 'evaled)(quote! ! ((I am  ! quoted))) = ((I am evaled))
(quote! try ! '(this ! one)) = (try (this ! one))
```

(bignum-to-list 'b_arg)

RETURNS: A list of the fixnums which are used to represent the bignum.

NOTE: The inverse of this function is *list-to-bignum.*

(list-to-bignum 'l_ints)

WHERE: l_ints is a list of fixnums.

RETURNS: A bignum constructed of the given fixnums.

NOTE: The inverse of this function is *bignum-to-list.*

2.1.2. list predicates

(dtpr 'g_arg)

RETURNS: t if g_arg is a list cell.

NOTE: (dtpr '()) is nil. The name **dtpr** is a contraction for "dotted pair".

(listp 'g_arg)

RETURNS: t if g_arg is a list object or nil.

(tailp 'l_x 'l_y)

RETURNS: l_x, if a list cell *eq* to l_x is found by *cdr*ing down l_y zero or more times, nil otherwise.

```
-> (setq x '(a b c d) y (cddr x))
(c d)
-> (and (dtpr x) (listp x))   ; x and y are dtprs and lists
t
-> (dtpr '())                 ; () is the same as nil and is not a dtpr
nil
-> (listp '())                ; however it is a list
t
-> (tailp y x)
(c d)
```

(length 'l_arg)

RETURNS: The number of elements in the top level of list l_arg.

2.1.3. list accessing

(car 'l_arg)
(cdr 'l_arg)

RETURNS: The appropriate part of *cons* cell. (*car* (*cons* x y)) is always x, (*cdr* (*cons* x y)) is always y. In FRANZ LISP, the cdr portion is located first in memory. This is hardly noticeable, and we mention it primarily as a curiosity.

(c..r 'lh_arg)

WHERE: The .. represents any positive number of **a**'s and **d**'s.

RETURNS: The result of accessing the list structure in the way determined by the function name. The **a**'s and **d**'s are read from right to left, a **d** directing the access down the cdr part of the list cell and an **a** down the car part.

NOTE: lh_arg may also be nil, and it is guaranteed that the car and cdr of nil is nil. If lh_arg is a hunk, then *(car 'lh_arg)* is the same as *(cxr 1 'lh_arg)* and *(cdr 'lh_arg)* is the same as *(cxr 0 'lh_arg)*.
It is generally hard to read and understand the context of functions with large strings of **a**'s and **d**'s, but these functions are supported by rapid accessing and open-compiling (see Chapter 12).

(nth 'x_index 'l_list)

RETURNS: The nth element of l_list, assuming zero-based index. Thus (nth 0 l_list) is the same as (car l_list). *nth* is both a function and a compiler macro so that more efficient code might be generated than for *nthelem* (described below).

NOTE: If x_arg1 is non-positive or greater than the length of the list, nil is returned.

(nthcdr 'x_index 'l_list)

RETURNS: The result of *cdr*ing down the list l_list x_index times.

NOTE: If x_index is less than 0, then *(cons nil 'l_list)* is returned.

(nthelem 'x_arg1 'l_arg2)

RETURNS: The x_arg1'*st* element of the list l_arg2.

NOTE: This somewhat non-standard name of this function comes from the PDP-11 Lisp system.

(last 'l_arg)

RETURNS: The last list cell in the list l_arg.

EXAMPLE: *last* does NOT return the last element of a list!
(last '(a b)) = (b)

(ldiff 'l_x 'l_y)

RETURNS: A list of all elements in l_x but not in l_y , i.e., the list difference of l_x and l_y.

NOTE: l_y must be a tail of l_x, i.e., *eq* to the result of applying some number of *cdr*'s to l_x. Note that the value of *ldiff* is always a new list structure unless l_y is nil, in which case *(ldiff l_x nil)* is l_x itself. If l_y is not a tail of l_x, *ldiff* generates an error.

EXAMPLE: *(ldiff 'l_x (member 'g_foo 'l_x))* gives all elements in l_x up to the first g_foo.

2.1.4. list manipulation

(rplaca 'lh_arg1 'g_arg2)

RETURNS: The modified lh_arg1.

SIDE EFFECT: The car of lh_arg1 is set to g_arg2. If lh_arg1 is a hunk then the second element of the hunk is set to g_arg2.

(rplacd 'lh_arg1 'g_arg2)

RETURNS: The modified lh_arg1.

SIDE EFFECT: The cdr of lh_arg2 is set to g_arg2. If lh_arg1 is a hunk then the first element of the hunk is set to g_arg2.

(attach 'g_x 'l_l)

RETURNS: l_l whose *car* is now g_x, whose *cadr* is the original *(car l_l)*, and whose *cddr* is the original *(cdr l_l)*.

NOTE: What happens is that g_x is added to the beginning of list l_l yet maintaining the same list cell at the beginning of the list.

(delete 'g_val 'l_list ['x_count])

RETURNS: The result of splicing g_val from the top level of l_list no more than x_count times.

NOTE: x_count defaults to a very large number, thus if x_count is not given, all occurrences of g_val are removed from the top level of l_list. g_val is compared with successive *car*'s of l_list using the function *equal*

SIDE EFFECT: l_list is modified using rplacd, no new list cells are used.

(delq 'g_val 'l_list ['x_count])
(dremove 'g_val 'l_list ['x_count])

RETURNS: The result of splicing g_val from the top level of l_list no more than x_count times.

NOTE: *delq* (and *dremove*) are the same as *delete* except that *eq* is used for comparison instead of *equal*

```
; note that you should use the value returned by delete or delq
; and not assume that g_val will always show the deletions.
; For example

-> (setq test '(a b c a d e))
(a b c a d e)
-> (delete 'a test)
(b c d e)       ; the value returned is what we would expect
-> test
(a b c d e)     ; but test still has the first a in the list!
```

(remq 'g_x 'l_l ['x_count]**)**
(remove 'g_x 'l_l**)**

RETURNS: A *copy* of l_l with all top level elements *equal* to g_x removed. *remq* uses *eq* instead of *equal* for comparisons.

NOTE: remove does not modify its arguments like *delete* and *delq* do.

(insert 'g_object 'l_list 'u_comparefn 'g_nodups**)**

RETURNS: A list consisting of l_list with g_object destructively inserted in a place determined by the ordering function u_comparefn.

NOTE: *(comparefn 'g_x 'g_y)* should return something non-nil, if g_x can precede g_y in sorted order; nil, if g_y must precede g_x. If u_comparefn is nil, alphabetical order is used. If g_nodups is non-nil, an element is not inserted, if an equal element is already in the list. *insert* does a binary search to determine where to insert the new element.

(merge 'l_data1 'l_data2 'u_comparefn**)**

RETURNS: The merged list of the two input sorted lists l_data1 and l_data1 using binary comparison function u_comparefn.

NOTE: *(comparefn 'g_x 'g_y)* should return something non-nil, if g_x can precede g_y in sorted order; nil, if g_y must precede g_x. If u_comparefn is nil, alphabetical order is used. u_comparefn should be thought of as "less than or equal to". *merge* changes both of its data arguments.

(subst 'g_x 'g_y 'l_s**)**
(dsubst 'g_x 'g_y 'l_s**)**

RETURNS: The result of substituting g_x for all *equal* occurrences of g_y at all levels in l_s.

NOTE: If g_y is a symbol, *eq* is used for comparisons. The function *subst* does not modify l_s but the function *dsubst* (destructive substitution) does.

(lsubst 'l_x 'g_y 'l_s**)**

RETURNS: A copy of l_s with l_x spliced in for every occurrence of g_y at all levels. Splicing in means that the parentheses surrounding the list l_x are removed as the example below shows.

```
-> (subst '(a b c) 'x '(x y z (x y z) (x y z)))
((a b c) y z ((a b c) y z) ((a b c) y z))
-> (lsubst '(a b c) 'x '(x y z (x y z) (x y z)))
(a b c y z (a b c y z) (a b c y z))
```

(subpair 'l_old 'l_new 'l_expr**)**

WHERE: There are the same number of elements in l_old as l_new.

RETURNS: The list l_expr with all occurrences of an object in l_old replaced by the corresponding one in l_new. When a substitution is made, a copy of the value to substitute in is not made.

EXAMPLE: *(subpair '(a c)' (x y) '(a b c d)) = (x b y d)*

(nconc 'l_arg1 'l_arg2 ['l_arg3 ...]**)**

RETURNS: A list consisting of the elements of l_arg1 followed by the elements of l_arg2 followed by l_arg3 and so on.

NOTE: The *cdr* of the last list cell of l_arg*i* is changed to point to l_arg*i+1*.

```
; nconc is faster than append because it doesn't allocate new list cells.
-> (setq lis1 '(a b c))
(a b c)
-> (setq lis2 '(d e f))
(d e f)
-> (append lis1 lis2)
(a b c d e f)
-> lis1
(a b c)      ; note that lis1 has not been changed by append
-> (nconc lis1 lis2)
(a b c d e f) ; nconc returns the same value as append
-> lis1
(a b c d e f) ; but in doing so alters lis1
```

(reverse 'l_arg)
(nreverse 'l_arg)

RETURNS: The list l_arg with the elements at the top level in reverse order.

NOTE: The function *nreverse* does the reversal in place; that is, the list structure is modified.

(nreconc 'l_arg 'g_arg)

EQUIVALENT TO: *(nconc (nreverse 'l_arg) 'g_arg)*

2.2. Predicates

The following functions test for properties of data objects. When the result of the test is either 'false' or 'true', then **nil** is returned for 'false' and something other than **nil** (often **t**) is returned for 'true'.

(arrayp 'g_arg)

RETURNS: t if g_arg is of type array.

(atom 'g_arg)

RETURNS: t if g_arg is not a list or hunk object.

NOTE: *(atom '())* returns t.

(bcdp 'g_arg)

RETURNS: t if g_arg is a data object of type binary.

NOTE: This function name is a throwback to the PDP-11 Lisp system. It stands for binary code predicate.

(bigp 'g_arg)

RETURNS: t if g_arg is a bignum.

(dtpr 'g_arg)

RETURNS: t if g_arg is a list cell.

NOTE: (dtpr '()) is nil.

(hunkp 'g_arg)

RETURNS: t if g_arg is a hunk.

(listp 'g_arg)

RETURNS: t if g_arg is a list object or nil.

(stringp 'g_arg)

RETURNS: t if g_arg is a string.

(symbolp 'g_arg)

RETURNS: t if g_arg is a symbol.

(valuep 'g_arg)

RETURNS: t if g_arg is a value cell

(vectorp 'v_vector)

RETURNS: t if the argument is a vector.

(vectorip 'v_vector)

RETURNS: t if the argument is an immediate-vector.

(type 'g_arg)
(typep 'g_arg)

RETURNS: A symbol whose pname describes the type of g_arg.

(signp s_test 'g_val)

RETURNS: t if g_val is a number and the given test s_test on g_val returns true.

NOTE: The fact that *signp* simply returns nil if g_val is not a number, is probably the most important reason that *signp* is used. The permitted values for s_test and what they mean are given in this table.

s_test	tested
l	g_val $<$ 0
le	g_val $\leqslant$ 0
e	g_val $=$ 0
n	g_val $\neq$ 0
ge	g_val $\geqslant$ 0
g	g_val $>$ 0

(eq 'g_arg1 'g_arg2)

RETURNS: t if g_arg1 and g_arg2 are the exact same lisp object.

NOTE: *Eq* simply tests if g_arg1 and g_arg2 are located in exactly the same place in memory. Lisp objects that print the same are not necessarily *eq*. The only objects guaranteed to be *eq* are interned symbols with the same print name. Unless a symbol is created in a special way (such as with *uconcat* or *maknam*) it is interned.

(neq 'g_x 'g_y)

RETURNS: t if g_x is not *eq* to g_y, otherwise nil.

(equal 'g_arg1 'g_arg2)
(eqstr 'g_arg1 'g_arg2)

RETURNS: t if g_arg1 and g_arg2 have the same structure as described below.

NOTE: g_arg and g_arg2 are *equal* if

(1) They are *eq*.

(2) They are both fixnums with the same value

(3) They are both flonums with the same value

(4) They are both bignums with the same value

(5) They are both strings and are identical.

(6) They are both lists and their cars and cdrs are *equal*

```
; eq is much faster than equal, especially in compiled code.
; However, you cannot use eq to test for equality of numbers outside
; of the range -1024 to 1023.  equal always works.
-> (eq 1023 1023)
t
-> (eq 1024 1024)
nil
-> (equal 1024 1024)
t
```

(not 'g_arg)
(null 'g_arg)

RETURNS: t if g_arg is nil.

(member 'g_arg1 'l_arg2)
(memq 'g_arg1 'l_arg2)

RETURNS: That part of the l_arg2 beginning with the first occurrence of g_arg1. If g_arg1 is not in the top level of l_arg2, nil is returned.

NOTE: *member* tests for equality with *equal*, *memq* tests for equality with *eq*.

2.3. Symbols and Strings

In many of the following functions, the distinction between symbols and strings is somewhat blurred. For FRANZ LISP, a string is a null terminated sequence of characters, stored as compactly as possible. Strings are used as constants in FRANZ LISP. They *eval* to themselves. A symbol has additional structure: a value, property list, function binding, as well as its external representation (or print-name). If a symbol is given to one of the string manipulation functions below, its print name is used as the string.

Another popular way to represent strings in Lisp is as a list of fixnums which represent characters. The suffix 'n' to a string manipulation function indicates that it returns a string in this form.

2.3.1. symbol and string creation

(concat ['stn_arg1 ...])
(uconcat ['stn_arg1 ...])

RETURNS: A symbol whose print name is the result of concatenating the print names, string characters, or numerical representations of the sn_arg*i*

NOTE: If no arguments are given, a symbol with a null pname is returned. *concat* places the symbol created on the oblist; the function *uconcat* does the same thing but does not place the new symbol on the oblist.

EXAMPLE: *(concat 'abc (add 3 4) "def")* = abc7def

(concatl 'l_arg)

EQUIVALENT TO: *(apply 'concat 'l_arg)*

(implode 'l_arg)
(maknam 'l_arg)

WHERE: l_arg is a list of symbols, strings and small fixnums.

RETURNS: The symbol whose print name is the result of concatenating the first characters of the print names of the symbols and strings in the list. Any fixnums are converted to the equivalent ASCII character. In order to concatenate entire strings or print names, use the function *concat*

NOTE: *implode* interns the symbol it creates, *maknam* does not.

(gensym ['s_leader])

RETURNS: A new uninterned atom beginning with the first character of s_leader's pname, or beginning with g if s_leader is not given.

NOTE: The symbol looks like x0nnnnn where x is s_leader's first character and nnnnn is the number of times you have called gensym.

(copysymbol 's_arg 'g_pred)

RETURNS: An uninterned symbol with the same print name as s_arg. If g_pred is non nil, then the value, function binding, and property list of the new symbol are made *eq* to those of s_arg.

(ascii 'x_charnum)

WHERE: x_charnum is between 0 and 255.

RETURNS: A symbol whose print name is the single character whose fixnum representation is x_charnum.

(intern 's_arg)

RETURNS: s_arg

SIDE EFFECT: s_arg is put on the oblist if it is not already there.

(remob 's_symbol)

RETURNS: s_symbol

SIDE EFFECT: s_symbol is removed from the oblist.

(rematom 's_arg)

RETURNS: t if s_arg is indeed an atom.

SIDE EFFECT: s_arg is put on the free atoms list, effectively reclaiming an atom cell.

NOTE: This function does *not* check to see if s_arg is on the oblist or is referenced anywhere. While *rematom* enables you to reclaim a small amount of storage, and can be used effectively with gensym'd atoms, you must be extremely cautious. If you use it on an interned atom which is referenced by some s-expression, you may be find that one or more different atoms are synonymous. This can lead to errors which are very difficult to detect. This function should be used only when storage optimization is important and you are creating many atoms which rapidly outlive their usefulness.

2.3.2. string and symbol predicates

(boundp 's_name)

RETURNS: Nil if s_name is unbound; that is, it has never been given a value. If x_name has the value g_val, then (nil . g_val) is returned. See also *makunbound.*

(alphalessp 'st_arg1 'st_arg2)

RETURNS: t if the 'name' of st_arg1 is alphabetically less than the name of st_arg2. If st_arg is a symbol, then its 'name' is its print name. If st_arg is a string, then its 'name' is the string itself.

2.3.3. symbol and string accessing

(symeval 's_arg)

RETURNS: The value of symbol s_arg.

NOTE: It is illegal to ask for the value of an unbound symbol. This function has the same effect as *eval,* but compiles into much more efficient code.

(get_pname 's_arg)

RETURNS: The string that is the print name of s_arg.

(plist 's_arg)

RETURNS: The property list of s_arg.

(getd 's_arg)

RETURNS: The function definition of s_arg or nil if there is no function definition.

NOTE: The function definition may turn out to be an array header.

(getchar 's_arg 'x_index)
(nthchar 's_arg 'x_index)
(getcharn 's_arg 'x_index)

RETURNS: The x_index*th* character of the print name of s_arg or nil if x_index is less than 1 or greater than the length of s_arg's print name.

NOTE: *getchar* and *nthchar* return a symbol with a single character print name; *getcharn* returns the fixnum representation of the character.

(substring 'st_string 'x_index ['x_length])
(substringn 'st_string 'x_index ['x_length])

RETURNS: A string of length at most x_length starting at x_index*th* character in the string.

NOTE: If x_length is not given, all of the characters for x_index to the end of the string are returned. If x_index is negative, the string begins at the x_index*th* character from the end. If x_index is out of bounds, nil is returned.

NOTE: *substring* returns a list of symbols; *substringn* returns a list of fixnums. If *substringn* is given a 0 x_length argument, then a single fixnum, which is the x_index*th* character, is returned.

2.3.4. symbol and string manipulation

(set 's_arg1 'g_arg2)

RETURNS: g_arg2.

SIDE EFFECT: The value of s_arg1 is set to g_arg2.

(setq s_atm1 'g_val1 [s_atm2 'g_val2 ... ...])

WHERE: The arguments are pairs of atom names and expressions.

RETURNS: The last g_val*i*

SIDE EFFECT: Each s_atm*i* is set to have the value g_val*i*

NOTE: *set* evaluates all of its arguments; *setq* does not evaluate the s_atm*i*

(desetq sl_pattern1 'g_exp1 [... ...])

RETURNS: g_expn

SIDE EFFECT: This acts just like *setq* if all the sl_pattern*i* are symbols. If sl_pattern*i* is a list, then it is a template which should have the same structure as g_exp*i*. The symbols in sl_pattern are assigned to the corresponding parts of g_exp. (See also *setf*)

EXAMPLE: *(desetq (a b (c . d)) '(1 2 (3 4 5)))*
sets a to 1, b to 2, c to 3, and d to (4 5).

(setplist 's_atm 'l_plist)

RETURNS: l_plist.

SIDE EFFECT: The property list of s_atm is set to l_plist.

(makunbound 's_arg)

RETURNS: s_arg

SIDE EFFECT: The value of s_arg is made 'unbound'. If the interpreter attempts to evaluate s_arg before it is again given a value, an unbound variable error occurs.

(aexplode 's_arg)
(explode 'g_arg)
(aexplodec 's_arg)
(explodec 'g_arg)
(aexploden 's_arg)
(exploden 'g_arg)

RETURNS: A list of the characters used to print out s_arg or g_arg.

NOTE: The functions beginning with 'a' are internal functions that are limited to symbol arguments. The functions *aexplode* and *explode* return a list of characters that *print* would use to print the argument. These characters include all necessary escape characters. The functions *aexplodec* and *explodec* return a list of characters that *patom* would use to print the argument (that is, no escape characters). The functions *aexploden* and *exploden* are similar to *aexplodec* and *explodec* except that a list of fixnum equivalents of characters are returned.

```
-> (setq x |quote this \| ok?|)
|quote this \| ok?|
-> (explode x)
(q u o t e |\\|| |t h i s |\\|| ||\\||\|||\\|| |o k ?)
; note that |\\| just means the single character: backslash.
; and |\| just means the single character: vertical bar
; and | | means the single character: space

-> (explodec x)
(q u o t e | |t h i s | ||\|| |o k ?)
-> (exploden x)
(113 117 111 116 101 32 116 104 105 115 32 124 32 111 107 63)
```

2.4. Vectors

See Chapter 9 for a discussion of vectors. They are slightly less efficient that hunks but more efficient than arrays.

2.4.1. vector creation

(**new-vector** 'x_size ['g_fill ['g_prop]])

RETURNS: A **vector** of length x_size. Each data entry is initialized to g_fill, or to nil, if the argument g_fill is not present. The vector's property is set to g_prop, or to nil, by default.

(**new-vectori-byte** 'x_size ['g_fill ['g_prop]])
(**new-vectori-word** 'x_size ['g_fill ['g_prop]])
(**new-vectori-long** 'x_size ['g_fill ['g_prop]])

RETURNS: A **vectori** with x_size elements in it. The actual memory requirement is two long words + x_size*(n bytes), where n is 1 for new-vector-byte, 2 for new-vector-word, or 4 for new-vectori-long. Each data entry is initialized to g_fill, or to zero, if the argument g_fill is not present. The vector's property is set to g_prop, or nil, by default.

Vectors may be created by specifying multiple initial values:

(**vector** ['g_val0 'g_val1 ...])

RETURNS: A **vector** with as many data elements as there are arguments. It is quite possible to have a vector with no data elements. The vector's property is a null list.

(**vectori-byte** ['x_val0 'x_val2 ...])
(**vectori-word** ['x_val0 'x_val2 ...])
(**vectori-long** ['x_val0 'x_val2 ...])

RETURNS: A **vectori** with as many data elements as there are arguments. The arguments are required to be fixnums. Only the low order byte or word is used in the case of vectori-byte and vectori-word. The vector's property is null.

2.4.2. vector reference

(**vref** 'v_vect 'x_index)
(**vrefi-byte** 'V_vect 'x_bindex)
(**vrefi-word** 'V_vect 'x_windex)
(**vrefi-long** 'V_vect 'x_lindex)

RETURNS: The desired data element from a vector. The indices must be fixnums. Indexing is zero-based. The vrefi functions sign extend the data.

(vprop 'Vv_vect)

RETURNS: The Lisp property associated with a vector.

(vget 'Vv_vect 'g_ind)

RETURNS: The value stored under g_ind if the Lisp property associated with 'Vv_vect is a disembodied property list.

(vsize 'Vv_vect)
(vsize-byte 'V_vect)
(vsize-word 'V_vect)

RETURNS: The number of data elements in the vector. For immediate-vectors, the functions vsize-byte and vsize-word return the number of data elements, if you think of the binary data as being compromised of bytes or words.

2.4.3. vector modfication

(vset 'v_vect 'x_index 'g_val)
(vseti-byte 'V_vect 'x_bindex 'x_val)
(vseti-word 'V_vect 'x_windex 'x_val)
(vseti-long 'V_vect 'x_lindex 'x_val)

RETURNS: The datum.

SIDE EFFECT: The indexed element of the vector is set to the value. As noted above, for vseti-word and vseti-byte, the index is construed as the number of the data element within the vector. It is not a byte address. Also, for those two functions, the low order byte or word of x_val is what is stored.

(vsetprop 'Vv_vect 'g_value)

RETURNS: g_value. This should be either a symbol or a disembodied property list whose *car* is a symbol identifying the type of the vector.

SIDE EFFECT: The property list of Vv_vect is set to g_value.

(vputprop 'Vv_vect 'g_value 'g_ind)

RETURNS: g_value.

SIDE EFFECT: If the vector property of Vv_vect is a disembodied property list, then vputprop adds the value g_value under the indicator g_ind. Otherwise, the old vector property is made the first element of the list.

2.5. Arrays

See Chapter 9 for a complete description of arrays. Some of these functions are part of a Maclisp array compatibility package representing only one simple way of using the array structure of FRANZ LISP.

2.5.1. array creation

(marray 'g_data 's_access 'g_aux 'x_length 'x_delta**)**

RETURNS: An array type with the fields set up from the above arguments in the obvious way (see § 1.2.10).

(*array 's_name 's_type 'x_dim1 ... 'x_dim*n***)**
(array s_name s_type x_dim1 ... x_dim*n***)**

WHERE: s_type may be one of t, nil, fixnum, flonum, fixnum-block, or flonum-block.

RETURNS: An array of type s_type with n dimensions of extents given by the x_dim*i*

SIDE EFFECT: If s_name is non nil, the function definition of s_name is set to the array structure returned.

NOTE: These functions create a Maclisp compatible array. In FRANZ LISP arrays of type t, nil, fixnum, and flonum are equivalent and the elements of these arrays can be any type of lisp object. Fixnum-block and flonum-block arrays are restricted to fixnums and flonums respectively and are used mainly to communicate with foreign functions (see §8.5).

NOTE: **array* evaluates its arguments, *array* does not.

2.5.2. array predicate

(arrayp 'g_arg**)**

RETURNS: t if g_arg is of type array.

2.5.3. array accessors

(getaccess 'a_array**)**
(getaux 'a_array**)**
(getdelta 'a_array**)**
(getdata 'a_array**)**
(getlength 'a_array**)**

RETURNS: The field of the array object a_array given by the function name.

(arrayref 'a_name 'x_ind)

RETURNS: The x_ind*th* element of the array object a_name. x_ind of zero accesses the first element.

NOTE: *arrayref* uses the data, length, and delta fields of a_name to determine which object to return.

(arraycall s_type 'as_array 'x_indl ...)

RETURNS: The element selected by the indices from the array a_array of type s_type.

NOTE: If as_array is a symbol, then the function binding of this symbol should contain an array object.
s_type is ignored by *arraycall* but is included for compatibility with Maclisp.

(arraydims 's_name)

RETURNS: A list of the type and bounds of the array s_name.

(listarray 'sa_array ['x_elements])

RETURNS: A list of all of the elements in array sa_array. If x_elements is given, then only the first x_elements are returned.

```
; This creates a 3 by 4 array of general lisp objects.
-> (array ernie t 3 4)
array[12]

; The array header is stored in the function definition slot of the
; symbol ernie.
-> (arrayp (getd 'ernie))
t
-> (arraydims (getd 'ernie))
(t 3 4)

; Store in ernie[2][2] the list (test list).
-> (store (ernie 2 2) '(test list))
(test list)

; Check to see if it is there.
-> (ernie 2 2)
(test list)

; Now use the low level function arrayref to find the same element.
; Arrays are 0 based and row-major (the last subscript varies the fastest);
; thus, element [2][2] is the 10th element, starting at 0.
-> (arrayref (getd 'ernie) 10)
(ptr to)(test list)    ; The result is a value cell (thus the (ptr to)).
```

2.5.4. array manipulation

(putaccess 'a_array 'su_func**)**
(putaux 'a_array 'g_aux**)**
(putdata 'a_array 'g_arg**)**
(putdelta 'a_array 'x_delta**)**
(putlength 'a_array 'x_length**)**

RETURNS: The second argument to the function.

SIDE EFFECT: The field of the array object given by the function name is replaced by the second argument to the function.

(store 'l_arexp 'g_val**)**

WHERE: l_arexp is an expression that references an array element.

RETURNS: g_val

SIDE EFFECT: The array location that contains the element that l_arexp references is changed to contain g_val.

(fillarray 's_array 'l_itms**)**

RETURNS: s_array

SIDE EFFECT: The array s_array is filled with elements from l_itms. If there are not enough elements in l_itms to fill the entire array, then the last element of l_itms is used to fill the remaining parts of the array.

2.6. Hunks

Hunks are vector-like objects whose size can range from 1 to 128 elements. Internally, hunks are allocated in sizes that are powers of 2. In order to create hunks of a given size, a hunk with at least that many elements is allocated, and a distinguished symbol EMPTY is placed in those elements not requested. Most hunk functions respect those distinguished symbols, but there are two *(*makhunk* and **rplacx)* that overwrite the distinguished symbol.

2.6.1. hunk creation

(hunk 'g_val1 ['g_val2 ... 'g_val*n*]**)**

RETURNS: A hunk of length n whose elements are initialized to the g_val*i*

NOTE: The maximum size of a hunk is 128.

EXAMPLE: *(hunk 4 'sharp 'keys)* = {4 sharp keys}

(makhunk 'xl_arg)

RETURNS: A hunk of length xl_arg initialized to all nils if xl_arg is a fixnum. If xl_arg is a list, then a hunk of size *(length 'xl_arg)* is returned, initialized to the elements in xl_arg.

NOTE: *(makhunk '(a b c))* is equivalent to *(hunk 'a 'b 'c)*.

EXAMPLE: *(makhunk 4)* = {*nil nil nil nil*}

(*makhunk 'x_arg)

RETURNS: A hunk of size 2^{x_arg} initialized to EMPTY.

NOTE: This is only to be used by such functions as *hunk* and *makhunk*, which create and initialize hunks for users.

2.6.2. hunk accessor

(cxr 'x_ind 'h_hunk)

RETURNS: Element x_ind (starting at 0) of hunk h_hunk.

(hunk-to-list 'h_hunk)

RETURNS: A list consisting of the elements of h_hunk.

2.6.3. hunk manipulators

(rplacx 'x_ind 'h_hunk 'g_val)
(*rplacx 'x_ind 'h_hunk 'g_val)

RETURNS: h_hunk

SIDE EFFECT: Element x_ind (starting at 0) of h_hunk is set to g_val.

NOTE: *rplacx* does not modify one of the distinguished (EMPTY) elements whereas **rplacx* does.

(hunksize 'h_arg)

RETURNS: The size of the hunk h_arg.

EXAMPLE: *(hunksize (hunk 1 2 3))* = 3

2.7. Bcds

A bcd object contains a pointer to compiled code and to the type of function object the compiled code represents.

(getdisc 'y_bcd**)**
(getentry 'y_bcd**)**

RETURNS: The field of the bcd object given by the function name.

(putdisc 'y_func 's_discipline**)**

RETURNS: s_discipline

SIDE EFFECT: Sets the discipline field of y_func to s_discipline.

2.8. Structures

There are three common structures constructed out of list cells: the assoc list, the property list, and the tconc list. The functions below manipulate these structures.

2.8.1. assoc list

An 'assoc list' (or alist) is a common lisp data structure. It has the form
((key1 . value1) (key2 . value2) (key3 . value3) ... (keyn . valuen))

(assoc 'g_arg1 'l_arg2**)**
(assq 'g_arg1 'l_arg2**)**

RETURNS: The first top level element of l_arg2 whose *car* is *equal* (with *assoc*) or *eq* (with *assq*) to g_arg1.

NOTE: Usually l_arg2 has an *a-list* structure and g_arg1 acts as key.

(sassoc 'g_arg1 'l_arg2 'sl_func**)**

RETURNS: The result of *(cond ((assoc 'g_arg 'l_arg2) (apply 'sl_func nil)))*

NOTE: sassoc is written as a macro.

(sassq 'g_arg1 'l_arg2 'sl_func**)**

RETURNS: the result of *(cond ((assq 'g_arg 'l_arg2) (apply 'sl_func nil)))*

NOTE: sassq is written as a macro.

```
; assoc or assq is given a key and an assoc list and returns
; the key and value item if it exists. They differ only in how they test
; for equality of the keys.

-> (setq alist '((alpha . a) ( (complex key) . b) (junk . x)))
((alpha . a) ((complex key) . b) (junk . x))

; You should use assq when the key is an atom;
-> (assq 'alpha alist)
(alpha . a)

; but it may not work when the key is a list.
-> (assq '(complex key) alist)
nil

; However, assoc always works.
-> (assoc '(complex key) alist)
((complex key) . b)
```

(sublis 'l_alst 'l_exp)

WHERE: l_alst is an *a-list*.

RETURNS: The list l_exp with every occurrence of key*i* replaced by val*i*.

NOTE: A new list structure is returned to prevent modification of l_exp. When a substitution is made, a copy of the value to substitute in, is not made.

2.8.2. property list

A property list consists of an alternating sequence of keys and values. Normally a property list is stored on a symbol. A list is a 'disembodied' property list if it contains an odd number of elements, the first of which is ignored.

(plist 's_name)

RETURNS: The property list of s_name.

(setplist 's_atm 'l_plist)

RETURNS: l_plist.

SIDE EFFECT: the property list of s_atm is set to l_plist.

(get 'ls_name 'g_ind**)**

RETURNS: The value under indicator g_ind in ls_name's property list if ls_name is a symbol.

NOTE: If there is no indicator g_ind in ls_name's property list, nil is returned. If ls_name is a list of an odd number of elements, then it is a disembodied property list. *get* searches a disembodied property list by starting at its *cdr* and comparing every other element with g_ind, using *eq*.

(getl 'ls_name 'l_indicators**)**

RETURNS: The property list ls_name beginning at the first indicator that is a member of the list l_indicators, or nil, if none of the indicators in l_indicators are on ls_name's property list.

NOTE: If ls_name is a list, then it is assumed to be a disembodied property list.

(putprop 'ls_name 'g_val 'g_ind**)**
(defprop ls_name g_val g_ind**)**

RETURNS: g_val.

SIDE EFFECT: Adds to the property list of ls_name the value g_val under the indicator g_ind.

NOTE: *putprop* evaluates its arguments; *defprop* does not. ls_name may be a disembodied property list. See *get*

(remprop 'ls_name 'g_ind**)**

RETURNS: The portion of ls_name's property list beginning with the property under the indicator g_ind. If there is no g_ind indicator in ls_name's plist, nil is returned.

SIDE EFFECT: The value under indicator g_ind and g_ind itself is removed from the property list of ls_name.

NOTE: ls_name may be a disembodied property list. See *get*

```
-> (putprop 'xlate 'a 'alpha)
a
-> (putprop 'xlate 'b 'beta)
b
-> (plist 'xlate)
(alpha a beta b)
-> (get 'xlate 'alpha)
a
; You can use a disembodied property list this way:
-> (get '(nil fateman rjf sklower kls foderaro jkf) 'sklower)
kls
```

2.8.3. tconc structure

A tconc structure is a special type of list, designed to make it easy to add objects to the end. It consists of a list cell whose *car* points to a list of the elements added with *tconc* or *lconc* and whose *cdr* points to the last list cell of the list pointed to by the *car*.

(tconc 'l_ptr 'g_x)

WHERE: l_ptr is a tconc structure.

RETURNS: l_ptr with g_x added to the end.

(lconc 'l_ptr 'l_x)

WHERE: l_ptr is a tconc structure.

RETURNS: l_ptr with the list l_x spliced in at the end.

```
; A tconc structure can be initialized in two  ways.
; Nil can be given to tconc, in which case tconc generates
; a tconc structure.

-> (setq foo (tconc nil 1))
((1) 1)

; Since tconc destructively adds to
; the list, you can now add to foo without using setq again.

-> (tconc foo 2)
((1 2) 2)
->foo
((1 2) 2)

; Another way to create a null  tconc structure
; is to use (ncons nil).

-> (setq foo (ncons nil))
(nil)
-> (tconc foo 1)
((1) 1)

; Now see what lconc can do:
->  (lconc foo nil)
((1) 1)                    ;There is no change.
->  (lconc foo '(2 3 4))
((1 2 3 4) 4)
```

2.8.4. fclosures

An fclosure is a functional object that admits some data manipulations. They are discussed in §8.4. Internally, they are constructed from vectors.

(fclosure 'l_vars 'g_funobj)

WHERE: l_vars is a list of variables; g_funobj is any object that can be funcalled (including, fclosures).

RETURNS: A vector that is the fclosure.

(fclosure-alist 'v_fclosure)

RETURNS: An association list representing the variables in the fclosure. This is a snapshot of the current state of the fclosure. If the bindings in the fclosure are changed, any previously calculated results of *fclosure-alist* do not change.

(fclosure-function 'v_fclosure)

RETURNS: The functional object part of the fclosure.

(fclosurep 'v_fclosure)

RETURNS: t if the argument is an fclosure.

(symeval-in-fclosure 'v_fclosure 's_symbol)

RETURNS: The current binding of a particular symbol in an fclosure.

(set-in-fclosure 'v_fclosure 's_symbol 'g_newvalue)

RETURNS: g_newvalue.

SIDE EFFECT: The variable s_symbol is bound in the fclosure to g_newvalue.

2.9. Random functions

The following functions do not fall into any of the classifications above.

(bcdad 's_funcname)

RETURNS: A fixnum that is the address in memory where the function s_funcname begins. If s_funcname is not a machine coded function (binary), then *bcdad* returns nil.

(copy 'g_arg)

RETURNS: A structure *equal* to g_arg but with new list cells.

(copyint* 'x_arg)

RETURNS: A fixnum with the same value as x_arg but in a freshly allocated cell.

(cpy1 'xvt_arg)

RETURNS: A new cell of the same type as xvt_arg with the same value as xvt_arg.

(getaddress 's_entry1 's_binder1 'st_discipline1 [... ... ...]**)**

RETURNS: The binary object that s_binder1's function field is set to.

NOTE: This looks in the running lisp's symbol table for a symbol with the same name as s_entry*i*. It then creates a binary object whose entry field points to s_entry*i* and whose discipline is st_discipline*i*. This binary object is stored in the function field of s_binder*i*. If st_discipline*i* is nil, then "subroutine" is used by default. This is especially useful for *cfasl* users.

(macroexpand 'g_form**)**

RETURNS: g_form after all macros in it are expanded.

NOTE: This function only macroexpands expressions that could be evaluated, and it does not know about the special nlambdas such as *cond* and *do*, thus, it misses many macro expansions.

(ptr 'g_arg**)**

RETURNS: A value cell initialized to point to g_arg.

(quote g_arg**)**

RETURNS: g_arg.

NOTE: The reader allows you to abbreviate (quote foo) as 'foo.

(kwote 'g_arg**)**

RETURNS: *(list (quote quote) g_arg).*

(replace 'g_arg1 'g_arg2**)**

WHERE: g_arg1 and g_arg2 must be the same type of lispval and not symbols or hunks.

RETURNS: g_arg2.

SIDE EFFECT: The effect of *replace* dependents on the type of the g_arg*i*, although you may notice a similarity in the effects. To understand what *replace* does to fixnum and flonum arguments, you must first understand that such numbers are 'boxed' in FRANZ LISP. This means that if the symbol x has a value 32412, then, in memory, the value element of x's symbol structure contains the address of another word of memory (called a box) with 32412 in it.

Thus, there are two ways of changing the value of x. The first way is to change the value element of x's symbol structure to point to a word of memory with a different value. The second way is to change the value in the box that x points to. The former method is used almost all of the time; the latter is used very rarely and may cause great confusion. The function *replace* allows you to do the latter, i.e., to actually change the value in the box.

You should watch out for these situations. If you do *(setq y x)*, then both x and y point to the same box. If you now *(replace x 12345)*, then y also has the value 12345. And, in fact, there may be many other pointers to that box.

Another problem with replacing fixnums is that some boxes are read-only. The fixnums between -1024 and 1023 are stored in a read-only area and attempts to replace them result in an "Illegal memory reference" error. See the description of *copyint** for a way around this problem.

For the other valid types, the effect of *replace* is easy to understand. The fields of g_val1's structure are made eq to the corresponding fields of g_val2's structure. For example, if x and y have lists as values then the effect of *(replace x y)* is the same as *(rplaca x (car y))* and *(rplacd x (cdr y))*.

(**scons** 'x_arg 'bs_rest)

WHERE: bs_rest is a bignum or nil.

RETURNS: A bignum whose first bigit (digit in the bignum base) is x_arg and whose higher order bigits are bs_rest.

(**setf** g_refexpr 'g_value)

NOTE: *setf* is a generalization of setq. Information may be stored by binding variables, replacing entries of arrays, and vectors, or by being put on property lists, among others. Setf allows you to store data into some location by mentioning the operation used to refer to the location. Thus, the first argument may be partially evaluated, but only to the extent needed to calculate a reference. *setf* returns g_value. (Compare to *desetq*)

```
(setf x 3)        = (setq x 3)
(setf (car x) 3)  = (rplaca x 3)
(setf (get foo 'bar) 3) = (putprop foo 3 'bar)
(setf (vref vector index) value) = (vset vector index value)
```

(**sort** 'l_data 'u_comparefn)

RETURNS: A list of the elements of l_data ordered by the comparison function u_comparefn.

SIDE EFFECT: The list l_data is modified rather than allocated in new storage.

NOTE: *(comparefn 'g_x 'g_y)* should return something non-nil, if g_x can precede g_y in sorted order; nil, if g_y must precede g_x. If u_comparefn is nil, alphabetical order is used.

(**sortcar** 'l_list 'u_comparefn)

RETURNS: A list of the elements of l_list with the *car*'s ordered by the sort function u_comparefn.

SIDE EFFECT: The list l_list is modified rather than copied.

NOTE: Like *sort*, if u_comparefn is nil, alphabetical order is used.

CHAPTER 3

Arithmetic Functions

This chapter describes FRANZ LISP's functions for doing arithmetic. Often the same function is known by many names. For example, *add* is also *plus* and *sum*. This is caused by our desire to be compatible with other Lisps. However, you should avoid using functions with names such as + and * unless their arguments are fixnums. The Lisp compiler takes advantage of these implicit declarations.

An attempt to divide or to generate a floating point result outside of the range of floating point numbers causes a floating exception signal from the operating system. You can catch and process this interrupt if desired. See the description of the *signal* function.

3.1. Simple Arithmetic Functions

(**add** ['n_arg1 ...])
(**plus** ['n_arg1 ...])
(**sum** ['n_arg1 ...])
(+ ['x_arg1 ...])

RETURNS: The sum of the arguments. If no arguments are given, 0 is returned.

NOTE: If the size of the partial sum exceeds the limit of a fixnum, the partial sum is converted to a bignum. If any of the arguments are flonums, the partial sum is converted to a flonum when that argument is processed and the result is thus a flonum. Currently, if, in the process of doing the addition, a bignum must be converted into a flonum, an error message results.

(**add1** 'n_arg)
(**1+** 'x_arg)

RETURNS: Its argument plus 1.

(**diff** ['n_arg1 ...])
(**difference** ['n_arg1 ...])
(− ['x_arg1 ...])

RETURNS: The result of subtracting from n_arg1 all subsequent arguments. If no arguments are given, 0 is returned.

NOTE: See the description of add for details on data type conversions and restrictions.

(sub1 'n_arg**)**
(1− 'x_arg**)**

RETURNS: Its argument minus 1.

(minus 'n_arg**)**

RETURNS: Zero minus n_arg.

(product ['n_arg1 ...]**)**
(times ['n_arg1 ...]**)**
(* ['x_arg1 ...]**)**

RETURNS: The product of all of its arguments. It returns 1 if there are no arguments.

NOTE: See the description of the function *add* for details and restrictions to the automatic data type coercion.

(quotient ['n_arg1 ...]**)**
(/ ['x_arg1 ...]**)**

RETURNS: The result of dividing the first argument by succeeding ones.

NOTE: If there are no arguments, 1 is returned. See the description of the function *add* for the details and restrictions of data type coercion. A divide by zero causes a floating exception interrupt. See the description of the *signal* function.

(*quo 'i_x 'i_y**)**

RETURNS: The integer part of i_x / i_y.

(Divide 'i_dividend 'i_divisor**)**

RETURNS: A list whose car is the quotient and whose cadr is the remainder of the division of i_dividend by i_divisor.

NOTE: This is restricted to integer division.

(Emuldiv 'x_fact1 'x_fact2 'x_addn 'x_divisor**)**

RETURNS: A list of the quotient and remainder of this operation: ((x_fact1 * x_fact2) + (sign extended) x_addn) / x_divisor.

NOTE: This is useful for creating a bignum arithmetic package in Lisp.

3.2. predicates

(numberp 'g_arg**)**

(numbp 'g_arg**)**

RETURNS: T iff g_arg is a number: fixnum, flonum, or bignum.

(fixp 'g_arg)

RETURNS: T iff g_arg is a fixnum or bignum.

(floatp 'g_arg)

RETURNS: T iff g_arg is a flonum.

(evenp 'x_arg)

RETURNS: T iff x_arg is even.

(oddp 'x_arg)

RETURNS: T iff x_arg is odd.

(zerop 'g_arg)

RETURNS: T iff g_arg is a number equal to 0.

(onep 'g_arg)

RETURNS: T iff g_arg is a number equal to 1.

(plusp 'n_arg)

RETURNS: T iff n_arg is greater than zero.

(minusp 'g_arg)

RETURNS: T iff g_arg is a negative number.

(greaterp ['n_arg1 ...])
(> 'fx_arg1 'fx_arg2)
(>& 'x_arg1 'x_arg2)

RETURNS: T iff the arguments are in a strictly decreasing order.

NOTE: In the functions *greaterp* and >, the function *difference* is used to compare adjacent values. If any of the arguments are non-numbers, the error message comes from the *difference* function. The arguments to > must be fixnums or both flonums. The arguments to >& must both be fixnums.

(lessp ['n_arg1 ...])
(< 'fx_arg1 'fx_arg2)
(<& 'x_arg1 'x_arg2)

RETURNS: T iff the arguments are in a strictly increasing order.

NOTE: In functions *lessp* and < the function *difference* is used to compare adjacent values. If any of the arguments are non numbers, the error message comes from the *difference* function. The arguments to < may be either fixnums or flonums but must be the same type. The arguments to <& must be fixnums.

(= 'fx_arg1 'fx_arg2)

(=& 'x_arg1 'x_arg2)

RETURNS: T iff the arguments have the same value. The arguments to = must be the either both fixnums or both flonums. The arguments to =& must be fixnums.

3.3. Trignometric Functions

Some of these functions are taken from the host math library, and we take no further responsibility for their accuracy.

(cos 'fx_angle)

RETURNS: The (flonum) cosine of fx_angle (which is assumed to be in radians).

(sin 'fx_angle)

RETURNS: The sine of fx_angle (which is assumed to be in radians).

(acos 'fx_arg)

RETURNS: The (flonum) arc cosine of fx_arg in the range 0 to π.

(asin 'fx_arg)

RETURNS: The (flonum) arc sine of fx_arg in the range $-\pi/2$ to $\pi/2$.

(atan 'fx_arg1 'fx_arg2)

RETURNS: The (flonum) arc tangent of fx_arg1/fx_arg2 in the range $-\pi$ to π.

3.4. Bignum/Fixnum Manipulation

(haipart bx_number x_bits)

RETURNS: A fixnum (or bignum) that contains the x_bits high bits of *(abs bx_number)* if x_bits is positive; otherwise, it returns the *(abs x_bits)* low bits of *(abs bx_number)*.

(haulong bx_number)

RETURNS: The number of significant bits in bx_number.

NOTE: The result is equal to the least integer greater than or equal to the base two logarithm of one plus the absolute value of bx_number.

(bignum-leftshift bx_arg x_amount)

RETURNS: bx_arg shifted left by x_amount. If x_amount is negative, bx_arg is shifted right by the magnitude of x_amount.

NOTE: If bx_arg is shifted right, it will be rounded to the nearest even number.

(sticky-bignum-leftshift 'bx_arg 'x_amount)

RETURNS: bx_arg shifted left by x_amount. If x_amount is negative, bx_arg will be shifted right by the magnitude of x_amount and rounded.

NOTE: Sticky rounding is done this way: after shifting, the low order bit is changed to 1 if any 1's were shifted off to the right.

3.5. Bit Manipulation

(boole 'x_key 'x_v1 'x_v2 ...)

RETURNS: The result of the bitwise boolean operation as described in the following table.

NOTE: If there are more than 3 arguments, then evaluation proceeds left to right with each partial result becoming the new value of x_v1. That is,
(boole 'key 'v1 'v2 'v3) ≡ *(boole 'key (boole 'key 'v1 'v2) 'v3).*
In the following table, * represents bitwise and + represents bitwise, or ⊕ represents bitwise xor and ¬ represents bitwise negation and is the highest precedence operator.

(boole 'key 'x 'y)								
key	0	1	2	3	4	5	6	7
result	0	x * y	¬ x * y	y	x * ¬ y	x	x ⊕ y	x + y
common names		and			bitclear		xor	or
key	8	9	10	11	12	13	14	15
result	¬ (x + y)	¬(x ⊕ y)	¬ x	¬ x + y	¬ y	x + ¬ y	¬ x + ¬ y	-1
common names	nor	equiv		implies			nand	

(lsh 'x_val 'x_amt)

RETURNS: x_val shifted left by x_amt if x_amt is positive. If x_amt is negative, then *lsh* returns x_val shifted right by the magnitude if x_amt.

NOTE: This always returns a fixnum even for those numbers whose magnitude is so large that they would normally be represented as a bignum; i.e., shifter bits are lost. For more general bit shifters, see *bignum-leftshift* and *sticky-bignum-leftshift.*

(rot 'x_val 'x_amt**)**

RETURNS: x_val rotated left by x_amt if x_amt is positive. If x_amt is negative, then x_val is rotated right by the magnitude of x_amt.

3.6. Other Functions

As noted above, some of the following functions are inherited from the host math library.

(abs 'n_arg**)**
(absval 'n_arg**)**

RETURNS: The absolute value of n_arg.

(exp 'fx_arg**)**

RETURNS: *e* raised to the fx_arg power (flonum).

(expt 'n_base 'n_power**)**

RETURNS: n_base raised to the n_power power.

NOTE: If either of the arguments are flonums, the calculation is done using *log* and *exp.*

(fact 'x_arg**)**

RETURNS: x_arg factorial -- fixnum or bignum.

(fix 'n_arg**)**

RETURNS: A fixnum as close as we can get to n_arg.

NOTE: *fix* rounds down. Currently, if n_arg is a flonum larger than the size of a fixnum, this fails.

(float 'n_arg**)**

RETURNS: A flonum as close as we can get to n_arg.

NOTE: If n_arg is a bignum larger than the maximum size of a flonum, then a floating exception occurs.

(log 'fx_arg**)**

RETURNS: The natural logarithm of fx_arg.

(max 'n_arg1 ... **)**

RETURNS: The maximum value in the list of arguments.

(min 'n_arg1 ...)

RETURNS: The minimum value in the list of arguments.

(mod 'i_dividend 'i_divisor)
(remainder 'i_dividend 'i_divisor)

RETURNS: The remainder when i_dividend is divided by i_divisor.

NOTE: The sign of the result has the same sign as i_dividend.

(*mod 'x_dividend 'x_divisor)

RETURNS: The balanced representation of x_dividend modulo x_divisor.

NOTE: The range of the balanced representation is abs(x_divisor)/2 to (abs(x_divisor)/2) − x_divisor + 1.

(random ['x_limit])

RETURNS: A fixnum between 0 and x_limit − 1 if x_limit is given. If x_limit is not given, any fixnum, positive or negative, might be returned.

(sqrt 'fx_arg)

RETURNS: The square root of fx_arg.

CHAPTER 4

Special Functions

This chapter describes the special functions, or forms of FRANZ LISP. While lisp is generally thought of as very simple, in fact system-building in lisp requires the inclusion of a fair selection of these special forms.

(and [g_arg1 ...])

RETURNS: The value of the last argument if all arguments evaluate to a non-nil value; otherwise, *and* returns nil. It returns t if there are no arguments.

NOTE: The arguments are evaluated left to right and evaluation ceases with the first nil encountered.

(apply 'u_func 'l_args)

RETURNS: The result of applying function u_func to the arguments in the list l_args.

NOTE: If u_func is a lambda, then the *(length l_args)* should equal the number of formal parameters for the u_func. If u_func is a nlambda or macro, then l_args is bound to the single formal parameter.

```
; add1 is a lambda of 1 argument
-> (apply 'add1 '(3))
4

; You can define plus1 as a macro that is equivalent to
add1.
-> (def plus1 (macro (arg) (list 'add1 (cadr arg))))
plus1
-> (plus1 3)
4

; Now if you apply a macro, you obtain the form it changes to.
-> (apply 'plus1 '(plus1 3))
(add1 3)

; If you funcall a macro ,however, the result
of the macro is evaled
; before it is returned.
-> (funcall 'plus1 '(plus1 3))
4

; For this particular macro, the car of the arg is not checked
; so that this too works.
-> (apply 'plus1 '(foo 3))
(add1 3)
```

(arg ['x_numb])

RETURNS: If x_numb is specified, then the x_numb'*th* argument to the enclosing lexpr. If x_numb is not specified, then this returns the number of arguments to the enclosing lexpr.

NOTE: It is an error to the interpreter if x_numb is given and out of range.

(break [g_message ['g_pred]])

WHERE: If g_message is not given, it is assumed to be the null string, and if g_pred is not given, it is assumed to be t.

RETURNS: The value of *(*break 'g_pred 'g_message)*

(*break 'g_pred 'g_message)

RETURNS: nil immediately if g_pred is nil; otherwise, the value of the next (return 'value) expression typed in at top level.

SIDE EFFECT: If the predicate, g_pred, evaluates to non-null, the Lisp system stops and prints out 'Break ' followed by g_message. It then enters a break loop that allows you to interactively debug a program. To continue execution from a break, you can use the *return* function. To return to top level or another break level, you can use *retbrk* or *reset*

(caseq 'g_key-form l_clause1 ...)

WHERE: l_clause*i* is a list of the form (g_comparator ['g_form*i* ...]). The comparators may be symbols, small fixnums, a list of small fixnums or symbols.

NOTE: The way caseq works is that it evaluates g_key-form, yielding a value called the selector. Each clause is examined until the selector is found consistent with the comparator. For a symbol, or a fixnum, this means the two must be *eq*. For a list, this means that the selector must be *eq* to some element of the list.

The comparator consisting of the symbol **t** has special semantics: it matches anything and, consequently, should be the last comparator.

In any case, having chosen a clause, *caseq* evaluates each form within that clause and returns the value of the last form

RETURNS: The value of the last form as indicated above. If no comparators are matched, *caseq* returns nil.

Here are two ways of defining the same function:

```
-> (defun fate (personna)
        (caseq personna
          (cow '(jumped over the moon))
          (cat '(played nero))
          ((dish spoon) '(ran away with each other))
          (t '(lived happily ever after))))
fate
-> (defun fate (personna)
        (cond
                ((eq personna 'cow) '(jumped over the moon))
                ((eq personna 'cat) '(played nero))
                ((memq personna '(dish spoon)) '(ran away with each other))
                (t '(lived happily ever after))))
fate
```

(catch g_exp [ls_tag]**)**

WHERE: If ls_tag is not given, it is assumed to be nil.

RETURNS: The result of *(*catch 'ls_tag g_exp)*

NOTE: Catch is defined as a macro.

(*catch 'ls_tag g_exp**)**

WHERE: ls_tag is either a symbol or a list of symbols.

RETURNS: The result of evaluating g_exp or, if the 'throw' pseudo-function is invoked with the argument ls_tag within the execution of g_exp, the value given by throw. (see throw, *throw) The *catch and throw or *throw construction is used for a non-local return of a value, and is typically used in an error return or some kind of break in the normal modularization of a program.

SIDE EFFECT: This proceeds as follows: *catch first sets up a 'catch frame' on the Lisp runtime stack. Then it begins to evaluate g_exp. If g_exp evaluates normally, its value is returned. If, however, a value is thrown during the evaluation of g_exp, then this *catch returns with that value if one of these cases is true:

(1) The tag thrown to is ls_tag.

(2) ls_tag is a list and the tag thrown to is a member of this list.

(3) ls_tag is nil.

NOTE: Errors are implemented as a special kind of throw. A catch with no tag does not catch an error, but a catch whose tag is the error type catches that type of error. See Chapter 10 for more information.

(comment [g_arg ...])

RETURNS: The symbol comment.

NOTE: This does absolutely nothing.

(cond [l_clause1 ...])

RETURNS: The last value evaluated in the first clause satisfied. If no clauses are satisfied, then nil is returned.

NOTE: This is the basic conditional 'statement' in Lisp. The clauses are processed from left to right. The first element of a clause is evaluated. If it evaluates to a non-null value, then that clause is satisfied and all following elements of that clause are evaluated. The last value computed is returned as the value of the cond. If there is just one element in the clause, then its value is returned. If the first element of a clause evaluates to nil, then the other elements of that clause are not evaluated and the system moves to the next clause.

(cvttointlisp)

SIDE EFFECT: The reader is modified to conform with the Interlisp syntax. The character % is made the escape character and special meanings for comma, backquote, and backslash are removed. Also the reader is told to convert upper case to lower case.

(cvttofranzlisp)

SIDE EFFECT: FRANZ LISP's default syntax is reinstated. You should run this function after having run any of the other *cvtto-* functions. Backslash is made the escape character, super-brackets work again, and the reader distinguishes between upper and lower case.

(cvttomaclisp)

SIDE EFFECT: The reader is modified to conform with Maclisp syntax. The character / is made the escape character, and the special meanings for backslash, left and right bracket are removed. The reader is made case-insensitive.

(cvttoucilisp)

SIDE EFFECT: The reader is modified to conform with UCI Lisp syntax. The character / is made the escape character; tilde is made the comment character; exclamation point takes on the unquote function normally held by comma, and backslash, comma, and semicolon become normal characters. Here too, the reader is made case-insensitive.

(debug s_msg)

SIDE EFFECT: Enter the Fixit package described in Chapter 15. This package allows you to examine the evaluation stack in detail. To leave the Fixit package type 'ok'.

(debugging 'g_arg)

SIDE EFFECT: If g_arg is non-null, FRANZ LISP unlinks the transfer tables, does a *(*rset t)* to turn on evaluation monitoring and sets the all-error catcher (ER%all) to be *debug-err-handler.* If g_arg is nil, all of the earlier changes are undone.

(declare [g_arg ...])

RETURNS: nil

NOTE: This is a no-op to the evaluator. It has special meaning to the compiler (see Chapter 12).

(def s_name (s_type l_argl g_exp1 ...))

WHERE: s_type is one of lambda, nlambda, macro or lexpr.

RETURNS: s_name

SIDE EFFECT: This defines the function s_name to the Lisp system. If s_type is nlambda or macro then the argument list l_argl must contain exactly one non-nil symbol.

(defmacro s_name l_arg g_exp1 ...)
(defcmacro s_name l_arg g_exp1 ...)

RETURNS: s_name

SIDE EFFECT: This defines the macro s_name. *defmacro* makes it easy to write macros since it makes the syntax just like *defun.* Further information on *defmacro* is in §8.3.2. *defcmacro* defines compiler-only macros, or cmacros. A cmacro is stored on the property list of a symbol under the indicator **cmacro**. Thus a function can have a normal definition and a cmacro definition. For an example of the use of cmacros, you can examine the definitions of nthcdr and nth in /lisp/lib/common2.l

(defun s_name [s_mtype] ls_argl g_exp1 ...)

WHERE: s_mtype is one of fexpr, expr, args or macro.

RETURNS: s_name

SIDE EFFECT: This defines the function s_name.

NOTE: This exists for Maclisp compatibility. It is just a macro that changes the defun form to the def form. If you are familiar with Maclisp, an s_mtype of fexpr is converted to nlambda and the Maclisp expr is simply our lambda. Macro remains the same. If ls_argl is a non-nil symbol, then the type is assumed to be lexpr and ls_argl is the symbol that is bound to the number of args when the function is entered.
For compatibility with the Lisp Machine Lisp, there are three types of optional parameters that can occur in ls_argl: *&optional* declares that the following symbols are optional, and may or may not appear in the argument list to the function; *&rest symbol* declares that all forms in the function call that are not accounted for by previous lambda bindings are to be assigned to *symbol,* and *&aux form1 ... formn* declares that the *formi* are either symbols, in which case they are lambda bound to **nil**, or lists, in which case the first element of the list is lambda bound to the second, evaluated element.

```
; def and defun here are used to define identical
functions.
; You can decide for yourself which is easier to use.
-> (def append1 (lambda (lis extra) (append lis (list extra))))
append1

-> (defun append1 (lis extra) (append lis (list extra)))
append1

; Using the & forms...
-> (defun test (a b &optional c &aux (retval 0) &rest z)
         (if c them (msg "Optional arg present" N
                          "c is" c N))
         (msg "rest is" z N
              "retval is" retval N))
test
-> (test 1 2 3 4)
Optional arg present
c is 3
rest is (4)
retval is 0
```

(defvar s_variable ['g_init]**)**

RETURNS: s_variable.

NOTE: This form is put at the top level in files, like *defun*

SIDE EFFECT: This declares s_variable to be special. If g_init is present and s_variable is unbound when the file is read in, s_variable is set to the value of g_init. An advantage of '(defvar foo)' over '(declare (special foo))' is that if a file containing defvars is loaded (or fasl'ed) in during compilation, the variables mentioned in the defvar's are declared special. The only way to have that effect with '(declare (special foo))' is to *include* the file.

(do l_vrbs l_test g_exp1 ...**)**

RETURNS: The last form in the cdr of l_test evaluated, or a value explicitly given by a return evaluated within the do body.

NOTE: This is the basic iteration form for FRANZ LISP. l_vrbs is a list of zero or more var-init-repeat forms. A var-init-repeat form looks like:

(s_name [g_init [g_repeat]])

There are three cases depending on what is present in the form. If just s_name is present, this means that when the do is entered, s_name is lambda-bound to nil and is never modified by the system (though the program is certainly free to modify its value). If the form is (s_name 'g_init) then the only difference is that s_name is lambda-bound to the value of g_init instead of nil. If g_repeat is also present then s_name is lambda-bound to g_init when the loop is entered and after each pass through the do body s_name is bound to the value of g_repeat.
l_test is either nil or has the form of a cond clause. If it is nil then the do body is evaluated only once and the do returns nil. Otherwise, before the do body is evaluated the car of l_test is evaluated, and, if the result is non-null, this signals an end to the looping. Then the rest of the forms in l_test are evaluated and the value of the last one is returned as the value of the do. If the cdr of l_test is nil, then nil is returned. Thus, this is not exactly like a cond clause.

g_expl and those forms that follow constitute the do body. A do body is like a prog body and, thus, may have labels. You can use the functions go and return. The sequence of evaluations is this:

(1) The init forms are evaluated left to right and stored in temporary locations.

(2) Simultaneously, all do variables are lambda bound to the value of their init forms or to nil.

(3) If l_test is non-null, then the car is evaluated, and, if it is non-null, the rest of the forms in l_test are evaluated, and the last value is returned as the value of the do.

(4) The forms in the do body are evaluated left to right.

(5) If l_test is nil the do function returns with the value nil.

(6) The repeat forms are evaluated and saved in temporary locations.

(7) The variables with repeat forms are simultaneously bound to the values of those forms.

(8) Go to step 3.

NOTE: There is an alternate form of do that can be used when there is only one do variable. It is described next.

```
; This is  a simple function that numbers the elements of a list.
; It uses a do function with two local variables.
-> (defun printem (lis)
          (do ((xx lis (cdr xx))
               (i 1 (1+ i)))
              ((null xx) (patom "all done") (terpr))
              (print i)
              (patom ": ")
              (print (car xx))
              (terpr)))
printem
-> (printem '(a b c d))
1: a
2: b
3: c
4: d
all done
nil
->
```

(do s_name g_init g_repeat g_test g_expl ...**)**

NOTE: This is another, less general, form of do. It is evaluated by:

(1) Evaluating g_init.

(2) Lambda binding s_name to value of g_init.

(3) g_test is evaluated, and, if it is not nil, the do function returns with nil.

(4) The do body is evaluated beginning at g_expl.

(5) The repeat form is evaluated and stored in s_name.

(6) Go to step 3.

RETURNS: Nil.

(environment [l_when1 l_what1 l_when2 l_what2 ...])
(environment-maclisp [l_when1 l_what1 l_when2 l_what2 ...])
(environment-lmlisp [l_when1 l_what1 l_when2 l_what2 ...])

WHERE: The when's are a subset of (eval compile load), and the symbols have the same meaning as they do in 'eval-when'.

The what's may be:
(files file1 file2 ... fileN)
which insure that the named files are loaded. To see if file*i* is loaded, these functions look for a 'version' property under file*i*'s property list. In order to make this work to prevent multiple loading, you should put
(putprop 'myfile t 'version),
at the end of myfile.l.

Another acceptable form for a what is
(syntax type)
Where type is either maclisp, intlisp, ucilisp, or franzlisp.

SIDE EFFECT: *environment-maclisp* sets the environment to what 'liszt +m' generates.

environment-lmlisp sets up the Lisp machine environment. This is like maclisp but it has additional macros.

For these specialized environments, only the **files** clauses are useful.
(environment-maclisp (compile eval) (files foo bar))

RETURNS: The last list of files requested.

(err ['s_value [nil]])

RETURNS: Nothing (it never returns).

SIDE EFFECT: This causes an error, and, if this error is caught by an *errset* then that *errset* returns s_value instead of nil. If the second arg is given, then it must be nil (for MAClisp compatibility).

(error ['s_message1 ['s_message2]])

RETURNS: Nothing (it never returns).

SIDE EFFECT: s_message1 and s_message2 are *patom*ed if they are given and then *err* is called (with no arguments), which causes an error.

(errset g_expr [s_flag])

RETURNS: A list of one element that is the value resulting from evaluating g_expr. If an error occurs during the evaluation of g_expr, then the locus of control returns to the *errset,* which then returns nil (unless the error was caused by a call to *err* with a non-null argument).

SIDE EFFECT: S_flag is evaluated before g_expr is evaluated. If s_flag is not given, then it is assumed to be t. If an error occurs during the evaluation of g_expr, and s_flag was evaluated to a non-null value, then the error message associated with the error is printed before control returns to the errset.

(eval 'g_val ['x_bind-pointer]**)**

RETURNS: The result of evaluating g_val.

NOTE: The evaluator evaluates g_val in the following way:
If g_val is a symbol, then the evaluator returns its value. If g_val had never been assigned a value, then this causes an 'Unbound Variable' error. If x_bind-pointer is given, then the variable is evaluated with respect to that pointer. See *evalframe* for details on bind-pointers.

If g_val is of type value, then its value is returned. If g_val is of any other type than list, g_val is returned.

If g_val is a list object, then g_val is either a function call or array reference. Let g_car be the first element of g_val. g_car is continually evaluated until it results in a symbol with a non-null function binding or a non-symbol. Call the result: g_func.

G_func must be one of three types: list, binary, or array. If it is a list, then the first element of the list, which is called g_functype, must be either lambda, nlambda, macro, or lexpr. If g_func is a binary, then its discipline, which is called g_functype, is either lambda, nlambda, macro, or a string. If g_func is an array, then this form is evaluated specially. See Chapter 9 on arrays. If g_func is a list or binary, then g_functype determines how the arguments to this function, the cdr of g_val, are processed. If g_functype is a string, then this is a foreign function call. See §8.5 for more details.

If g_functype is lambda or lexpr, the arguments are evaluated (by calling *eval* recursively) and stacked. If g_functype is nlambda, then the argument list is stacked. If g_functype is macro, then the entire form, g_val, is stacked.

Next, the formal variables are lambda bound. The formal variables are the cadr of g_func. If g_functype is nlambda, lexpr, or macro, there should only be one formal variable. The values on the stack are lambda bound to the formal variables except in the case of a lexpr, where the number of actual arguments is bound to the formal variable.

After the binding is done, the function is invoked, either by jumping to the entry point in the case of a binary or by evaluating the list of forms beginning at cddr g_func. The result of this function invocation is returned as the value of the call to eval.

(evalframe 'x_pdlpointer**)**

RETURNS: An evalframe descriptor for the evaluation frame just before x_pdlpointer. If x_pdlpointer is nil, it returns the evaluation frame of the frame just before the current call to *evalframe*.

NOTE: An evalframe descriptor describes a call to *eval*, *apply*, or *funcall*. The form of the descriptor is
(type pdl-pointer expression bind-pointer np-index lbot-index),
where type is 'eval' if this describes a call to *eval* or 'apply' if this is a call to *apply* or *funcall*. pdl-pointer is a number that describes this context. It can be passed to *evalframe* to obtain the next descriptor and can be passed to *freturn* to cause a return from this context. bind-pointer is the size of variable binding stack when this evaluation began. The bind-pointer can be given as a second argument to *eval* in order to evaluate variables in the same context as this evaluation. If type is 'eval', then expression has the form *(function-name arg1 ...)*. If type is 'apply',

then expression has the form *(function-name (arg1 ...))*. np-index and lbot-index are pointers into the argument stack (also known as the *namestack* array) at the time of call. lbot-index points to the first argument; np-index points one beyond the last argument.
In order for there to be enough information for *evalframe* to return, you must call *(*rset t)*.

EXAMPLE: *(progn (evalframe nil))*
returns *(eval 2147478600 (progn (evalframe nil)) 1 8 7)*

(evalhook 'g_form 'su_evalfunc ['su_funcallfunc])

RETURNS: The result of evaluating g_form after lambda binding 'evalhook' to su_evalfunc, and, if it is given, lambda binding 'funcallhook' to su_funcallhook.

NOTE: As explained in §14.4, the function *eval* may pass the job of evaluating a form to a user 'hook' function when various switches are set. The hook function normally prints the form to be evaluated on the terminal and then evaluates it by calling *evalhook*. *Evalhook* does the lambda binding mentioned earlier and then calls *eval* to evaluate the form after setting an internal switch to tell *eval* not to call the user's hook function just this one time. This allows the evaluation process to advance one step and yet insure that further calls to *eval* cause traps to the hook function (if su_evalfunc is non-null).
In order for *evalhook* to work, *(*rset t)* and *(sstatus evalhook t)* must have been done previously.

(exec s_arg1 ...)

RETURNS: the result of forking and executing the command named by concatenating the s_arg*i* together with spaces in between.

(exece 's_fname ['l_args ['l_envir]])

RETURNS: The error code from the system if it was unable to execute the command s_fname with arguments l_args and with the environment set up as specified in l_envir. If this function is successful, it is not returned, instead the Lisp system is overlaided by the new command.

(freturn 'x_pdl-pointer 'g_retval)

RETURNS: g_retval from the context given by x_pdl-pointer.

NOTE: A pdl-pointer denotes a certain expression currently being evaluated. The pdl-pointer for a given expression can be obtained from *evalframe*.

(frexp 'f_arg)

RETURNS: A list cell *(exponent . mantissa)* that represents the given flonum.

NOTE: The exponent is a fixnum; the mantissa a 56 bit bignum. If you think of the the binary point occurring right after the high order bit of mantissa, then f_arg = 2^{exponent} * mantissa.

(funcall 'u_func ['g_arg1 ...])

RETURNS: the value of applying function u_func to the arguments g_arg*i* and then evaluating that result if u_func is a macro.

NOTE: If u_func is a macro or nlambda, then there should be only one g_arg. *funcall* is the function that the evaluator uses to evaluate lists. If *foo* is a lambda, lexpr, or array, then *(funcall 'foo 'a 'b 'c)* is equivalent to *(foo 'a 'b 'c)*. If *foo* is an nlambda, then *(funcall 'foo '(a b c))* is equivalent to *(foo a b c)*. Finally, if *foo* is a macro, then *(funcall 'foo '(foo a b c))* is equivalent to *(foo a b c)*.

(funcallhook 'l_form 'su_funcallfunc ['su_evalfunc])

RETURNS: the result of the following sequence of actions. First, it lambda-binds 'funcallhook' to su_funcallfunc and, if it is given, lambda binds 'evalhook' to su_evalhook. Then it proceeds to *funcall* the *(car l_form)* on the already evaluated arguments in the *(cdr l_form)*

NOTE: This function is designed to continue the evaluation process with as little work as possible after a funcallhook trap has occurred. It is for this reason that the form of l_form is unorthodox: its *car* is the name of the function to call and its *cdr* are a list of arguments to stack (without evaluating again) before calling the given function. After stacking the arguments but before calling *funcall*, an internal switch is set to prevent *funcall* from passing the job of funcalling to su_funcallfunc. If *funcall* is called recursively in funcalling l_form and if su_funcallfunc is non-null, then the arguments to *funcall* are actually given to su_funcallfunc (a lexpr) to be funcalled.
In order for *evalhook* to work, *(*rset t)* and *(sstatus evalhook t)* must have been done previously. A more detailed description of *evalhook* and *funcallhook* is given in Chapter 14.

(function u_func)

RETURNS: The function binding of u_func if it is a symbol with a function binding; otherwise, u_func is returned.

(getdisc 'y_func)

RETURNS: The discipline of the machine coded function -- lambda, nlambda, or macro.

(go g_labexp)

WHERE: g_labexp is either a symbol or an expression.

SIDE EFFECT: If g_labexp is an expression, that expression is evaluated and should result in a symbol. The locus of control moves to just following the symbol g_labexp in the current prog or do body.

NOTE: This is only valid in the context of a prog or do body. The interpreter and compiler allow non-local *go*'s, although the compiler does not allow a *go* to leave a function body. The compiler does not allow g_labexp to be an expression.

(if 'g_a 'g_b**)**
(if 'g_a 'g_b 'g_c ...**)**
(if 'g_a **then** 'g_b [...] [**elseif** 'g_c **then** 'g_d ...] [**else** 'g_e [...]**)**
(if 'g_a **then** 'g_b [...] [**elseif** 'g_c **thenret**] [**else** 'g_d [...]**)**

NOTE: The various forms of *if* are intended to be easily readable conditional statements -- to be used in place of *cond*. There are two varieties of *if*: with and without keywords. The keyword-less variety is inherited from common Maclisp usage. A keyword-less, two argument *if* is equivalent to a one-clause *cond*, i.e., (*cond* (a b)). Any other keyword-less *if* must have at least three arguments. The first two arguments are the first clause of the equivalent *cond*, and all remaining arguments are shoved into a second clause beginning with **t**. Thus, the second form of *if* is equivalent to

(*cond* (a b) (t c ...)).

The keyword variety has the following grouping of arguments: a predicate, a then-clause, and an optional else-clause. The predicate is evaluated, and if the result is non-nil, the then-clause is performed, in the sense described later. Otherwise, that is, the result of the predicate evaluation was precisely nil, the else-clause is performed.

Then-clauses are either consist entirely of the single keyword **thenret**, or start with the keyword **then**, and followed by at least one general expression. (These general expressions must not be one of the keywords.) To actuate a **thenret** means to cease further evaluation of the *if* and to return the value of the predicate just calculated. The performance of the longer clause means to evaluate each general expression in turn and then return the last value calculated.

The else-clause may begin with the keyword **else** and be followed by at least one general expression. The rendition of this clause is just like that of a then-clause. An else-clause may begin alternatively with the keyword **elseif** and be followed (recursively) by a predicate, then-clause, and optional else-clause. Evaluation of this clause, is just evaluation of an *if* form, with the same predicate, then- and else-clauses.

(I-throw-err 'l_token**)**

WHERE: l_token is the *cdr* of the value returned from a **catch* with the tag ER%unwind-protect.

RETURNS: Nothing (never returns in the current context).

SIDE EFFECT: The error or throw denoted by l_token is continued.

NOTE: This function is used to implement *unwind-protect* which allows the processing of a transfer of control though a certain context to be interrupted, a user function to be executed, and then the transfer of control to continue. The form of l_token is either
(t tag value) for a throw or
(nil type message valret contuab uniqueid [arg ...]) for an error.
This function is not to be used for implementing throws or errors and is only documented here for completeness.

(let l_args g_expl ... g_exprn)

RETURNS: The result of evaluating g_exprn within the bindings given by l_args.

NOTE: l_args is either nil (in which case *let* is just like *progn*) or it is a list of binding objects. A binding object is a list *(symbol expression)*. When a *let* is entered, all of the expressions are evaluated and then simultaneously lambda-bound to the corresponding symbols. In effect, a *let* expression is just like a lambda expression except that the symbols and their initial values are next to each other, making the expression easier to understand. There are some added features to the *let* expression: A binding object can just be a symbol, in which case the expression corresponding to that symbol is 'nil'. If a binding object is a list and the first element of that list is another list, then that list is assumed to be a binding template and *let* does a *desetq* on it.

(let* l_args g_expl ... g_expn)

RETURNS: The result of evaluating g_exprn within the bindings given by l_args.

NOTE: This is identical to *let* except the expressions in the binding list l_args are evaluated and bound sequentially instead of in parallel.

(lexpr-funcall 'g_function ['g_arg1 ...] 'l_argn)

NOTE: This is a cross between funcall and apply. The last argument must be a list (possibly empty). The elements of list arg are stacked and then the function is funcalled.

EXAMPLE: (lexpr-funcall 'list 'a '(b c d)) is the same as
(funcall 'list 'a 'b 'c 'd)

(listify 'x_count)

RETURNS: A list of x_count of the arguments to the current function (which must be a lexpr).

NOTE: Normally arguments 1 through x_count are returned. If x_count is negative then a list of last abs(x_count) arguments are returned.

(map 'u_func 'l_arg1 ...)

RETURNS: l_arg1

NOTE: The function u_func is applied to successive sublists of the l_arg*i*. All sublists should have the same length.

(mapc 'u_func 'l_arg1 ...)

RETURNS: l_arg1.

NOTE: The function u_func is applied to successive elements of the argument lists. All of the lists should have the same length.

(mapcan 'u_func 'l_arg1 ...)

RETURNS: nconc applied to the results of the functional evaluations.

NOTE: The function u_func is applied to successive elements of the argument lists. All sublists should have the same length.

(mapcar 'u_func 'l_arg1 ...)

RETURNS: A list of the values returned from the functional application.

NOTE: The function u_func is applied to successive elements of the argument lists. All sublists should have the same length.

(mapcon 'u_func 'l_arg1 ...)

RETURNS: nconc applied to the results of the functional evaluation.

NOTE: The function u_func is applied to successive sublists of the argument lists. All sublists should have the same length.

(maplist 'u_func 'l_arg1 ...)

RETURNS: A list of the results of the functional evaluations.

NOTE: The function u_func is applied to successive sublists of the arguments lists. All sublists should have the same length.

You may find the following summary table useful in remembering the differences between the six mapping functions:

	Value returned is		
Argument to functional is	l_arg1	list of results	*nconc* of results
elements of list	mapc	mapcar	mapcan
sublists	map	maplist	mapcon

(mfunction t_entry 's_disc)

RETURNS: A Lisp object of type binary composed of t_entry and s_disc.

NOTE: t_entry is a pointer to the machine code for a function, and s_disc is the discipline (e.g., lambda).

(oblist)

RETURNS: A list of all symbols on the oblist.

(or [g_arg1 ...])

RETURNS: The value of the first non-null argument or nil if all arguments evaluate to nil.

NOTE: Evaluation proceeds left to right and stops as soon as one of the arguments evaluates to a non-null value.

(prog l_vrbls g_exp1 ...)

RETURNS: The value explicitly given in a return form or else nil if no return is done by the time the last g_exp*i* is evaluated.

NOTE: The local variables are lambda-bound to nil, then the g_exp*i* are evaluated from left to right. This is a prog body (obviously) and this means that any symbols seen are not evaluated, but are treated as labels. This also means that return's and go's are allowed.

(prog1 'g_exp1 ['g_exp2 ...])

RETURNS: g_exp1

(prog2 'g_exp1 'g_exp2 ['g_exp3 ...])

RETURNS: g_exp2

NOTE: The forms are evaluated from left to right and the value of g_exp2 is returned.

(progn 'g_exp1 ['g_exp2 ...])

RETURNS: The last g_exp*i*

(progv 'l_locv 'l_initv g_exp1 ...)

WHERE: l_locv is a list of symbols and l_initv is a list of expressions.

RETURNS: The value of the last g_exp*i* evaluated.

NOTE: The expressions in l_initv are evaluated from left to right and then lambda-bound to the symbols in l_locv. If there are too few expressions in l_initv, then the missing values are assumed to be nil. If there are too many expressions in l_initv, then the extra ones are ignored (although they are evaluated). Then the g_exp*i* are evaluated left to right. The body of a progv is like the body of a progn, it is ***not*** a prog body. (C.f. *let*)

(purcopy 'g_exp)

RETURNS: A copy of g_exp with new pure cells allocated wherever possible.

NOTE: Pure space is never swept up by the garbage collector, so this should only be done on expressions that are not likely to become garbage in the future. In certain cases, data objects in pure space become read-only after a ***dumplisp,*** and then an attempt to modify the object results in an illegal memory reference.

(purep 'g_exp)

RETURNS: t iff the object g_exp is in pure space.

(putd 's_name 'u_func)

RETURNS: u_func

SIDE EFFECT: This sets the function binding of symbol s_name to u_func.

(return ['g_val])

RETURNS: g_val (or nil if g_val is not present) from the enclosing prog or do body.

NOTE: This form is only valid in the context of a prog or do body.

(selectq 'g_key-form [l_clause1 ...])

NOTE: This function is just like *caseq* (see earlier), except that the symbol **otherwise** has the same semantics as the symbol **t**, when used as a comparator.

(setarg 'x_argnum 'g_val)

WHERE: x_argnum is greater than zero and less than or equal to the number of arguments to the lexpr.

RETURNS: g_val

SIDE EFFECT: The lexpr's x_argnum'th argument is set to g-val.

NOTE: This can only be used within the body of a lexpr.

(throw 'g_val [s_tag])

WHERE: If s_tag is not given, it is assumed to be nil.

RETURNS: The value of *(*throw 's_tag 'g_val)*.

(*throw 's_tag 'g_val)

RETURNS: g_val from the first enclosing catch with the tag s_tag or with no tag at all. Thus the value is accompanied by a change in control.

NOTE: This is used in conjunction with *catch to cause a clean jump to an enclosing context.

(unwind-protect g_protected [g_cleanup1 ...])

RETURNS: The result of evaluating g_protected.

NOTE: Normally g_protected is evaluated and its value remembered, then the g_cleanup*i* are evaluated, and, finally, the saved value of g_protected is returned. If something should happen when evaluating g_protected which causes control to pass through g_protected, and, thus, through the call to the unwind-protect, then the g_cleanup*i* is still evaluated. This is useful if g_protected does something sensitive which must be cleaned up whether or not g_protected completes itself. Programs which 'temporarily' mess up a structure and then straighten the structure can use this scheme to protect the straightening-up process from being cut off by a keyboard interrupt.

CHAPTER 5

Input/Output

The following functions are used to read from and write to external devices (e.g. files) and programs through pipes. All I/O goes through the Lisp data type called the port. A port may be open for either reading or writing but usually not both simultaneously (see *fileopen*). There are only a limited number of ports (20) and they are not reclaimed unless they are *closed*. All ports are reclaimed by a *resetio* call, but this drastic action is not necessary if the program closes ports that it uses.

If a port argument is not supplied to a function that requires one, or if a bad port argument (such as nil) is given, then FRANZ LISP uses the default port according to this scheme: if input is being done, then the default port is the value of the symbol **piport** and, if output is being done, then the default port is the value of the symbol **poport**. Furthermore, if the value of piport or poport is not a valid port, then the standard input or standard output is used, respectively.

The standard input and standard output are usually the keyboard and terminal display unless your job is running in the background and its input or output is connected to a pipe. All output that goes to the standard output also goes to the port **ptport,** if it is a valid port. Output destined for the standard output does not reach the standard output if the symbol **^w** is non-nil, although it still goes to **ptport** if **ptport** is a valid port.

(cfasl 'st_file 'st_entry 'st_funcname ['st_disc ['st_library]])

RETURNS: T

SIDE EFFECT: This is used to load in a foreign function (see §8.4). The object file st_file is loaded into the Lisp system. St_entry should be an entry point in the file just loaded. The function binding of the symbol s_funcname is set to point to st_entry so that, when the Lisp function s_funcname is called, st_entry is run. st_disc is the discipline to be given to s_funcname. st_disc defaults to "subroutine" if it is not given or if it is given as nil. If st_library is non-null, then after st_file is loaded, the libraries given in st_library are searched to resolve external references. The form of st_library should be something like "+llibname". The C library (" +lclib ") is always searched so that when loading in a C file, you probably will not need to specify a library.

NOTE: This function may be used to load the output of the assembler, C compiler, Fortran compiler, and Pascal compiler but NOT the Lisp compiler. Use *fasl* for that. If a file has more than one entry point, then use *getaddress* to locate and setup other foreign functions.
It is an error to load in a file that has a global entry point of the same name as a global entry point in the running Lisp. As soon as you load in a file with *cfasl*, its global entry points become part of the Lisp's entry points. Thus, you cannot *cfasl* in the same file twice unless you use *removeaddress* to change certain global entry points to local entry points.

(close 'p_port)

RETURNS: T

SIDE EFFECT: The specified port is drained and closed, releasing the port.

NOTE: The standard defaults are not used in this case since you probably never want to close the standard output or standard input.

(cprintf 'st_format 'xfst_val ['p_port])

RETURNS: xfst_val

SIDE EFFECT: The operating system formatted output function printf is called with arguments st_format and xfst_val. If xfst_val is a symbol, then its print name is passed to printf. The format string may contain characters that are printed literally, and it may contain special formatting commands preceded by a percent sign. The complete set of formatting characters is described in the operating system manual. Some useful ones are %d for printing a fixnum in decimal, %f or %e for printing a flonum, and %s for printing a character string (or print name of a symbol).

EXAMPLE: *(cprintf "Pi equals %f" 3.14159)* prints 'Pi equals 3.14159'

(drain ['p_port])

RETURNS: nil

SIDE EFFECT: If this is an output port, then the characters in the output buffer are all sent to the device. If this is an input port, then all pending characters are flushed. The default port for this function is the default output port.

(fasl 'st_name ['st_mapf ['g_warn]])

WHERE: st_mapf and g_warn default to nil.

RETURNS: T if the function succeeded, nil otherwise.

SIDE EFFECT: This function is designed to load in an object file generated by the Lisp compiler Liszt. File names for object files usually end in '.o', so *fasl* append '.o' to st_name, if it is not already present. If st_mapf is non-nil, then it is the name of the map file to create. *Fasl* writes in the map file the names and addresses of the functions it loads and defines. Normally, the map file is created (i.e. truncated if it exists), but if *(sstatus appendmap t)* is done, then the map file is appended. If g_warn is non-nil and if a function is loaded from the file that is already defined, then a warning message is printed.

NOTE: *fasl* only looks in the current directory for the file to load. The function *load* looks through a user-supplied search path and calls *fasl* if it finds a file with the same root name and a '.o' extension. In most cases, you should use the function *load* rather than calling *fasl* directly.

(filepos 'p_port ['x_pos])

RETURNS: The current position in the file if x_pos is not given or else x_pos if x_pos is given.

SIDE EFFECT: If x_pos is given, the next byte to be read or written to the port is at position x_pos.

(filestat 'st_filename)

RETURNS: A vector containing various numbers that the operating system assigns to files. If the file does not exist, an error is invoked. Use *probef* to determine if the file exists.

NOTE: The individual entries can be accessed by mnemonic functions of the form filestat:*field*, where field may be any of: dev, ino, mode, mtime, nlink, size, type, or uid. See the operating system programmers manual for a more detailed description of these quantities.

(flatc 'g_form ['x_max])

RETURNS: The number of characters required to print g_form using *patom*. If x_max is given and, if *flatc* determines that it returns a value greater than x_max, then it gives up and returns the current value it has computed. This is useful if you just want to see if an expression is larger than a certain size.

(flatsize 'g_form ['x_max])

RETURNS: The number of characters required to print g_form using *print*. The meaning of x_max is the same as for flatc.

NOTE: Currently this just *explode*'s g_form and checks its length.

(fseek 'p_port 'x_offset 'x_flag)

RETURNS: The position in the file after the function is performed.

SIDE EFFECT: this function positions the read/write pointer before a certain byte in the file. If x_flag is 0 then the pointer is set to x_offset bytes from the beginning of the file. If x_flag is 1 then the pointer is set to x_offset bytes from the current location in the file. If x_flag is 2 then the pointer is set to x_offset bytes from the end of the file.

(infile 's_filename)

RETURNS: A port ready to read s_filename.

SIDE EFFECT: This tries to open s_filename, and, if it cannot or if there are no ports available, it gives an error message.

NOTE: To allow your program to continue on a file-not-found error, you can use something like:

```
(cond ((null (setq myport (car (errset (infile name) nil))))
          (patom "couldn't open the file")))
```

which sets myport to the port to read from if the file exists or prints a message if it could not open it and also sets myport to nil. To simply determine if a file exists, use *probef*.

(load 's_filename ['st_map ['g_warn]])

RETURNS: T

NOTE: The function of *load* has changed since previous releases of FRANZ LISP and the following description should be read carefully.

SIDE EFFECT: *load* now serves the function of both *fasl* and the old *load*. *Load* searches a user-defined search path for a Lisp source or object file with the filename s_filename (with the extension .l or .o added as appropriate). The search path that *load* uses is the value of *(status load-search-path)*. The default is (.| /lisp/lib), which means: look in the current directory first and then /lib/lisp. The file that *load* looks for depends on the last two characters of s_filename. If s_filename ends with ".l", then *load* only looks for a file name s_filename and assumes that this is a FRANZ LISP source file. If s_filename ends with ".o", then *load* only looks for a file named s_filename and assumes that this is a FRANZ LISP object file to be *fasl*ed in. Otherwise, *load* first looks for s_filename.o, then s_filename.l, and, finally, s_filename itself. If it finds s_filename.o, it assumes that this is an object file; otherwise, it assumes that it is a source file. An object file is loaded using *fasl* and a source file is loaded by reading and evaluating each form in the file. The optional arguments st_map and g_warn are passed to *fasl* should *fasl* be called.

NOTE: *load* requires a port to open the file s_filename. It then lambda binds the symbol piport to this port and reads and evaluates the forms.

(makereadtable ['s_flag])

WHERE: If s_flag is not present it is assumed to be nil.

RETURNS: A readtable equal to the original readtable if s_flag is non-null, or else equal to the current readtable. See Chapter 7 for a description of readtables and their uses.

(msg [l_option ...] ['g_msg ...])

NOTE: This function is intended for printing short messages. Any of the arguments or options presented can be used any number of times in any order. The messages themselves (g_msg) are evaluated, and then they are transmitted to *patom*. Typically, they are strings, which evaluate to themselves. The options are interpreted specially:

msg Option Summary

(P p_portname)	Causes subsequent output to go to the port p_portname; port should be opened previously.
B	Print a single blank.
(B 'n_b)	Evaluate n_b and print that many blanks.
N	Print a single newline by calling *terpr*.
(N 'n_n)	Evaluate n_n and transmit that many newlines to the stream.
D	*drain* the current port.

(nwritn ['p_port]**)**

RETURNS: The number of characters in the buffer of the given port but not yet written out to the file or device. The buffer is flushed automatically when filled or when *terpr* is called.

(outfile 's_filename ['st_type]**)**

RETURNS: A port or nil

SIDE EFFECT: This opens a port to write s_filename. If st_type is given and if it is a symbol or string whose name begins with 'a', then the file is opened in append mode; that is, the current contents are not lost, and the next data is written at the end of the file. Otherwise, the file opened is truncated by *outfile* if it existed beforehand. If there are no free ports, outfile returns nil. If one cannot write on s_filename, an error is signalled.

(patom 'g_exp ['p_port]**)**

RETURNS: g_exp

SIDE EFFECT: g_exp is printed to the given port or the default port. If g_exp is a symbol or string, the print name is printed without any escape characters around special characters in the print name. If g_exp is a list, then *patom* has the same effect as *print*

(pntlen 'xfs_arg**)**

RETURNS: The number of characters needed to print xfs_arg.

(portp 'g_arg)

RETURNS: T iff g_arg is a port.

(pp [l_option] s_name1 ...)

RETURNS: T

SIDE EFFECT: If s_name*i* has a function binding, it is pretty-printed; otherwise, if s_name*i* has a value, then that is pretty-printed. Normally, the output of the pretty-printer goes to the standard output port poport. The options allow you to redirect it.

PP Option Summary

(F s_filename)	Direct future printing to s_filename.
(P p_portname)	Causes output to go to the port p_portname; port should be opened previously.
(E g_expression)	Evaluate g_expression and do not print.

(princ 'g_arg ['p_port])

EQUIVALENT TO: patom.

(print 'g_arg ['p_port])

RETURNS: Nil

SIDE EFFECT: Prints g_arg on the port p_port or the default port.

(probef 'st_file)

RETURNS: T iff the file st_file exists.

NOTE: Just because it exists doesn't mean you can read it.

(pp-form 'g_form ['p_port])

RETURNS: T

SIDE EFFECT: g_form is pretty-printed to the port p_port (or poport if p_port is not given). This is the function that *pp* uses. *pp-form* does not look for function definitions or values of variables, it just prints out the form it is given.

NOTE: This is useful as a top-level-printer. See *top-level* in Chapter 6.

(ratom ['p_port ['g_eof]])

RETURNS: The next atom read from the given or default port. On end of file, g_eof (default nil) is returned.

(read ['p_port ['g_eof]])

RETURNS: The next Lisp expression read from the given or default port. On end of file, g_eof (default nil) is returned.

NOTE: An error occurs if the reader is given an ill formed expression. The most common error is too many right parentheses. (Note that this is not considered an error in Maclisp).

(readc ['p_port ['g_eof]])

RETURNS: The next character read from the given or default port. On end of file, g_eof (default nil) is returned.

(readlist 'l_arg)

RETURNS: The Lisp expression read from the list of characters in l_arg.

(removeaddress 's_name1 ['s_name2 ...])

RETURNS: Nil

SIDE EFFECT: The entries for the s_name*i* in the Lisp symbol table are removed. This is useful if you wish to *cfasl* in a file twice, since it is illegal for a symbol in the file you are loading to already exist in the Lisp symbol table.

(resetio)

RETURNS: Nil

SIDE EFFECT: All ports except the standard input, output, and error are closed.

(setsyntax 's_symbol 's_synclass ['ls_func])

RETURNS: T

SIDE EFFECT: This sets the code for s_symbol to sx_code in the current readtable. If s_synclass is *macro* or *splicing,* then ls_func is the associated function. See Chapter 7 on the reader for more details.

(sload 's_file)

SIDE EFFECT: The file s_file (in the current directory) is opened for reading, and each form is read, printed, and evaluated. If the form is recognizable as a function definition, only its name is printed; otherwise, the whole form is printed.

NOTE: This function is useful when a file refuses to load because of a syntax error and you would like to determine where the error is.

(tab 'x_col ['p_port])

SIDE EFFECT: Enough spaces are printed to put the cursor on column x_col. If the cursor is beyond x_col to start with, a *terpr* is done first.

(terpr ['p_port])

RETURNS: Nil

SIDE EFFECT: A terminate line character sequence is sent to the given port or the default port. This also drains the port.

(terpri ['p_port])

EQUIVALENT TO: terpr.

(tyi ['p_port])

RETURNS: The fixnum representation of the next character read. On end of file, -1 is returned.

(tyipeek ['p_port])

RETURNS: The fixnum representation of the next character to be read.

NOTE: This does not cause an official 'read' of the character, it just peeks at it and returns the value which would be returned if it were read. (It 'peeks'.)

(tyo 'x_char ['p_port])

RETURNS: x_char.

SIDE EFFECT: The character whose fixnum representation is x_code is printed as a character on the given output port or the default output port.

(untyi 'x_char ['p_port])

SIDE EFFECT: x_char is put back in the input buffer so a subsequent *tyi* or *read* reads it first.

NOTE: A maximum of one character may be put back.

(zapline)

RETURNS: nil

SIDE EFFECT: All characters up to and including the line termination character are read and discarded from the last port used for input.

NOTE: This is used as the macro function for the semicolon character when it acts as a comment character.

CHAPTER 6

System Functions

This chapter describes the functions used to interact with internal components of the Lisp system and operating system.

(allocate 's_type 'x_pages)

WHERE: s_type is one of the FRANZ LISP data types described in §1.3.

RETURNS: x_pages.

SIDE EFFECT: FRANZ LISP attempts to allocate x_pages of type s_type. If there aren't x_pages of memory left, no space is allocated and an error occurs. The storage that is allocated is not given to the caller, instead it is added to the free storage list of s_type. The functions *segment* and *small-segment* allocate blocks of storage and return it to the caller.

(argv 'x_argnumb)

RETURNS: A symbol whose pname is the x_argnumb*th* argument (starting at 0) on the command line that invoked the current Lisp.

NOTE: If x_argnumb is less than zero, a fixnum whose value is the number of arguments on the command line is returned. *(argv 0)* returns the name of the Lisp you are running.

(baktrace)

RETURNS: nil

SIDE EFFECT: The Lisp runtime stack is examined and the name of (most) of the functions currently in execution are printed, most active first.

NOTE: This occasionally misses the names of compiled Lisp functions due to incomplete information on the stack. If you are tracing compiled code, then *baktrace* is not able to interpret the stack unless *(sstatus translink nil)* was done. See the function *showstack* for another way of printing the Lisp runtime stack. This misspelling is from Maclisp.

(chdir 's_path)

RETURNS: t iff the system call succeeds.

SIDE EFFECT: The current directory is set to s_path. Among other things, this affects the default location where the input/output functions look for and create files.

NOTE: *chdir* follows the standard operating system conventions. If s_path does not begin with a slash, the default path is changed to the current path with s_path appended.

(command-line-args)

RETURNS: A list of the arguments typed on the command line either to the Lisp interpreter, or saved Lisp dump, or application compiled with the autorun option (liszt +r).

(deref 'x_addr)

RETURNS: The contents of x_addr, when thought of as a longword memory location.

NOTE: This may be useful in constructing arguments to C functions out of 'dangerous' areas of memory.

(dumplisp s_name)

RETURNS: nil

SIDE EFFECT: The current Lisp is dumped to the named file. When s_name is executed, you are in a Lisp in the same state as when the dumplisp was done.

NOTE: dumplisp fails if you try to write over the current running file. The operating system does not allow you to modify the file you are running.

(eval-when l_time g_exp1 ...)

SIDE EFFECT: l_time may contain any combination of the symbols *load*, *eval*, and *compile*. The effects of load and compile are discussed in §12.3.2.1 on the compiler. If eval is present, however, this simply means that the expressions g_exp1, and so on, are evaluated from left to right. If eval is not present, the forms are not evaluated.

(exit ['x_code])

RETURNS: Nothing (it never returns a lisp value).

SIDE EFFECT: The Lisp system dies with exit code x_code or 0 if x_code is not specified.

(fake 'x_addr)

RETURNS: The Lisp object at address x_addr.

NOTE: This is intended to be used by people debugging the Lisp system.

(fork)

RETURNS: nil to the child process and the process number of the child to the parent.

SIDE EFFECT: A copy of the current Lisp system is made in memory, and both Lisp systems now begin to run. This function can be used interactively to temporarily save the state of Lisp (as shown later), but you must be careful that only one of the Lisp's interacts with the terminal after the fork. The *wait* function is useful for this.

```
-> (setq foo 'bar)          ;; Set a variable.
bar
-> (cond ((fork)(wait)))    ;; Duplicate the Lisp system and
nil                                ;; make the parent wait.
-> foo                             ;; Check the value of the variable.
bar
-> (setq foo 'baz)          ;; Give it a new value.
baz
-> foo                             ;; Make sure it worked.
baz
-> (exit)                          ;; Exit the child.
(5274 . 0)                         ;; The wait function returns this.
-> foo                             ;; Check to make sure parent was
bar                                ;; not modified.
```

(gc)

RETURNS: nil

SIDE EFFECT: This causes a garbage collection.

NOTE: The function *gcafter* is not called automatically after this function finishes. Normally, the user does not have to call *gc* since garbage collection occurs automatically whenever internal free lists are exhausted.

(gcafter s_type)

WHERE: s_type is one of the FRANZ LISP data types listed in §1.3.

NOTE: This function is called by the garbage collector after a garbage collection that was caused by running out of data type s_type. This function should determine if more space need be allocated, and, if so, should allocate it. There is a default gcafter function, but if you want control over space allocation, you can define your own. However, be sure that it is an nlambda.

(hashtabstat)

RETURNS: A list of fixnums representing the number of symbols in each 'bucket' of the oblist.

NOTE: The oblist is organized as a hash table of linked lists (the buckets). An ideal distribution of identifiers would place about the same number of symbols in each bucket. A very poor distribution would make reading slow.

(include s_filename)

RETURNS: nil

SIDE EFFECT: The given filename is *load*ed into the Lisp system.

NOTE: This is similar to load except that the argument is not evaluated. Include means something special to the compiler.

(include-if 'g_predicate s_filename)

RETURNS: nil

SIDE EFFECT: This has the same effect as include but is only actuated if the predicate is non-nil.

(includef 's_filename)

RETURNS: nil

SIDE EFFECT: This is the same as *include* except that the argument is evaluated.

(includef-if 'g_predicate s_filename)

RETURNS: nil

SIDE EFFECT: This has the same effect as includef but is only actuated if the predicate is non-nil.

(maknum 'g_arg)

RETURNS: The address of its argument converted into a fixnum.

(opval 's_arg ['g_newval])

RETURNS: The value associated with s_arg before the call.

SIDE EFFECT: If g_newval is specified, the value associated with s_arg is changed to g_newval.

NOTE: *opval* keeps track of storage allocation. If s_arg is one of the data types, then *opval* returns a list of three fixnums representing the number of items of that type in use, the number of pages allocated, and the number of items of that type per page. You should never try to change the value that *opval* associates with a data type using *opval*
If s_arg is *pagelimit,* then *opval* returns (and sets if g_newval is given) the maximum amount of Lisp data pages it allocates. This limit should remain small unless you know your program requires lots of space because this limit catches programs in infinite loops, which gobble up memory.

(*process 'st_command ['g_readp ['g_writep]])

RETURNS: Either a fixnum if one argument is given, or a list of two ports and a fixnum if two or three arguments are given.

NOTE: **process* starts another process by passing st_command to the shell. (/bin/shell). If only one argument is given to **process*, **process* waits for the new process to die and then returns the exit code of the new process. If more than two or three arguments are given, **process* starts the process and then returns a list which, depending on the value of g_readp and g_writep, may contain i/o ports for communcating with the new process. If g_writep is non-null, then a port is created that the Lisp program can use to send characters to the new process. If g_readp is non-null, then a port is created that the Lisp program can use to read characters from the new process. The value returned by **process* is (readport writeport pid), where readport and writeport are either nil or a port based on the value of g_readp and g_writep. Pid is the process id of the new process. Since it is hard to remember the order of g_readp and g_writep, the functions **process-send* and **process-receive* are written to perform the common functions.

(*process-receive 'st_command**)**

RETURNS: A port that can be read.

SIDE EFFECT: The command st_command is given to the shell, and it is started running in the background. The output of that command is available for reading via the port returned. The input of the command process is set to /dev/null.

(*process-send 'st_command**)**

RETURNS: A port that can be written to.

SIDE EFFECT: The command st_command is given to the shell, and it is started runing in the background. The Lisp program can provide input for that command by sending characters to the port returned by this function. The output of the command process is set to /dev/null.

(process s_pgrm [s_frompipe s_topipe]**)**

RETURNS: If the optional arguments are not present, a fixnum that is the exit code when s_prgm dies. If the optional arguments are present, it returns a fixnum that is the process id of the child.

NOTE: This command is obsolete. New programs should use one of the **process* commands given earlier.

SIDE EFFECT: If s_frompipe and s_topipe are given, they are bound to ports that are pipes that direct characters from FRANZ LISP to the new process and to FRANZ LISP from the new process respectively. *Process* forks a process named s_prgm and waits for it to die if and only if there are no pipe arguments given.

(ptime)

RETURNS: A list of two elements. The first is the amount of processor time used by the Lisp system so far, and the second is the amount of time used by the garbage collector so far.

NOTE: The time is measured in those units used by the *times*(2) system call, usually 60*ths* of a second. The first number includes the second number. The amount of time used by garbage collection is not recorded until the first call to ptime. This is done to prevent overhead when the user is not interested in garbage collection times.

(reset)

SIDE EFFECT: The Lisp runtime stack is cleared and the system restarts at the top level.

(*rset 'g_flag)

RETURNS: g_flag

SIDE EFFECT: If g_flag is non-nil, then the Lisp system maintains extra information about calls to *eval* and *funcall*. This record keeping slows down the evaluation, but this is required for the functions *evalhook*, *funcallhook*, and *evalframe* to work. To debug compiled Lisp code, the transfer tables should be unlinked: *(sstatus translink nil)*

(segment 's_type 'x_size)

WHERE: s_type is one of the data types given in §1.3.

RETURNS: A segment of contiguous lispvals of type s_type.

NOTE: In reality, *segment* returns a new data cell of type s_type and allocates space for x_size − 1 more s_type's beyond the one returned. *Segment* always allocates new space and does so in 512 byte chunks. If you ask for 2 fixnums, segment actually allocates 128 of them, thus, wasting 126 fixnums. The function *small-segment* is a smarter space allocator and should be used whenever possible.

(shell)

RETURNS: The exit code of the shell when it dies.

SIDE EFFECT: This forks a new shell and returns when the shell dies.

(showstack)

RETURNS: nil

SIDE EFFECT: All forms currently in evaluation are printed, beginning with the most recent. For compiled code, showstack reveals only the function name, and it may miss some functions which you might expect from interpreted code.

(signal 'x_signum 's_name)

RETURNS: nil if no previous call to signal has been made, if a previous call has occurred, it will return the previously installed s_name.

SIDE EFFECT: This identifies the function named s_name to handle the signal number x_signum. If s_name is nil, the signal is ignored. Presently, only four operating system signals are caught. They and their numbers are: Interrupt(2), Floating exception(8), Alarm(14), and Hang-up(1).

(sizeof 'g_arg)

RETURNS: The number of bytes required to store one object of type g_arg, encoded as a fixnum.

(small-segment 's_type 'x_cells)

WHERE: s_type is one of fixnum, flonum, and value.

RETURNS: A segment of x_cells data objects of type s_type.

SIDE EFFECT: This may call *segment* to allocate new space, or it may be able to fill the request on a page already allocated. The value returned by *small-segment* is usually stored in the data subpart of an array object.

(sstatus g_type g_val)

RETURNS: g_val

SIDE EFFECT: If g_type is not one of the special sstatus codes described in the next few pages, this simply sets g_val as the value of status type g_type in the system status property list.

(sstatus appendmap g_val)

RETURNS: g_val

SIDE EFFECT: If g_val is non-null, when *fasl* is told to create a load map, it appends to the file name given in the *fasl* command rather than creating a new map file. The initial value is nil.

(sstatus automatic-reset g_val)

RETURNS: g_val

SIDE EFFECT: If g_val is non-null when an error occurs that no one wants to handle, a *reset* is done instead of entering a primitive internal break loop. The initial value is t.

(sstatus chainatom g_val)

RETURNS: g_val

SIDE EFFECT: If g_val is non-nil and a *car* or *cdr* of a symbol is done, then nil is returned instead of an error being signaled. This only affects the interpreter not the compiler. The initial value is nil.

(sstatus dumpcore g_val)

RETURNS: g_val

SIDE EFFECT: If g_val is nil, FRANZ LISP tells the operating system that a segmentation violation or bus error should cause a core dump. If g_val is non-nil then FRANZ LISP catches those errors and prints a message advising the user to reset.

NOTE: The initial value for this flag is nil, and only those knowledgeable of the inner characteristics of the Lisp system should ever set this flag non-nil.

(sstatus evalhook g_val)

RETURNS: g_val

SIDE EFFECT: When g_val is non-nil, this enables the evalhook and funcallhook traps in the evaluator. See §14.4 for more details.

(sstatus feature g_val)

RETURNS: g_val

SIDE EFFECT: g_val is added to the *(status features)* list.

(sstatus ignoreeof g_val)

RETURNS: g_val

SIDE EFFECT: If g_val is non-null when an end of file (CNTL-D on the operating system) is typed to the standard top-level interpreter, it is ignored rather than cause the Lisp system to exit. If the the standard input is a file or pipe, then this has no effect. An EOF always causes Lisp to exit. The initial value is nil.

(sstatus nofeature g_val)

RETURNS: g_val

SIDE EFFECT: g_val is removed from the status features list if it is present.

(sstatus translink g_val)

RETURNS: g_val

SIDE EFFECT: If g_val is nil, then all transfer tables are cleared and further calls through the transfer table do not cause the fast links to be set up. If g_val is the symbol *on,* then all possible transfer table entries are linked and the flag is set to cause fast links to be set up dynamically. Otherwise, all that is done is to set the flag to cause fast links to be set up dynamically. The initial value is nil.

NOTE: For a discussion of transfer tables, see §12.8.

(sstatus uctolc g_val)

RETURNS: g_val

SIDE EFFECT: If g_val is not nil, then all unescaped capital letters in symbols read by the readeris converted to lower case.

NOTE: This allows FRANZ LISP to be compatible with single case Lisp systems (e.g. Maclisp, Interlisp and UCILisp).

(status g_code)

RETURNS: The value associated with the status code g_code if g_code is not one of the special cases given later

(status ctime)

RETURNS: A symbol whose print name is the current time and date.

EXAMPLE: *(status ctime)* = |Sun Jun 29 16:51:26 1980|

NOTE: This has been made obsolete by *time-string,* described later.

(status feature g_val)

RETURNS: T iff g_val is in the status features list.

(status features)

RETURNS: The value of the features code, which is a list of features that are present in this system. You add to this list with *(sstatus feature 'g_val)* and test if feature g_feat is present with *(status feature 'g_feat)*.

(status isatty)

RETURNS: T iff the standard input is a terminal.

(status localtime)

RETURNS: A list of fixnums representing the current time.

EXAMPLE: *(status localtime)* = (3 51 13 31 6 81 5 211 1)
means 3*rd* second, 51*st* minute, 13*th* hour (1 p.m), 31*st* day, month 6 (0 = January), year 81 (0 = 1900), day of the week 5 (0 = Sunday), 211*th* day of the year with daylight savings time in effect.

(status syntax s_char)

NOTE: This function should not be used. See the description of *getsyntax,* in Chapter 7, for a replacement.

(status undeffunc)

RETURNS: A list of all functions that transfer table entries point to but that are not defined at this point.

NOTE: Some of the undefined functions listed could be arrays which are not yet created.

(status version)

RETURNS: A string that is the current Lisp version name.

EXAMPLE: *(status version)* = "Franz Lisp, Opus 41.10"

(sys:access 'st_filename 'x_mode)
(sys:chmod 'st_filename 'x_mode)
(sys:getpid)
(sys:link 'st_oldfilename 'st_newfilename)
(sys:time)
(sys:unlink 'st_filename)

NOTE: The actual system call numbers may vary among different operating systems. If you are concerned about portability, you may wish to use this group of functions. Another advantage is that tilde-expansion is performed on all filename arguments. These functions do what is described in the system call section of your operating system manual.

sys:getpwnam returns a vector of four entries from the password file. These entries are: the user name, user id, group id, and home directory.

(time-string ['x_seconds]**)**

RETURNS: An ASCII string giving the time and date that was x_seconds after operating system's idea of creation (Midnight, Jan 1, 1970 GMT). If no argument is given, time-string returns the current date. This supplants *(status ctime)*, and may be used to make the results of *filestat* more intelligible.

(top-level)

RETURNS: Nothing (it never returns)

NOTE: This function is the top-level read-eval-print loop. It never returns any value. Its main use is that if you redefine it and do a (reset), then the redefined (top-level) is then invoked. The default top-level for FRANZ LISP allows you to specify your own printer or reader by binding the symbols **top-level-printer** and **top-level-reader**. You can let the default top-level do most of the drudgery in catching *reset*'s, and reading in .lisprc files by binding the symbol **user-top-level** to a routine that concerns itself only with the read-eval-print loop.

(wait)

RETURNS: A dotted pair *(processid . status)* when the next child process dies.

CHAPTER 7

The Lisp Reader

7.1. Introduction

The *read* function is responsible for converting a stream of characters into a Lisp expression. *Read* is table driven and the table it uses is called a *readtable.* The *print* function does the inverse of *read*, it converts a Lisp expression into a stream of characters. Typically, the conversion is done in such a way that if a stream of characters is read by *read*, the result is an expression equal to the one *print* is given. *Print* must also refer to the readtable in order to determine how to format its output. The *explode* function, which returns a list of characters rather than printing them, must also refer to the readtable.

A readtable is created with the *makereadtable* function, modified with the *setsyntax* function and interrogated with the *getsyntax* function. The structure of a readtable is hidden from the user -- a readtable should only be manipulated with the three functions mentioned earlier.

There is one distinguished readtable called the *current readtable* whose value determines what *read*, *print*, and *explode* do. The current readtable is the value of the symbol *readtable*. Thus, it is possible to rapidly change the current syntax by lambda-binding a different readtable to the symbol *readtable.* When the binding is undone, the syntax reverts to its old form.

7.2. Syntax Classes

The readtable describes how each of the 128 ASCII characters should be treated by the reader and printer. Each character belongs to a *syntax class*, which has three properties:

character class -
: Tells what the reader should do when it sees this character. There are a large number of character classes. They are described later.

separator -
: Most types of tokens the reader constructs are one character long. Four token types have an arbitrary length: number (1234), symbol print name (franz), escaped symbol print name (|franz|), and string ("franz"). The reader can easily determine when it has come to the end of one of the last two types: it just looks for the matching delimiter (| or "). When the reader is reading a number or symbol print name, it stops reading when it comes to a character with the *separator* property. The separator character is pushed back into the input stream and is the first character read when the reader is called again.

escape -
: Tells the printer when to put escapes in front of, or around, a symbol whose print name contains this character. There are three possibilities: (1) always escape a symbol with this character in it, (2) only escape a symbol if this is the only

character in the symbol, and (3) only escape a symbol if this is the first character in the symbol. (Note that the printer always escapes a symbol which, if printed out, looks like a valid number.)

When the Lisp system is built, Lisp code is added to a C-coded kernel and the result becomes the standard Lisp system. The readtable present in the C-coded kernel, called the *raw readtable*, contains the bare necessities for reading in Lisp code. During the construction of the complete Lisp system, a copy is made of the raw readtable and then the copy is modified by adding macro characters. The result is what is called the *standard readtable*. When a new readtable is created with *makereadtable,* a copy is made of either the raw readtable or the current readtable, which is likely to be the standard readtable.

7.3. Reader Operations

The reader has a very simple algorithm. It is either *scanning* for a token, *collecting* a token, or *processing* a token. Scanning involves reading characters and throwing away those that do not start tokens, such as blanks and tabs. Collecting means gathering the characters that make up a token into a buffer. Processing may involve creating symbols, strings, lists, fixnums, bignums, or flonums; or calling a user written function called a character macro.

The components of the syntax class determine when the reader switches between the scanning, collecting, and processing states. The reader continues scanning as long as the character class of the characters it reads is *cseparator.* When it reads a character whose character class is not *cseparator*, it stores that character in its buffer and begins the collecting phase.

If the character class of that first character is *ccharacter*, *cnumber*, *cperiod*, or *csign*, then it continues collecting until it runs into a character whose syntax class has the *separator* property. (That last character is pushed back into the input buffer and is the first character read next time.) Now, the reader goes into the processing phase, checking to see if the token it reads is a number or symbol. It is important to note that after the first character is collected the component of the syntax class that tells the reader to stop collecting is the *separator* property, not the character class.

If the character class of the character that stopped the scanning is not *ccharacter*, *cnumber*, *cperiod*, or *csign*, then the reader processes that character immediately. The character classes *csingle-macro*, *csingle-splicing-macro*, and *csingle-infix-macro* acts like *ccharacter* if the following token is not a *separator.* The processing that is done for a given character class is described in detail in the next section.

7.4. Character Classes

ccharacter

raw readtable:A-Z a-z ^H !#$%&*,/:;<=>?@^_‘{}~
standard readtable:A-Z a-z ^H !$%&*/:;<=>?@^_{}~

A normal character.

cnumber

raw readtable:0-9
standard readtable:0-9

This type is a digit. The syntax for an integer (fixnum or bignum) is a string of *cnumber* characters optionally followed by a *cperiod.* If the digits are not followed by a *cperiod*, then they are interpreted in base *ibase*, which must be eight or ten. The syntax for a floating point number is either zero or more *cnumber*'s followed by a *cperiod* and then followed by one or more *cnumber*'s. A floating point number may also be an integer or floating point number followed by 'e' or 'd', an optional '+' or '−' , and then zero or more *cnumber*'s.

csign — raw readtable:+ − ; standard readtable:+ −

A leading sign for a number. No other characters should be given this class.

cleft-paren — raw readtable:(; standard readtable:(

A left parenthesis. Tells the reader to begin forming a list.

cright-paren — raw readtable:) ; standard readtable:)

A right parenthesis. Tells the reader that it has reached the end of a list.

cleft-bracket — raw readtable:[; standard readtable:[

A left bracket. Tells the reader that it should begin forming a list. See the description of *cright-bracket* for the difference between cleft-bracket and cleft-paren.

cright-bracket — raw readtable:] ; standard readtable:]

A right bracket. A *cright-bracket* finishes the formation of the current list and all enclosing lists until it finds one that begins with a *cleft-bracket* or until it reaches the top level list.

cperiod — raw readtable:. ; standard readtable:.

The period is used to separate element of a cons cell; that is, (a . (b . nil)) is the same as (a b). *cperiod* is also used in numbers as described earlier.

cseparator — raw readtable:^I-^M esc space ; standard readtable:^I-^M esc space

Separates tokens. When the reader is scanning, these character are passed over. Note: there is a difference between the *cseparator* character class and the *separator* property of a syntax class.

csingle-quote — raw readtable:' ; standard readtable:'

This causes *read* to be called recursively and the list (quote <value read>) to be returned.

csymbol-delimiter — raw readtable:|

standard readtable:|

This causes the reader to begin collecting characters and to stop only when another identical *csymbol-delimiter* is seen. The only way to escape a *csymbol-delimiter* within a symbol name is with a *cescape* character. The collected characters are converted into a string which becomes the print name of a symbol. If a symbol with an identical print name already exists, then the allocation is not done, rather the existing symbol is used.

cescape raw readtable:\
standard readtable:\

This causes the next character that is read in to be treated as a **vcharacter**. A character whose syntax class is **vcharacter** has a character class *ccharacter* and does not have the *separator* property so it does not separate symbols.

cstring-delimiter raw readtable:"
standard readtable:"

This is the same as *csymbol-delimiter* except that the result is returned as a string instead of a symbol.

csingle-character-symbol raw readtable:none
standard readtable:none

This returns a symbol whose print name is the the single character that has been collected.

cmacro raw readtable:none
standard readtable:‘,

The reader calls the macro function associated with this character and the current readtable, passing it no arguments. The result of the macro is added to the structure the reader is building, just as if that form were directly read by the reader. More details on macros are provided later.

csplicing-macro raw readtable:none
standard readtable:#;

A *csplicing-macro* differs from a *cmacro* in the way the result is incorporated in the structure the reader is building. A *csplicing-macro* must return a list of forms (possibly empty). The reader acts as if it read each element of the list itself without the surrounding parenthesis.

csingle-macro raw readtable:none
standard readtable:none

This causes the reader to check the next character. If it is a *cseparator*, then this acts like a *cmacro*. Otherwise, it acts like a *ccharacter*.

csingle-splicing-macro raw readtable:none
standard readtable:none

This is triggered like a *csingle-macro*. However, the result is spliced in like a *csplicing-macro*.

cinfix-macro
raw readtable:none
standard readtable:none

This differs from a *cmacro* in that the macro function is passed a form representing what the reader has read so far. The result of the macro replaces what the reader had read so far.

csingle-infix-macro
raw readtable:none
standard readtable:none

This differs from the *cinfix-macro* in that the macro is only triggered if the character following the *csingle-infix-macro* character is a *cseparator.*

cillegal
raw readtable:^@-^G^N-^Z^\-^_rubout
standard readtable:^@-^G^N-^Z^\-^_rubout

The characters cause the reader to signal an error if read.

7.5. Syntax Classes

The readtable maps each character into a syntax class. The syntax class contains three pieces of information: the character class, whether this is a separator, and the escape properties. The first two properties are used by the reader, the last by the printer (and *explode*). The initial Lisp system has the following syntax classes defined. You may add syntax classes with *add-syntax-class.* For each syntax class, the properties of the class and which characters have this syntax class by default are listed. More information about each syntax class can be found under the description of the syntax class's character class.

vcharacter
ccharacter
raw readtable:A-Z a-z ^H !#$%&*,/:;<=>?@^_‘{}~
standard readtable:A-Z a-z ^H !$%&*/:;<=>?@^_{}~

vnumber
cnumber
raw readtable:0-9
standard readtable:0-9

vsign
csign
raw readtable:+-
standard readtable:+-

vleft-paren
cleft-paren
escape-always
separator
raw readtable:(
standard readtable:(

vright-paren
cright-paren
escape-always
separator
raw readtable:)
standard readtable:)

vleft-bracket
cleft-bracket
escape-always
separator
raw readtable:[
standard readtable:[

vright-bracket	raw readtable:]
cright-bracket	standard readtable:]
escape-always	
separator	
vperiod	raw readtable:.
cperiod	standard readtable:.
escape-when-unique	
vseparator	raw readtable:^I-^M esc space
cseparator	standard readtable:^I-^M esc space
escape-always	
separator	
vsingle-quote	raw readtable:'
csingle-quote	standard readtable:'
escape-always	
separator	
vsymbol-delimiter	raw readtable:|
csingle-delimiter	standard readtable:|
escape-always	
vescape	raw readtable:\
cescape	standard readtable:\
escape-always	
vstring-delimiter	raw readtable:"
cstring-delimiter	standard readtable:"
escape-always	
vsingle-character-symbol	raw readtable:none
csingle-character-symbol	standard readtable:none
separator	
vmacro	raw readtable:none
cmacro	standard readtable:`,
escape-always	
separator	
vsplicing-macro	raw readtable:none
csplicing-macro	standard readtable:#;
escape-always	
separator	
vsingle-macro	raw readtable:none
csingle-macro	standard readtable:none

escape-when-unique	
vsingle-splicing-macro *csingle-splicing-macro* *escape-when-unique*	raw readtable:none standard readtable:none
vinfix-macro *cinfix-macro* *escape-always* *separator*	raw readtable:none standard readtable:none
vsingle-infix-macro *csingle-infix-macro* *escape-when-unique*	raw readtable:none standard readtable:none
villegal *cillegal* *escape-always* *separator*	raw readtable:^@-^G^N-^Z^\-^_rubout standard readtable:^@-^G^N-^Z^\-^_rubout

7.6. Character Macros

Character macros are user-written functions that are executed during the reading process. The value returned by a character macro may or may not be used by the reader, depending on the type of macro and the value returned. Character macros are always attached to a single character with the *setsyntax* function.

7.6.1. Types

There are three types of character macros: normal, splicing, and infix. These types differ in the arguments they are given or in what is done with the result they return.

7.6.1.1. Normal

A normal macro is passed no arguments. The value returned by a normal macro is simply used by the reader as if it had read the value itself. Here is an example of a macro that returns the abbreviation for a given state.

```
-> (defun stateabbrev nil
        (cdr (assq (read) '((california . ca) (pennsylvania . pa)))))
stateabbrev
-> (setsyntax \! 'vmacro 'stateabbrev)
t
-> '( ! california ! wyoming ! pennsylvania)
(ca nil pa)
```

Notice what happened to *!wyoming*. Since it was not in the table, the associated function returned nil. The creator of the macro may have wanted to leave the list alone, in such a case, but could not with this type of reader macro. The splicing macro, described next, allows a character macro function to return a value that is ignored.

7.6.1.2. Splicing

The value returned from a splicing macro must be a list or nil. If the value is nil, then the value is ignored; otherwise, the reader acts as if it read each object in the list. Usually, the list only contains one element. If the reader is reading at the top level (that is, not collecting elements of list), then it is illegal for a splicing macro to return more than one element in the list. The major advantage of a splicing macro over a normal macro is the ability of the splicing macro to return nothing. The comment character (usually ;) is a splicing macro bound to a function which reads to the end of the line and always returns nil. Here is the previous example written as a splicing macro

```
-> (defun stateabbrev nil
        ((lambda (value)
              (cond (value (list value))
                    (t nil)))
         (cdr (assq (read) '((california . ca) (pennsylvania . pa))))))
-> (setsyntax '! 'vsplicing-macro 'stateabbrev)
-> '(!pennsylvania ! foo !california)
(pa ca)
-> '!foo !bar !pennsylvania
pa
->
```

7.6.1.3. Infix

Infix macros are passed a *conc* structure representing what has been read so far. Briefly, a tconc structure is a single list cell whose car points to a list and whose cdr points to the last list cell in that list. The interpretation by the reader of the value returned by an infix macro depends on whether the macro is called while the reader is constructing a list or whether it is called at the top level of the reader. If the macro is called while a list is being constructed, then the value returned should be a tconc structure. The car of that structure replaces the list of elements that the reader has been collecting. If the macro is called at top level, then it is passed the value nil, and the value it returns should either be nil or a tconc structure. If the macro returns nil, then the value is ignored and the reader continues to read. If the macro returns a tconc structure of one element, that is, whose car is a list of one element, then that single element is returned as the value of *read*. If the macro returns a tconc structure of more than one element, then that list of elements is returned as the value of read.

```
-> (defun plusop (x)
     (cond ((null x) (tconc nil \ +))
           (t (lconc nil (list 'plus (caar x) (read))))))

plusop
-> (setsyntax \ + 'vinfix-macro 'plusop)
t
-> '(a + b)
(plus a b)
-> '+
|+|
->
```

7.6.2. Invocations

There are three different circumstances in which you would like a macro function to be triggered.

Always -

Whenever the macro character is seen, the macro should be invoked. This is accomplished by using the character classes *cmacro*, *csplicing-macro*, or *cinfix-macro* and by using the *separator* property. The syntax classes **vmacro**, **vsplicing-macro**, and **vsingle-macro** are defined this way.

When first -

The macro should only be triggered when the macro character is the first character found after the scanning process. A syntax class for a *when first* macro is defined using *cmacro*, *csplicing-macro*, or *cinfix-macro* but not including the *separator* property.

When unique -

The macro should only be triggered when the macro character is the only character collected in the token collection phase of the reader; that is, the macro character is preceded by zero or more *cseparators* and followed by a *separator*. A syntax class for a *when unique* macro is defined using *csingle-macro*, *csingle-splicing-macro*, or *csingle-infix-macro* but not including the *separator* property. The syntax classes so defined are **vsingle-macro**, **vsingle-splicing-macro**, and **vsingle-infix-macro**.

7.7. Functions

(setsyntax 's_symbol 's_synclass ['ls_func]**)**

WHERE: ls_func is the name of a function or a lambda body.

RETURNS: t

SIDE EFFECT: S_symbol should be a symbol whose print name is only one character. The syntax class for that character is set to s_synclass in the current readtable. If s_synclass is a class that requires a character macro, then ls_func must be supplied.

NOTE: The symbolic syntax codes are new to this version of FRANZ LISP. For compatibility, s_synclass can be one of the fixnum syntax codes that appeared in older versions of the FRANZ LISP Manual. This compatibility is only temporary: existing code which uses the fixnum syntax codes should be converted.

(getsyntax 's_symbol**)**

RETURNS: The syntax class of the first character of s_symbol's print name. s_symbol's print name must be exactly one character long.

NOTE: This function is new to this version of FRANZ LISP. It supersedes *(status syntax)* that no longer exists.

(add-syntax-class 's_synclass 'l_properties**)**

RETURNS: s_synclass

SIDE EFFECT: Defines the syntax class s_synclass to have properties l_properties. The list l_properties should contain a character class mentioned earlier. l_properties may contain one of the escape properties: *escape-always*, *escape-when-unique*, or *escape-when-first*. l_properties may contain the *separator* property. After a syntax class has been defined with *add-syntax-class*, the *setsyntax* function can be used to give characters that syntax class.

```
; Define a non-separating macro character.
; This type of macro character is used in UCI-Lisp, and
; it corresponds to a  FIRST MACRO in Interlisp.

-> (add-syntax-class 'vuci-macro '(cmacro escape-when-first))
vuci-macro
->
```

CHAPTER 8

Functions, Fclosures, and Macros

8.1. valid function objects

There are many different objects that can occupy the function field of a symbol object. Table 8.1 shows all of the possibilities, how to recognize them, and where to look for documentation.

8.2. functions

The basic Lisp function is the lambda function. When a lambda function is called, the actual arguments are evaluated from left to right and are lambda-bound to the formal parameters of the lambda function.

An nlambda function is usually used for functions that are invoked at top level. Some built-in functions which evaluate their arguments in special ways are also nlambdas (for example *cond*, *do*, and *or*). When an nlambda function is called, the list of unevaluated arguments is lambda bound to the single formal parameter of the nlambda function.

In this case, some programmers use an nlambda function when they are not sure how many arguments will be passed. Then, the first thing the nlambda function does is map *eval* over the list of unevaluated arguments it has been passed. This is usually the wrong thing to do because it does not work compiled if any of the arguments are local variables. The solution is to use a lexpr. When a lexpr function is called, the arguments are evaluated and a fixnum, whose value is the number of arguments, is lambda-bound to the single formal parameter of the lexpr function. The lexpr can then access the arguments using the *arg* function.

When a function is compiled, *special* declarations may be needed to preserve its behavior. An argument is not lambda-bound to the name of the corresponding formal parameter unless that formal parameter has been declared *special* (see §12.3.2.2).

Lambda and lexpr functions both compile into a binary object with a discipline of lambda. However, a compiled lexpr still acts like an interpreted lexpr.

8.3. macros

An important feature of Lisp is its ability to manipulate programs as data. As a result of this, most Lisp implementations have very powerful macro facilities. The Lisp language's macro facility can be used to incorporate popular features of the other languages into Lisp. For example, there are macro packages that allow you to create

informal name	object type	documentation
interpreted lambda function	list with *car* *eq* to lambda	8.2
interpreted nlambda function	list with *car* *eq* to nlambda	8.2
interpreted lexpr function	list with *car* *eq* to lexpr	8.2
interpreted macro	list with *car* *eq* to macro	8.3
fclosure	vector with *vprop* *eq* to fclosure	8.4
compiled lambda or lexpr function	binary with discipline *eq* to lambda	8.2
compiled nlambda function	binary with discipline *eq* to nlambda	8.2
compiled macro	binary with discipline *eq* to macro	8.3
foreign subroutine	binary with discipline of "subroutine"	8.5
foreign function	binary with discipline of "function"	8.5
foreign integer function	binary with discipline of "integer-function"	8.5
foreign real function	binary with discipline of "real-function"	8.5
foreign C function	binary with discipline of "c-function"	8.5
foreign double function	binary with discipline of "double-c-function"	8.5
foreign structure function	binary with discipline of "vector-c-function"	8.5
array	array object	9

Table 8.1

records (as in Pascal) and refer to elements of those records by the field names. The *struct* package imported from Maclisp does this. Another popular use for macros is to create more readable control structures which expand into *cond*, *or*, and *and*. One such example is the If macro. It allows you to write

```
(If (equal numb 0) then (print 'zero) (terpr)
 elseif (equal numb 1) then (print 'one) (terpr)
 else (print |I give up|))
```

which expands to

```
(cond
    ((equal numb 0) (print 'zero) (terpr))
    ((equal numb 1) (print 'one) (terpr))
    (t (print |I give up|)))
```

8.3.1. macro forms

A macro is a function that accepts a Lisp expression as input and returns another Lisp expression. The action the macro takes is called macro expansion. Here is a simple example:

```
-> (def first (macro (x) (cons 'car (cdr x))))
first
-> (first '(a b c))
a
-> (apply 'first '(first '(a b c)))
(car '(a b c))
```

The first input line defines a macro called *first*. Notice that the macro has one formal parameter, *x*. On the second input line, you ask the interpreter to evaluate *(first '(a b c))*. *Eval* sees that *first* has a function definition of type macro, so it evaluates *first*'s definition, passing to *first*, as an argument, the form *eval* itself was trying to evaluate: *(first '(a b c))*. The *first* macro discards the car of the argument with *cdr*, cons' a *car* at the beginning of the list and returns *(car '(a b c))*, which *eval* evaluates. The value *a* is returned as the value of *(first '(a b c))*. Thus, whenever *eval* tries to evaluate a list whose car has a macro definition, it ends up doing (at least) two operations: the first of which is a call to the macro to let it macro expand the form, and the second of which is the evaluation of the result of the macro. The result of the macro may be yet another call to a macro, so *eval* may have to do even more evaluations until it can finally determine the value of an expression. One way to see how a macro expands is to use *apply* as shown on the third input line earlier.

8.3.2. defmacro

The macro *defmacro* makes it easier to define macros because it allows you to name the arguments to the macro call. For example, suppose you find yourself often writing code like *(setq stack (cons newelt stack)*. You could define a macro named *push* to do this. One way to define it is:

```
-> (def push
        (macro (x) (list 'setq (caddr x) (list 'cons (cadr x) (caddr x)))))
push
```

then *(push newelt stack)* expands to the form mentioned earlier. The same macro written using defmacro would be:

```
-> (defmacro push (value stack)
     (list 'setq ,stack (list 'cons ,value ,stack)))
push
```

Defmacro allows you to name the arguments of the macro call and makes the macro definition look more like a function definition.

8.3.3. the backquote character macro

The default syntax for FRANZ LISP has four characters with associated character macros. One is semicolon for comments. Two others are the backquote and comma, which are used by the backquote character macro. The fourth is the sharp sign macro described in the next section.

The backquote macro is used to create lists where many of the elements are fixed (quoted). This makes it very useful for creating macro definitions. In the simplest case, a backquote acts just like a single quote:

```
-> `(a b c d e)
(a b c d e)
```

If a comma precedes an element of a backquoted list, then that element is evaluated and its value is put in the list.

```
-> (setq d '(x y z))
(x y z)
-> `(a b c ,d e)
(a b c (x y z) e)
```

If a comma, followed by an at sign, precedes an element in a backquoted list, then that element is evaluated and spliced into the list with *append*

```
-> `(a b c ,@d e)
(a b c x y z e)
```

Once a list begins with a backquote, the commas may appear anywhere in the list as this example shows:

```
-> `(a b (c d ,(cdr d)) (e f (g h ,@ (cddr d) ,@ d)))
(a b (c d (y z)) (e f (g h z x y z)))
```

It is also possible, and sometimes even useful, to use the backquote macro within itself. As a final demonstration of the backquote macro, define the first and push macros using all the power at your disposal: defmacro and the backquote macro.

```
-> (defmacro first (list) `(car ,list))
first
-> (defmacro push (value stack) `(setq ,stack (cons ,value ,stack)))
stack
```

8.3.4. sharp sign character macro

The sharp sign macro can perform a number of different functions at read time. The character directly following the sharp sign determines which function is done, and the following Lisp s-expressions may serve as arguments. A full list of sharp sign macro capabilities can be found in Chapter 14.

8.3.4.1. conditional inclusion

If you plan to run one source file in more than one environment, then you may want some pieces of code to be included or not included depending on the environment. The C language uses "#ifdef" and "#ifndef" for this purpose, and Lisp uses "#+" and "#−". The environment that the sharp sign macro checks is the *(status features)* list, which is initialized when the Lisp system is built and which may be altered by *(sstatus feature foo)* and *(sstatus nofeature bar)*. The form of conditional inclusion is

# +when what

where *when* is either a symbol or an expression involving symbols and the functions *and*, *or*, and *not*. The meaning is that *what* is only read in if *when* is true. A symbol in *when* is true only if it appears in the *(status features)* list.

```
; Suppose you want to write a program that references a file
; and that can run at ucb, ucsd, and cmu where the file naming conventions
; are different.
;
-> (defun howold (name)
      (terpr)
      (load # +(or ucb ucsd) "/usr/lib/lisp/ages.l"
            # +cmu "/usr/lisp/doc/ages.l")
      (patom name)
      (patom " is ")
      (print (cdr (assoc name agefile)))
      (patom "years old")
      (terpr))
```

The form

# −when what

is equivalent to

# +(not when) what

8.3.4.2. fixnum character equivalents

When you work with fixnum equivalents of characters, it is often hard to remember the number corresponding to a character. The form

#/c

is equivalent to the fixnum representation of character c.

```
; A function that returns t if the user types y else it returns nil.
;
-> (defun yesorno nil
      (progn (ans)
            (setq ans (tyi))
            (cond ((equal ans #/y) t)
                  (t nil))))
```

8.3.4.3. read time evaluation

Occasionally you want to express a constant as a Lisp expression, yet you do not want to pay the penalty of evaluating this expression each time it is referenced. The form

#.expression
evaluates the expression at read time and returns its value.

```
; Here is a function to test if any of bits 1, 3 or 12 are set in a fixnum.
;
-> (defun testit (num)
        (cond ((zerop (boole 1 num #.(+ (lsh 1 1) (lsh 1 3) (lsh 1 12))))
                nil)
              (t t)))
```

8.4. fclosures

Fclosures are a type of functional object. Their purpose is to remember the values of some variables between invocations of the functional object and to protect this data from being inadvertently overwritten by other Lisp functions. Fortran programs often exhibit this kind of memory, although some versions of Fortran (correctly) require such permanent storage to be in COMMON. Using this remembered data it is easy to write a linear congruent random number generator in Fortran merely by keeping the seed as a variable within the function. It is much more risky to do so in Lisp, since any special variable you pick might be used by some other function. Fclosures are an attempt to provide most of the same functionality as closures in Lisp Machine Lisp to users of FRANZ LISP. Fclosures are related to closures in this way:
(fclosure '(a b) 'foo) <==>
        (let ((a a) (b b)) (closure '(a b) 'foo))

8.4.1. an example

```
++ lisp
Franz Lisp, Opus 40.03
->(defun code (me count)
  (print (list 'in x))
  (setq x (+ 1 x))
  (cond ((greaterp count 1) (funcall me me (sub1 count))))
  (print (list 'out x)))
code
->(defun tester (object count)
  (funcall object object count) (terpri))
tester
->(setq x 0)
0
->(setq z (fclosure '(x) 'code))
fclosure[8]
-> (tester z 3)
(in 0)(in 1)(in 2)(out 3)(out 3)(out 3)
nil
->x
0
```

The function *fclosure* creates a new object that is called an fclosure, although it is actually a vector. The fclosure contains a functional object, and a set of symbols and values for the symbols. In the earlier example, the fclosure functional object is the function code. The set of symbols and values just contains the symbol 'x' and zero, the value of 'x' when the fclosure was created.

When an fclosure is funcall'ed:

1) The Lisp system lambda binds the symbols in the fclosure to their values in the fclosure.
2) It continues the funcall on the functional object of the fclosure.
3) Finally, it un-lambda binds the symbols in the fclosure and at the same time stores the current values of the symbols in the fclosure.

Notice that the fclosure is saving the value of the symbol 'x'. Each time a fclosure is created, new space is allocated for saving the values of the symbols. Thus, if you execute fclosure again, over the same function, you can have two independent counters:

```
-> (setq zz (fclosure '(x) 'code))
fclosure[1]
-> (tester zz 2)
(in 0)(in 1)(out 2)(out 2)
-> (tester zz 2)
(in 2)(in 3)(out 4)(out 4)
-> (tester z 3)
(in 3)(in 4)(in 5)(out 6)(out 6)(out 6)
```

8.4.2. useful functions

Here are some quick summaries of functions dealing with closures. They are formally defined in §2.8.4. To recap, fclosures are made by *(fclosure 'l_vars 'g_funcobj)*. l_vars is a list of symbols (not containing nil); g_funcobj is any object that can be funcalled. (Objects that can be funcalled include compiled Lisp functions, lambda expressions, symbols, foreign functions, etc.) In general, if you want a compiled function to be closed over a variable, you must declare the variable to be special within the function. Another example is:

```
(fclosure '(a b) #'(lambda (x) (plus x a)))
```

Here, the #' construction makes the compiler compile the lambda expression.

There are times when you want to share variables between fclosures. This can be done if the fclosures are created at the same time using *fclosure-list*.

(fclosure-list l_list u_function [...]**)**

RETURNS: A list of the fclosures of the functions over the just previous list of variables.

NOTE: Any number of list-function pairs may be given. An fclosure of each u_function is created with respect to the values in l_list, which should be a list of variables. All the variables specified in a l_list are closed in the subsequent function. If the same symbol appears in more than one l_list, all its occurrences are treated as references to the same variable. A list of the fclosures of each function is returned.

The function *fclosure-alist* returns an assoc list giving the symbols and values in the fclosure. The predicate *fclosurep* returns t if and only if its argument is an fclosure. Other functions imported from Lisp Machine Lisp are *symeval-in-fclosure, let-fclosed,* and *set-in-fclosure.* Finally, the function *fclosure-function* returns the function argument.

8.4.3. internal structure

Currently, closures are implemented as vectors with property being the symbol fclosure. The functional object is the first entry. The remaining entries are structures that point to the symbols and values for the closure, with a reference count to determine if a recursive closure is active. This particular implementation is subject to change in the interests of efficiency and generality.

8.5. Foreign subroutines and functions

FRANZ LISP has the ability to dynamically load object files produced by other compilers and to call functions defined in those files to the extent that the other language processors abide by the same operating system standards for function calls.

Most implementations of FRANZ LISP co-exist with a C compiler, a Fortran compiler, and a Pascal compiler. (These may be available only as optional languages).

This section deals with defining and using these so-called *foreign* functions.* There are seven types of foreign functions. They are characterized by the type of the result each returns and by differences in the interpretation of their arguments. They come from two families: a group suited for languages that pass arguments by reference (e.g., Fortran), and a group suited for languages which pass arguments by value (e.g., C).

There are four types in the first group:

subroutine
This does not return anything. The Lisp system always returns t after calling a subroutine.

function
This returns whatever the function returns. This must be a valid Lisp object or it may cause the Lisp system to fail.

integer-function
This returns an integer that the Lisp system makes into a fixnum and returns.

real-function
This returns a double precision real number that the Lisp system makes into a

*This topic is also discussed in Report PAM-124 of the Center for Pure and Applied Mathematics, UC Berkeley, entitled "Parlez-Vous Franz? An Informal Introduction to Interfacing Foreign Functions to Franz LISP", by James R.

flonum and returns.

There are three types in the second group:

c-function
This is like an integer function except for its different interpretation of arguments.

double-c-function
This is like a real-function.

vector-c-function
This is for C functions that return a structure. The first argument to such functions must be a vector, of type vectori, into which the result is stored. The second Lisp argument becomes the first argument to the C function, and so on.

A foreign function is accessed through a binary object just like a compiled Lisp function. The difference is that the discipline field of a binary object for a foreign function is a string whose first character is given in the following table:

letter	type
s	subroutine
f	function
i	integer-function
r	real-function.
c	c-function
v	vector-c-function
d	double-c-function

Two functions are provided for setting-up foreign functions. *Cfasl* loads an object file into the Lisp system and sets up one foreign function binary object. If there is more than one function in an object file, *getaddress* can be used to set up additional foreign function objects.

Foreign functions are called just like other functions, for example, *(funname arg1 arg2)*. When a function in the Fortran group is called, the arguments are evaluated and then examined. List, hunk, and symbol arguments are passed unchanged to the foreign function. Fixnum and flonum arguments are copied into a temporary location and a pointer to the value is passed. (This is because Fortran uses call by reference and it is dangerous to modify the contents of a fixnum or flonum which something else might point to.) If the argument is an array object, the data field of the array object is passed to the foreign function. (This is the easiest way to send large amounts of data to, and receive large amounts of data from, a foreign function.) If a binary object is an argument, the entry field of that object is passed to the foreign function. (The entry field is the address of a function, so this amounts to passing a function as an argument).

When a function in the C group is called, fixnum and flownum arguments are passed by value. For almost all other arguments, the address is merely provided to the C routine. The only exception arises when you want to invoke a C routine that expects a "structure" argument. Recall that a (rarely used) feature of the C language is the ability to pass structures by value. This copies the structure onto the stack. Since FRANZ LISP's nearest equivalent to a C structure is a vector, you are provided an escape clause to copy the contents of an immediate-type vector by value. If the property field of a vectori argument is the symbol "value-structure-argument", then the binary data of this immediate-type vector is copied into the argument list of the C routine.

The method a foreign function uses to access the arguments provided by Lisp is dependent on the language of the foreign function. The following scripts demonstrate

Larus

how Lisp can interact with three languages: C, Pascal, and Fortran. C and Pascal have pointer types and the first script shows how to use pointers to extract information from Lisp objects. There are two functions defined for each language. The first (cfoo in C, pfoo in Pascal) is given four arguments: a fixnum, a flonum-block array, a hunk of at least two fixnums, and a list of at least two fixnums. To demonstrate that the values were passed, each ?foo function prints its arguments (or parts of them). The ?foo function then modifies the second element of the flonum-block array and returns a 3 to Lisp. The second function (cmemq in C, pmemq in Pascal) acts just like the Lisp *memq* function except that it does not work for fixnums whereas the Lisp *memq* does work for small fixnums. In the script, typed input is in **bold**, computer output is in roman, and comments are in *italic.*

These are the C coded functions
++ list ch8auxc.c

```
/* demonstration of c coded foreign integer-function */

/* The following is used to extract fixnums out of a list of fixnums */
struct listoffixnumscell
{   struct listoffixnumscell *cdr;
    int *fixnum;
};

struct listcell
{           struct listcell *cdr;
            int car;
};

cfoo(a,b,c,d)
int *a;
double b[];
int *c[];
struct listoffixnumscell *d;
{
   printf("a: %d, b[0]: %f, b[1]: %f0, *a, b[0], b[1]);
   printf(" c (first): %d   c (second): %d0,
                *c[0],*c[1]);
   printf(" ( %d %d ... ) ", *(d->fixnum), *(d->cdr->fixnum));
   b[1] = 3.1415926;
   return(3);
}

struct listcell *
cmemq(element,list)
int element;
struct listcell *list;
{
  for( ; list && element != list->car ; list = list->cdr);
  return(list);
}
```

These are the Pascal coded functions
++ list ch8auxp.p

```
type      pinteger = ^integer;
          realarray = array[0..10] of real;
          pintarray = array[0..10] of pinteger;
          listoffixnumscell = record
                                    cdr  : ^listoffixnumscell;
                                    fixnum : pinteger;
                               end;
          plistcell = ^listcell;
```

```
     listcell = record
                   cdr : plistcell;
                   car : integer;
                 end;

function pfoo ( var a : integer ;
                var b : realarray;
                var c : pintarray;
                var d : listoffixnumscell) : integer;
begin
  writeln(' a:',a, ' b[0]:', b[0], ' b[1]:', b[1]);
  writeln(' c (first):', c[0]^,' c (second):', c[1]^);
  writeln(' ( ', d.fixnum^, d.cdr^.fixnum^, ' ...) ');
  b[1] := 3.1415926;
  pfoo := 3
end ;

{ The function, pmemq, looks for the Lisp pointer given as the first argument
  in the list pointed to by the second argument.
  Note that you should declare " a : integer " instead of " var a : integer " since
  you are interested in the pointer value instead of what it points to, which
  could be any Lisp object.
}
function pmemq( a : integer; list : plistcell) : plistcell;
begin
 while (list <> nil) and (list^.car <> a) do list := list^.cdr;
 pmemq := list;
end ;
```

The files are compiled

```
++ cc +r ch8auxc.c
++ pc +r ch8auxp.p
```

```
++ lisp
Franz Lisp, Opus 41.10
```

First the files are loaded, and one foreign function binary is set up. There are two functions in each file so you must choose one to tell cfasl about. The choice is arbitrary.

```
-> (cfasl 'ch8auxc.r '_cfoo 'cfoo "integer-function")
#63000-"integer-function"
-> (cfasl 'ch8auxp.r '_pfoo 'pfoo "integer-function" "+lpc")
#63200-"integer-function"
```

Here you set up the other foreign function binary objects

```
-> (getaddress '_cmemq 'cmemq "function" '_pmemq 'pmemq "function")
#6306c-"function"
```

Suppose you want to create and initialize an array to pass to the cfoo function. In this case, you create an unnamed array and store it in the value cell of testarr. When you create an array to pass to the Pascal program, you can use a named array just to demonstrate the different way that named and unnamed arrays are created and accessed.

```
-> (setq testarr (array nil flonum-block 2))
array[2]
-> (store (funcall testarr 0) 1.234)
1.234
-> (store (funcall testarr 1) 5.678)
5.678
-> (cfoo 385 testarr (hunk 10 11 13 14) '(15 16 17))
a: 385, b[0]: 1.234000, b[1]: 5.678000
 c (first): 10   c (second): 11
 ( 15 16 ... )
 3
```

Note that cfoo has returned 3 as it should. It also had the side effect of changing the second value of the array to 3.1415926 which check next.

```
-> (funcall testarr 1)
3.1415926
```

In preparation for calling pfoo, you create an array.

```
-> (array test flonum-block 2)
array[2]
-> (store (test 0) 1.234)
1.234
-> (store (test 1) 5.678)
5.678
-> (pfoo 385 (getd 'test) (hunk 10 11 13 14) '(15 16 17))
 a:      385 b[0]: 1.23400000000000E+00 b[1]: 5.67800000000000E+00
 c (first):       10 c (second):       11
 (       15       16 ...)
3
-> (test 1)
3.1415926

 Now to test out the memq's
-> (cmemq 'a '(b c a d e f))
(a d e f)
-> (pmemq 'e '(a d f g a x))
nil
```

The Fortran example is much shorter since in Fortran you cannot follow pointers as you can in other languages. The Fortran function, ffoo, is given three arguments: a fixnum, a fixnum-block array, and a flonum. These arguments are printed out to verify that they made it, and, then, the first value of the array is modified. The function returns a double precision value, which is converted to a flonum by Lisp and printed.

```
++ list ch8auxf.f
        double precision function ffoo(a,b,c)
        integer a,b(10)
        double precision c
        print 2,a,b(1),b(2),c
2       format(' a=',i4,', b(1)=',i5,', b(2)=',i5,' c=',f6.4)
        b(1) = 22
        ffoo = 1.23456
        return
        end
++ fortran +r ch8auxf.f
ch8auxf.f:
   ffoo:
++ lisp
Franz Lisp, Opus 40.03
-> (cfasl 'ch8auxf.o '_ffoo_ 'ffoo "real-function" "-lI77 -lF77")
#6307c-"real-function"

-> (array test fixnum-block 2)
array[2]
-> (store (test 0) 10)
10
-> (store (test 1) 11)
11
-> (ffoo 385 (getd 'test) 5.678)
 a= 385, b(1)=    10, b(2)=    11 c=5.6780
1.234559893608093
-> (test 0)
22
```

CHAPTER 9

Arrays and Vectors

Arrays and vectors are two means of expressing aggregate data objects in FRANZ LISP. Vectors may be thought of as sequences of data. They are intended as a vehicle for user-defined data types. This use of vectors is still experimental and subject to revision. As a simple data structure, they are similar to hunks and strings. Vectors are used to implement closures and are useful to communicate with foreign functions. Both of these topics were discussed in Chapter 8. Later in this chapter, the current implementation of vectors is described and you are advised what is most likely to change.

Arrays in FRANZ LISP provide a programmable data structure access mechanism. One possible use for FRANZ LISP arrays is to implement Maclisp style arrays, which are simple vectors of fixnums, flonums, or general Lisp values. This is described in more detail in §9.3, but first how array references are handled by the Lisp system is described.

The structure of an array object is given in §1.3.10 and reproduced here. lisp values.

Subpart name	Get value	Set value	Type
access function	getaccess	putaccess	binary, list or symbol
auxiliary	getaux	putaux	lispval
data	arrayref	replace set	block of contiguous lispval
length	getlength	putlength	fixnum
delta	getdelta	putdelta	fixnum

9.1. general arrays Suppose the evaluator is told to evaluate *(foo a b)* and the function cell of the symbol foo contains an array object, which is called foo_arr_obj. First. the evaluator evaluates and stacks the values of *a* and *b*. Next, it stacks the array object foo_arr_obj. Finally, it calls the access function of foo_arr_obj. The access function should be a lexpr[†] or a symbol whose function cell contains a lexpr. The access function is responsible for locating and returning a value from the array. The array access function is free to interpret the arguments as it wishes. The Maclisp compatible array access function, which is provided in the standard FRANZ LISP system, interprets the arguments as subscripts in the same way as languages like Fortran and Pascal.

The array access function also is called upon to store elements in the array. For example, *(store (foo a b) c)* automatically expands to (foo c a b), and, when the evaluator is called to evaluate this, it evaluates the arguments *c*, *b*, and *a*. Then it stacks the array object, which is stored in the function cell of foo, and calls the array access function with (now) four arguments. The array access function must be able to tell this is a store operation, which it can do by checking the number of arguments it has been given. (A lexpr can do this very easily.)

[†]A lexpr is a function that accepts any number of arguments, which are evaluated before the function is called.

9.2. subparts of an array object An array is created by allocating an array object with *marray* and filling in the fields. Certain Lisp functions interpret the values of the subparts of the array object in special ways as described in the following text. Placing illegal values in these subparts may cause the Lisp system to fail.

9.2.1. access function The purpose of the access function has been described earlier. The contents of the access function should be a lexpr: either a binary (compiled function) or a list (interpreted function). It may also be a symbol whose function cell contains a function definition. This subpart is used by *eval*, *funcall*, and *apply* when evaluating array references.

9.2.2. auxiliary This can be used for any purpose. If it is a list and the first element of that list is the symbol unmarked_array, then the data subpart is not marked by the garbage collector. Note that this is used in the Maclisp compatible array package and has the potential for causing strange errors if used incorrectly.

9.2.3. data This is either nil or points to a block of data space allocated by *segment* or *small-segment*.

9.2.4. length This is a fixnum whose value is the number of elements in the data block. This is used by the garbage collector and by *arrayref* to determine if your index is in bounds.

9.2.5. delta This is a fixnum whose value is the number of bytes in each element of the data block. This is four for an array of fixnums or value cells and eight for an array of flonums. This is used by the garbage collector and *arrayref* as well.

9.3. The Maclisp compatible array package

A Maclisp style array is similar to what is known as an array structure in other languages: a block of homogeneous data elements that is indexed by one or more integers called subscripts. The data elements can be all fixnums, flonums, or general Lisp objects. An array is created by a call to the function *array* or **array*. The only difference is that **array* evaluates its arguments. This call: *(array foo t 3 5)* sets up an array called foo of dimensions 3 by 5. The subscripts are zero based. The first element is *(foo 0 0)*, the next is *(foo 0 1)* and so on up to *(foo 2 4)*. The t indicates a general Lisp object array, which means each element of foo can be any type. Each element can be any type since all that is stored in the array is a pointer to a Lisp object, not the object itself. *Array* does this by allocating an array object with *marray* and then allocating a segment of 15 consecutive value cells with *small-segment* and storing a pointer to that segment in the data subpart of the array object. The length and delta subpart of the array object are filled in (with 15 and 4 respectively) and the access function subpart is set to point to the appropriate array access function. In this case, there is a special access function for two dimensional value cell arrays called arrac-twoD, and this access function is used. The auxiliary subpart is set to (t 3 5) which describes the type of array and the bounds of the subscripts. Finally, this array object is placed in the function cell of the symbol foo. Now when *(foo 1 3)* is evaluated, the array access function is invoked with three arguments: 1, 3, and the array object. From the auxiliary field of the array object it gets a description of the particular array. It then determines which element *(foo 1 3)* refers to and uses arrayref to extract that element.

Since this is an array of value cells, what arrayref returns is a value cell whose value is what is wanted, so the value cell is evaluated and it is returned as the value of *(foo 1 3)*.

In Maclisp, the call *(array foo fixnum 25)* returns an array whose data object is a block of 25 memory words. When fixnums are stored in this array, the actual numbers are stored instead of pointers to the numbers as is done in general Lisp object arrays. This is efficient under Maclisp but inefficient in FRANZ LISP since every time a value was referenced from an array it had to be copied and a pointer to the copy returned to prevent aliasing[†]. Thus t, fixnum, and flonum arrays are all implemented in the same manner. This should not affect the compatibility of Maclisp and FRANZ LISP. If there is an application where a block of fixnums or flonums is required, then exactly the same effect of fixnum and flonum arrays in Maclisp can be achieved by using fixnum-block and flonum-block arrays. Such arrays are required if you want to pass a large number of arguments to a Fortran or C coded function and then get answers back.

The Maclisp compatible array package is just one example of how a general array scheme can be implemented. Another type of array you can implement is the hashed array. The subscript can be anything, not just a number. The access function hashes the subscript and uses the result to select an array element. With the generality of arrays also comes extra cost; if you just want a simple aggregate of less than 128 general Lisp objects, you would be wise to look into using hunks.

9.4. vectors Vectors were invented to fix two shortcomings of hunks. They can be longer than 128 elements. They also have a tag associated with them, which is intended to say, for example, "Think of me as a *Blobit*." Thus, a **vector** is an arbitrarily sized hunk with a property list.

Continuing the example, the Lisp kernel may not know how to print out or evaluate *blobits*, but this is information that is common to all *blobits*. On the other hand, for each individual blobit, there are particulars that are likely to change: height, weight, or eye-color. This is the part that would previously have been stored in the individual entries in the hunk and are stored in the data slots of the vector. Here is a summary of the structure of a vector in tabular form:

[†]Aliasing happens when two variables share the same storage location. For example, if the copying mentioned were not done, then, after *(setq x (foo 2))* was done, the value of x and (foo 2) would share the same location. Then should the value of (foo 2) change, x's value would change as well. This is considered dangerous and, as a result, pointers are never returned into the data space of arrays.

Subpart name	Get value	Set value	Type
datum[i]	vref	vset	lispval
property	vprop	vsetprop vputprop	lispval
size	vsize	—	fixnum

Vectors are created specifying size and optional fill value using the function (*new-vector* 'x_size ['g_fill ['g_prop]]) or by initial values: (*vector* ['g_val ...]).

9.5. anatomy of vectors

There are some technical details about vectors that you should know:

9.5.1. size

You are not free to alter this. It is noted when the vector is created and is used by the garbage collector. The garbage collector coalesces two free vectors, which are neighbors in the heap. Internally, this is kept as the number of bytes of data. Thus, a vector created by (*vector* 'foo) has a size of 4.

9.5.2. property

Currently, you expect the property to be either a symbol or a list whose first entry is a symbol. The symbols **fclosure** and **structure-value-argument** are reserved for special system uses
and their effect is described in Chapter 8. If the property is a non-null symbol, the vector is printed out as <symbol>[<size>]. Another case is if the property is actually a (disembodied) property-list, which contains a value for the indicator **print**. The value is taken to be a Lisp function, which the printer invokes with two arguments: the vector and the current output port. Otherwise, the vector is printed as vector[<size>].

9.5.3. internal order

In memory, vectors start with a longword containing the size, which is immediate data within the vector. The next cell contains a pointer to the property. Any remaining cells, if any, are for data. Vectors are handled differently from any other object in FRANZ LISP in that a pointer to a vector is a pointer to the first data cell, that is, a pointer to the *third* longword of the structure. This was done for efficiency in compiled code and for uniformity in referencing immediate-vectors (described later). You should never return a pointer to any other part of a vector because this may cause the garbage collector to follow an invalid pointer.

9.6. immediate-vectors

Immediate-vectors are similar to vectors. However, they differ in that binary data are stored in space directly within the vector. Thus, the garbage collector preserves the vector itself, if used, and only traverses the property cell. The data may be referenced as longwords, shortwords, or even bytes. Shorts and bytes are returned sign-extended. The compiler open-codes such references, and avoids boxing the resulting integer data, where possible. Thus, immediate vectors may be used for efficiently processing character data. They are also useful in storing results from functions written in other languages.

Subpart name	Get value	Set value	Type
datum[i]	vrefi-byte vrefi-word vrefi-long	vseti-byte vseti-word vseti-long	fixnum fixnum fixnum
property	vprop	vsetprop vputprop	lispval
size	vsize vsize-byte vsize-word	—	fixnum fixnum fixnum

To create immediate vectors specifying size and fill data, you can use the functions *new-vectori-byte*, *new-vectori-word*, or *new-vectori-long*. You can also use the functions *vectori-byte*, *vectori-word*, or *vectori-long*. All of these functions are described in Chapter 2.

CHAPTER 10

Exception Handling

10.1. Errset and Error Handler Functions

FRANZ LISP allows you to handle in a number of ways the errors that arise during computation. One way is through the use of the *errset* function. If an error occurs during the evaluation of the *errset's* first argument, then the locus of control returns to the errset which returns nil (except in special cases, such as *err*). The other method of error handling is through an error handler function. When an error occurs, the error handler is called and is given as an argument a description of the error that just occurred. The error handler may take one of the following actions:

(1) It could take some drastic action like a *reset* or a *throw*.

(2) It could, if that the error is continuable, return to the function that noticed the error. The error handler indicates that it wants to return a value from the error by returning a list whose *car* is the value it wants to return.

(3) It could decide not to handle the error and return a non-list to indicate this fact.

10.2. The Anatomy of an error

Each error is described by a list of these items:

(1) Error type - This is a symbol that indicates the general classification of the error. This classification may determine which function handles this error.

(2) fixnum id - a fixnum identifying the error. In the future each error will have a unique number.

(3) Continuable - If this is non-nil, then this error is continuable.

(4) Message string - This is a symbol whose print name is a message describing the error.

(5) Data - There may be from zero to three Lisp values that help describe this particular error. For example, the unbound variable error contains one datum value, the symbol whose value is unbound. The list describing that error might look like:

```
(ER%misc 0 t |Unbound Variable:| foobar)
```

10.3. Error handling algorithm

This is the sequence of operations when an error occurs:

(1) If the symbol **ER%all** has a non-nil value, then this value is the name of an error handler function. That function is called with a description of the error. If that function returns (and, of course, it may choose not to) and the value is a list and this error is continuable, then the *car* of the list to the function which called the error is returned. Presumably, the function uses this value to retry the operation.

On the other hand, if the error handler returns a non-list, then it has chosen not to handle this error, which leads to step (2). Something special happens before the **ER%all** error handler is called, which does not happen in any of the other cases described later. To help insure that infinitely recursive errors do not occur, if **ER%all** is set to a bad value, the value of **ER%all** is set to nil before the handler is called. Thus, it is the responsibility of the **ER%all** handler to 'reenable' itself by storing its name in **ER%all.**

(2) Next, the specific error handler for the type of error that just occurred is called, if one exists, to see if it wants to handle the error. The names of the handlers for the specific types of errors are stored as the values of the symbols whose names are the types. For example, the handler for miscellaneous errors is stored as the value of **ER%misc.** Of course, if **ER%misc** has a value of nil, then there is no error handler for this type of error. Appendix B contains a list of all error types. The process of classifying the errors is not complete, and, thus, most errors are lumped into the **ER%misc** category. Just as in step (1), the error handler function may choose not to handle the error by returning a non-list, which leads to step (3).

(3) Next, a check is made to see if there is an *errset* surrounding this error. If so the second argument to the *errset* call is examined. If the second argument was not given or is non-nil then the error message associated with this error is printed. Finally, the stack is popped to the context of the *errset* and then the *errset* returns nil. If there was no *errset* step (4) is executed.

(4) If the symbol **ER%tpl** has a value, then it is the name of an error handler that is called in a manner similar to that discussed earlier. If it chooses not to handle the error, step (5) is executed.

(5) At this point, it has been determined that you do not want to handle this error. Thus, the error message is printed out and a *reset* is done to send the flow of control to the top-level.

To summarize the error handling system: When an error occurs, you have two chances to handle it before the search for an *errset* is done. Then, if there is no *errset*, you have one more chance to handle the error before control jumps to the top level. Every error handler works in the same way: It is given a description of the error (as described in the previous section). It may or may not return. If it returns, then it returns either a list or a non-list. If it returns a list and the error is continuable, then the *car* of the list is returned to the function that noticed the error. Otherwise, the error handler has decided not to handle the error.

10.4. Default aids

There are two standard error handlers that probably handle the needs of most users. One of these is the Lisp-coded function *break-err-handler*, which is the default value of **ER%tpl.** Thus, when all other handlers have ignored an error, *break-err-handler* takes over. It prints out the error message and goes into a read-eval-print loop. The other standard error handler is *debug-err-handler*. This handler is designed to be connected to **ER%all** and is useful if your program uses *errset* and you want to look at the error before it is thrown up to the *errset*.

10.5. Autoloading

When *eval*, *apply*, or *funcall* are told to call an undefined function, an **ER%undef** error is signaled. The default handler for this error is *undef-func-handler*. This function checks the property list of the undefined function for the indicator, autoload. If it is present, the value of that indicator should be the name of the file that contains the definition of the undefined function. *Undef-func-handler* loads the file and check if it has defined the function which caused the error. If it has, the error handler returns and the computation continues as if the error did not occur. This provides a way for you to tell the Lisp system about the location of commonly used functions. The trace package sets up an autoload property to point to /lisp/lib/trace.

10.6. Interrupt processing

The operating system provides one user-interrupt character that defaults to ^C.[†] You may select a Lisp function to run when an interrupt occurs. Since this interrupt could occur at any time and, in particular, could occur at a time when the internal stack pointers are in an inconsistent state, the processing of the interrupt may be delayed until a safe time. When the first ^C is typed, the Lisp system sets a flag that an interrupt has been requested. This flag is checked at safe places within the interpreter and in the *qlinker* function. If the Lisp system does not respond to the first ^C, another ^C should be typed. This causes all of the transfer tables to be cleared, forcing all calls from compiled code to go through the *qlinker* function where the interrupt flag is checked. If the Lisp system still doesn't respond, a third ^C causes an immediate interrupt. This interrupt is not necessarily in a safe place, so the user should *reset* the Lisp system as soon as possible.

[†]Actually there are two but the lisp system does not allow you to catch the QUIT interrupt.

CHAPTER 11

The Lister Trace Package

The Lister Trace package is an important tool for the interactive debugging of a Lisp program. It allows you to examine selected calls to a function or functions, and optionally to stop execution of the Lisp program to examine the values of variables.

The trace package is a set of Lisp programs located in the Lisp program library (usually in the file /lisp/lib/trace.l). Although not normally loaded in the Lisp system, the package is loaded when the first call to *trace* is made.

(trace [ls_arg1 ...])

WHERE: The form of the ls_arg*i* is described later.

RETURNS: A list of the function sucessfully modified for tracing. If no arguments are given to *trace*, a list of all functions currently being traced is returned.

SIDE EFFECT: The definitions of the functions indicated in the argument list are (usually temporarily) modified.

The ls_arg*i* can have one of the following forms:

foo - When foo is entered and exited, the trace information is printed.

(foo break) - When foo is entered and exited, the trace information is printed. Also, just after the trace information for foo is printed upon entry, you are put in a special break loop. The prompt is 'T{1}' and you may type any Lisp expression and see its value printed. The *i*th argument to the function just called can be accessed as (arg *i*).

To leave the trace loop, just type ?ret and execution continues.

(foo if expression) - When foo is entered and the expression evaluates to non-nil, then the trace information is printed for both exit and entry. If expression evaluates to nil, then no trace information is printed.

(foo ifnot expression) - When foo is entered and the expression evaluates to nil, then the trace information is printed for both entry and exit. If both **if** and **ifnot** are specified, then the **if** expression must evaluate to non nil AND the **ifnot** expression must evaluate to nil for the trace information to be printed out.

(foo evalin expression) - When foo is entered and after the entry trace information is printed, expression is evaluated. Exit trace information is printed when foo exits.

(foo evalout expression) - When foo is entered, entry trace information is printed. When foo exits, and before the exit trace information is printed, expression is evaluated.

(foo evalinout expression) - This has the same effect as (trace (foo evalin expression evalout expression)).

(foo lprint) - This tells *trace* to use the level printer when printing the arguments to and the result of a call to foo. The level printer prints only the top levels of list structure. Any structure below three levels is printed as an &. This allows you to trace functions with massive arguments or results.

Ordinarily the output from the trace package is printed with prinlevel bound to trace-prinlevel (default 4) and prinlength bound to trace-prinlength (default 5). Prinlevel and prinlength, which are useful in cutting off verbose or infinite (cyclical) structures, are described in Appendix B. If you wish to always print full lists then setting trace-prinlevel and trace-prinlength each to nil, will accomplish this.

The following trace options permit you to have greater control over each action that takes place when a function is traced. These options are only meant to be used by programmers who need special hooks into the trace package. Most programmers should skip reading this section.

(foo traceenter tefunc) - This tells *trace* that the function to be called when foo is entered is tefunc. tefunc should be a lambda of two arguments. The first argument is bound to the name of the function being traced, foo in this case. The second argument is bound to the list of arguments to which foo should be applied. The function tefunc should print some sort of "entering foo" message. It should not apply foo to the arguments, however. That is done later on.

(foo traceexit txfunc) - This tells *trace* that the function to be called when foo is exited is txfunc. txfunc should be a lambda of two arguments. The first argument is bound to the name of the function being traced, foo in this case. The second argument is bound to the result of the call to foo. The function txfunc should print some sort of "exiting foo" message.

(foo evfcn evfunc) - This tells *trace* that the form evfunc should be evaluated to get the value of foo applied to its arguments. This option is a bit different from the other special options since evfunc is usually an expression, not just the name of a function, and that expression is specific to the evaluation of function foo. The argument list to be applied is available as T-arglist.

(foo printargs prfunc) - This tells *trace* to use prfunc to print the arguments to be applied to the function foo. prfunc should be a lambda of one argument. You may want to use this option if you want a print function which can handle circular lists. This option works only if you do not specify your own **traceenter** function. Specifying the option **lprint** is just a simple way of changing the printargs function to the level printer.

(foo printres prfunc) - This tells *trace* to use prfunc to print the result of evaluating foo. prfunc should be a lambda of one argument. This option works only if you do not specify your own **traceexit** function. Specifying the option **lprint** changes printres to the level printer.

You may specify more than one option for each function traced. For example:

(trace (foo if (eq 3 (arg 1)) break lprint) (bar evalin (print xyzzy)))

This tells *trace* to trace two more functions, foo and bar. Should foo be called with the first argument *eq* to 3, then the entering foo message is printed with the level printer. Next it enters a trace break loop, allowing you to evaluate any lisp expressions. When you exit the trace break loop, foo is applied to its arguments and the resulting value is printed, again using the level printer. Bar is also traced, and each time bar is entered, an entering bar message is printed and then the value of xyzzy is printed. Next bar is applied to its arguments and the result is printed. If you tell *trace* to trace a function that is already traced, it first *untraces* it. Thus, if you want to specify more than one trace option for a function, you must do it all at once. The following is *not* equivalent to the preceding call to *trace* for foo:

(trace (foo if (eq 3 (arg 1))) (foo break) (foo lprint))

In this example, only the last option, lprint, is in effect.

If the symbol $tracemute is given a non nil value, printing of the function name and arguments on entry and exit is surpressed. This is particularly useful if the function you are tracing fails after many calls to it. In this case, you would tell *trace* to trace the function, set $tracemute to t, and begin the computation. When an error occurs, you can use *tracedump* to print out the current trace frames.

Generally, the trace package has its own internal names for the Lisp functions it uses, so that you can feel free to trace system functions like *cond* and not worry about adverse interaction with the actions of the trace package. You can trace any type of function: lambda, nlambda, lexpr, or macro, whether compiled or interpreted, and you can even trace array references. However, you should not attempt to store in an array that has been traced.

When you are tracing compiled code, keep in mind that many function calls are translated directly to machine language or other equivalent function calls. A full list of open-coded functions is listed at the beginning of the Liszt compiler source. *Trace* does a *(sstatus translink nil)* to insure that the new traced definitions it defines are called instead of the old untraced ones. You may notice that compiled code runs slower after this is done.

(traceargs s_func [x_level])

WHERE: If x_level is missing, it is assumed to be 1.

RETURNS: The arguments to the x_level*th* call to traced function s_func are returned.

(tracedump)

SIDE EFFECT: The currently active trace frames are printed on the terminal. It returns a list of functions untraced.

(untrace [s_arg1 ...])

RETURNS: A list of the functions that were untraced.

NOTE: If no arguments are given, all functions are untraced.

SIDE EFFECT: The old function definitions of all traced functions are restored except in the case where it appears that the current definition of a function was not created by trace.

CHAPTER 12

Liszt - the Lisp compiler

12.1. General strategy of the compiler

The purpose of the Lisp compiler, Liszt, is to create an object module that, when brought into the Lisp system using *fasl*, has the same effect as bringing in the corresponding Lisp-coded source module with *load* with one important exception: functions are defined as sequences of machine language instructions instead of Lisp S-expressions. Liszt is not a function compiler; it is a *file* compiler. Such a file can contain more than function definitions; it can contain other Lisp S-expressions, which are evaluated at load time. These other S-expressions are also stored in the object module produced by Liszt and are evaluated at fasl time.

As is almost universally true of Lisp compilers, the main pass of Liszt is written in Lisp.

12.2. Running the compiler

The compiler is normally run in this manner:

++ liszt foo

This compiles the file foo.l or foo. (The preferred way to indicate a Lisp source file is to end the file name with '.l'.) The result of the compilation is placed in the file foo.o, if no fatal errors were detected. All messages that Liszt generates go to the standard output. Normally each function name is printed before it is compiled. (However, the +q option suppresses this.)

12.3. Special forms

Liszt makes one pass over the source file. It processes each form in this way:

12.3.1. macro expansion

If the form is a macro invocation (that is, it is a list whose car is a symbol whose function binding is a macro), then that macro invocation is expanded. This is repeated until the top level form is not a macro invocation. When Liszt begins, there are already some macros defined, in fact some functions, such as defun, are actually macros. You may define your own macros as well. For a macro to be used, it must be defined in the Lisp system in which Liszt runs.

12.3.2. classification

After all macro expansion is done, the form is classified according to its *car.* If the form is not a list, then it is classified as an *other.*)

12.3.2.1. eval-when

The form of eval-when is

(eval-when (time1 time2 ...) form1 form2 ...)

where the time*i* are one of *eval, compile,* or *load.* The compiler examines the form*i* in sequence and the action taken depends on what is in the time list. If *compile* is in the list then the compiler invokes *eval* on each form*i* as it examines it. If *load* is in the list, then the compile recursively calls itself to compile each form*i* as it examines it. Note that if *compile* and *load* are in the time list, then the compiler both evaluates and compiles each form. This is useful if you need a function to be defined in the compiler at both compile time, perhaps to aid macro expansion, and at run time after the file is *fasl*ed in.

12.3.2.2. declare

Declare is used to provide information about functions and variables to the compiler. It is (almost) equivalent to

(eval-when (compile) ...).

You may declare functions to be one of three types: lambda (*expr), nlambda (*fexpr), or lexpr (*lexpr). The names in parenthesis are the Maclisp names and are accepted by the compiler as well, and not just when the compiler is in Maclisp mode. Functions are assumed to be lambdas until they are declared otherwise or are defined differently. The compiler treats calls to lambdas and lexprs equivalently, so you need not worry about declaring lexprs either. It is important to declare nlambdas or define them before calling them. Another attribute you can declare for a function is localf, which makes the function 'local'. A local function's name is known only to the functions defined within the file itself. The advantage of a local function is that is can be entered and exited very quickly and it can have the same name as a function in another file and there will be no name conflict.

Variables may be declared special or unspecial. When a special variable is lambda bound, either in a lambda, prog, or do expression, its old value is stored away on a stack for the duration of the lambda, prog, or do expression. This takes time and is often not necessary. Therefore, the default classification for variables is unspecial. Space for unspecial variables is dynamically allocated on a stack. An unspecial variable can only be accessed from within the function where it is created by its presence in a lambda, prog, or do expression variable list. It is possible to declare that all variables are special as will be shown later.

You may declare any number of things in each declare statement. A sample declaration is

```
(declare
   (lambda func1 func2)
   (*fexpr func3)
   (*lexpr func4)
   (localf func5)
   (special var1 var2 var3)
```

(unspecial var4))

You may also declare all variables to be special with *(declare (specials t))*. You may declare that macro definitions should be compiled as well as evaluated at compile time by *(declare (macros t))*. In fact, as was mentioned earlier, declare is much like *(eval-when (compile) ...)*. Thus, if the compiler sees *(declare (foo bar))* and foo is defined, then it evaluates *(foo bar)*. If foo is not defined, then an undefined declare attribute warning is issued.

12.3.2.3. (progn 'compile form1 form2 ... formn)

When the compiler sees this it simply compiles form1 through formn as if they too were seen at top level. One use for this is to allow a macro at top-level to expand into more than one function definition for the compiler to compile.

12.3.2.4. include/includef

Include and *includef* cause another file to be read and compiled by the compiler. The result is the same as if the included file were textually inserted into the original file. The only difference between *include* and *includef* is that include does not evaluate its argument and includef does. Nested includes are allowed.

12.3.2.5. def

A def form is used to define a function. The macros *defun* and *defmacro* expand to a def form. If the function being defined is a lambda, nlambda, or lexpr, then the compiler converts the Lisp definition to a sequence of machine language instructions. If the function being defined is a macro, then the compiler evaluates the definition -- thus defining the macro within the running Lisp compiler. Furthermore, if the variable *macros* is set to a non-nil value, then the macro definition also is translated to machine language and, thus, is defined when the object file is fasled in. The variable *macros* is set to t by *(declare (macros t))*.

When a function or macro definition is compiled, macro expansion is done whenever possible. If the compiler can determine that a form would be evaluated if this function were interpreted, then it macro-expands it. It does not macro-expand arguments to an nlambda unless the characteristics of the nlambda are known, as is the case with *cond*. The map functions (*map*, *mapc*, *mapcar*, and so on) are expanded to a *do* statement. This allows the first argument to the map function to be a lambda expression that references local variables of the function being defined.

12.3.2.6. other forms

All other forms are simply stored in the object file and are evaluated when the file is *fasl*ed in.

12.4. Using the compiler

The previous section describes exactly what the compiler does with its input. Generally, you do not have to worry about all that detail because files that work interpreted, work compiled. The following is a list of steps you should follow to insure that a file compiles correctly.

[1] Make sure all macro definitions precede their use in functions or other macro definitions. If you want the macros to be around when you *fasl* in the object file, you should include this statement at the beginning of the file: *(declare (macros t))*

[2] Make sure all nlambdas are defined or declared before they are used. If the compiler comes across a call to a function that has not been defined in the current file, that does not currently have a function binding, and whose type has not been declared, then it assumes that the function needs its arguments evaluated. That is, it is a lambda or lexpr and generates code accordingly. This means that you do not have to declare nlambda functions like *status* since they have an nlambda function binding.

[3] Locate all variables that are used for communicating values between functions. These variables must be declared special at the beginning of a file. In most cases, there aren't many special declarations, but, if you fail to declare a variable special that should be declared, references to those variables which are used 'free' will not access the expected values. Examining the compiler listing will provide indications of variables used 'free' but not declared 'special'. You may eliminate all such messages by adding declarations. Unusual constructions calling interpreted code with 'free' variables can still fail if called from compiled code in which those variables are not declared 'special'. Here is an example. Assume that a file contains just these three lines:

```
(def aaa (lambda (glob loc) (bbb loc)))
(def bbb (lambda (myloc) (add glob myloc)))
(def ccc (lambda (glob loc) (bbb loc)))
```

You can see that if you load in these two definitions, then (aaa 3 4) is the same as (add 3 4) and gives us 7. Suppose you compile the file containing these definitions. When Liszt compiles aaa, it assumes that both glob and loc are local variables and allocates space on the temporary stack for their values when aaa is called. Thus, the values of the local variables glob and loc do not affect the values of the symbols glob and loc in the Lisp system. Now, Liszt moves on to function bbb. Myloc is assumed to be local. When it sees the add statement, it finds a reference to a variable called glob. This variable is not a local variable to this function, and, therefore, glob must refer to the value of the symbol glob. Liszt automatically declares glob to be special, and it prints a warning to that effect. Thus, subsequent uses of glob always refer to the symbol glob. Next, Liszt compiles ccc and treats glob as a special and loc as a local. When the object file is *fasl*ed in and (ccc 3 4) is evaluated, the symbol glob is lambda-bound to 3, bbb is called and returns 7. However, (aaa 3 4) fails since when bbb is called, glob is unbound. What should be done here is to put *(declare (special glob)* at the beginning of the file.

[4] Make sure that all calls to *arg* are within the lexpr whose arguments they reference. If *foo* is a compiled lexpr and it calls *bar*, then *bar* cannot use *arg* to get at *foo*'s arguments. If both *foo* and *bar* are interpreted, this works however. The macro *listify* can be used to put all or some of
a lexpr's arguments in a list, which can then be passed to other functions.

12.5. Compiler options

The compiler recognizes a number of options that are described later. The options are typed anywhere on the command line preceded by a plus sign. The entire command line is scanned and all options recorded before any action is taken. Thus

```
++ liszt +mx foo
++ liszt +m +x foo
++ liszt foo +mx
```

are all equivalent. The meanings of the options are:

C The assembler language output of the compiler is commented. This is useful when debugging the compiler and is not normally done since it slows down compilation.

I The next command line argument is taken as a filename and loaded prior to compilation.

e Evaluate the next argument on the command line before starting compilation. For example,

```
@ liszt +e '(setq foobar "foo string")' foo
```

evaluates the earlier s-expression. Note that the shell requires that the arguments be surrounded by single quotes.

m Compile this program in Maclisp mode. The reader syntax is changed to the Maclisp syntax and a file of macro definitions is loaded in, usually named /lisp/lib/machacks. However FRANZ LISP cannot guarantee that this switch allows you to compile any given program without some change.

o Select a different object or assembler language file name. For example,

```
++ liszt foo +o xxx.o
```

compiles foo and into xxx.o instead of the default foo.o, and

```
++ liszt bar +S +o xxx.s
```

compiles to assembler language into xxx.s instead of bar.s.

q Run in quiet mode. The names of functions being compiled and various "Note"'s are not printed.

Q Print compilation statistics and warn of strange constructs. This is the inverse of the **q** switch and is the default.

r Place bootstrap code at the beginning of the object file, which, when the object file is executed, causes a Lisp system to be invoked and the object file *fasl*ed in. This is known as 'autorun' and is described later.

S Create an assembler language file only.

```
++ liszt +S foo
```

Creates the assembler language file foo.s but does not attempt to assemble it. If this option is not specified, the assembler language file is put in the temporary disk area under an automatically generated name based on the Lisp compiler's process id. Then, if there are no compilation errors, the assembler is invoked to assemble the file.

T Print the assembler language output on the standard output file. This is useful when debugging the compiler.

u Run in UCI-Lisp mode. The character syntax is changed to that of UCI-Lisp and a UCI-Lisp compatibility package of macros is read in.

w Suppress warning messages.

x Create a cross reference file.

```
++ liszt +x foo
```

not only compiles foo into foo.o but also generates the file foo.x. The file foo.x is Lisp-readable and lists for each function all functions which that function could call. The program lxref reads one or more of these ".x" files and produces a human-readable cross reference listing.

12.6. autorun

The object file that Liszt writes does not contain all the functions necessary to run the Lisp program, which was compiled. In order to use the object file, a Lisp system must be started and the object file *fasl*ed in. When the +r switch is given to Liszt, the object file created contains a small piece of bootstrap code at the beginning, and the object file is made executable. Now, when the name of the object file is given to the operating system command interpreter (shell) to run, the bootstrap code at the beginning of the object file causes a Lisp system to be started. The first action the Lisp system takes is to *fasl* in the object file that started it. In effect, the object file has created an environment in which it can run.

Autorun is an alternative to *dumplisp*. The advantage of autorun is that the object file that starts the whole process is typically small, whereas the minimum *dumplisp*ed file is very large -- one half megabyte. The disadvantage of autorun is that the file must be *fasl*ed into a Lisp system each time it is used, whereas the file which *dumplisp* creates can be run as is. Liszt itself is a *dumplisp*ed file since it is used so often and is large enough that too much time is spent *fasl*ing it in each time it is used. The Lisp cross reference program, lxref, uses *autorun*, since it is a small and rarely used program.

In order to have the program *fasl*ed in, begin execution (rather than starting a Lisp top level), the value of the symbol user-top-level should be set to the name of the function to get control. An example of this is shown next.

Suppose you want to replace the operating system date program with one written in Lisp.

```
++ cat lispdate.l
(defun mydate nil
   (patom "The date is ")
   (patom (status ctime))
   (terpr)
   (exit 0))
(setq user-top-level 'mydate)

++ liszt +r lispdate
Compilation begins with Lisp Compiler 5.2
source: lispdate.l, result: lispdate.o
mydate
%Note: lispdate.l: Compilation complete
%Note: lispdate.l:  Time: Real: 0:3, CPU: 0:0.28, GC: 0:0.00 for 0 gcs
%Note: lispdate.l: Assembly begins
%Note: lispdate.l: Assembly completed successfully
```

This changes the name to remove the ".d', (this isn't necessary).

```
++ move lispdate.o lispdate
```

This tests it out.

```
++ lispdate
The date is Sat Aug  1 16:58:33 1984
++
```

12.7. pure literals

Normally, the quoted lisp objects (literals) that appear in functions are treated as constants. Consider this function:

```
(def foo
   (lambda nil (cond ((not (eq 'a (car (setq x '(a b)))))
                      (print 'impossible!!))
                     (t (rplaca x 'd)))))
```

At first glance it seems that the first cond clause is never true, since the *car* of *(a b)* should always be *a*. However, if you run this function twice, it prints 'impossible!!' the second time. This is because the following clause modifies the 'constant' list *(a b)* with the *rplaca* function. Such modification of literal Lisp objects can cause programs to behave strangely as the earlier example shows, but, more importantly, it can cause garbage collection problems if done to compiled code. When a file is *fasl*ed in, if the symbol $purcopylits is non-nil, the literal Lisp data is put in 'pure' space; that is, it is put in space that need not be looked at by the garbage collector. This reduces the work the garbage collector must do, but it is dangerous, since if the literals are modified to point to non-pure objects, the marker may not mark the non-pure objects. If the symbol $purcopylits is nil, then the literal Lisp data is put in impure space and the compiled code acts like the interpreted code when literal data is modified. The default value for $purcopylits is t.

12.8. transfer tables

A transfer table is setup by *fasl* when the object file is loaded in. There is one entry in the transfer table for each function that is called in that object file. The entry for a call to the function *foo* has two parts whose contents are:

[1] Function address — This initially points to the internal function *qlinker.* It may some time in the future point to the function *foo*, if certain conditions are satisfied. (See later for more on this.)

[2] Function name — This is a pointer to the symbol *foo.* This is used by *qlinker.*

When a call is made to the function *foo*, the call actually is made to the address in the transfer table entry and ends up in the *qlinker* function. *Qlinker* determines that *foo* is the function being called by locating the function name entry in the transfer table†. If the function being called is not compiled, then *qlinker* just calls *funcall* to perform the function call. If *foo* is compiled and if *(status translink)* is non-nil, then *qlinker* modifies the function address part of the transfer table to point directly to the function *foo.* Finally, *qlinker* calls *foo* directly . The next time a call is made to *foo* the call goes directly to *foo* and not through *qlinker.* This results in a substantial speedup in compiled code to compiled code transfers. A disadvantage is that no debugging information is left on the stack, so *showstack* and *baktrace* are useless. Another disadvantage is that if you redefine a compiled function either through loading in a new version, or interactively defining it, then the old version may still be called from compiled code, if the fast linking described earlier has already been done. The solution to these problems is to use *(sstatus translink value).* If value is

nil All transfer tables are cleared. That is, all function addresses are set to point to *qlinker.* This means that the next time a function is called *qlinker* is called and looks at the current definition. Also, no fast links are set up since *(status translink)* is nil. The result is that *showstack* and *baktrace* work and the function definition at the time of call is always used.

on This causes the Lisp system to go through all transfer tables and set up fast links wherever possible. This is normally used after you have *fasl*ed in all of your files. Furthermore, since *(status translink)* is not nil, *qlinker* makes new fast links if the situation arises, which is not likely unless you *fasl* in another file.

t This or any other value not previously mentioned just makes *(status translink)* be non-nil and, as a result, fast links is made by *qlinker* if the called function is compiled.

12.9. Fixnum functions

The compiler generates inline arithmetic code for fixnum only functions. Such functions include +, −, *, /, \, 1+ and 1−. The code generated is much faster than using *add*, *difference*, etc. However it only works if the arguments to and results of the functions are fixnums. No type checking is done.

† *Qlinker* does this by tracing back the call stack until it finds the *calls* machine instruction that called it. The address field of the *calls* contains the address of the transfer table entry.

CHAPTER 13

TPL: the Top-Level Listener

13.1. Introduction

Tpl is the default top-level "listener" for FRANZ LISP. This program reads input from the keyboard, evaluates the input, and prints the value(s) returned by the evaluation. While it is possible for a Lisp system to provide just this bare "read-eval-print loop" and be quite useful, most users prefer a more "user-friendly" top level.

Part of the attraction of Lisp is that this or any other top-level interface to the user is easy to change for special uses. In many cases, serious application programs replace tpl with a different top level. Several widely-used programs replace it with an algebraic infix parser; others use a natural language (English) parser, or a database command language. Since the source text for tpl is available in the lisp library as tpl.l, the code can be used by programmers as a basis for other top-level "listeners".

13.2. A top-level for debugging Lisp programs

The particular goal of *this* top-level listener is to provide a natural link to FRANZ LISP debugging facilities, and support the programmer with various mechanisms to keep track of command histories, simplify the setting and examination of debugging flags, etc. Tpl provides enhanced debugging facilities, history command substitution, a file package, frame evaluation, and lisp stack manipulating functions.

13.3. How to use tpl

If you start up the FRANZ LISP system as delivered, tpl is the program which reads your keyboard input and determines what is done with it. Tpl prints a prompt "=> ".

Any input that is valid use at any level in FRANZ LISP will have exactly the same meaning to tpl, with the exception that new lines beginning with a question mark (?) are interpreted as special commands to tpl, and not passed immediately to Lisp for evaluation.

In the description of the tpl commands, the notation [...] indicates optional arguments, and the notation [a | b] means 'a' or 'b' or neither.

?help [topic]

prints the help text associated with a particular command within tpl. 'topic' can be selected from one of the tpl keywords; if no argument is given, a list of the keywords and a brief summary of their meanings is printed. For example:

```
=> ?help history
```
prints an explanation of what you get when you type "?history".
```
=> ?help ?
```
similarly, prints an explanation of "??"

?? [location-specifier]
: finds a particular previous command line identified by the location-specifier, and re-executes it as if it were retyped to tpl directly. If the location-specifier is a non-numeric ymbol, tpl scans backward through the commands to find an expression whose 'car' is equal to the symbol. For example, "?? print" will repeat the last command beginning (print ....). It is not necessary to type the whole symbol: you can type an asterisk to match "the rest of the atom". For example, "?? pr *" will also find (print ...) unless some more recent command also begins with (pr... ...). If the location-specifier is a positive integer, the command line with that number is re-executed. If location-specifier is a negative number (-N) then the Nth previous command is redone. If location-specifier is not given, then the last command is redone. Thus "??" is equivalent to "?? -1".

?his[tory] [r]
: prints the history list of recent Lisp commands. You may set the variable tpl-history-show to alter the number of the most recent commands which are displayed. Invoking the 'r' option will display the results of those commands.

?re[set]
: is equivalent to typing the Lisp command (reset).

?tr [fn1 fn2 ...]
: traces 'fn1 fn2 ...'. While this will usually be a simple enumeration of functions to be traced, more options can be passed to the trace function by using (name option-list) expressions as in the normal trace package (e.g. "?tr (foo break)").

?untr [fn1 fn2 ...]
: untraces the specified functions, or if given no argument, will untrace all traced functions.

?step [t | fn1 fn2 ...]
: initiates a mode of single-step execution of Lisp, If 't' is the argument, this is done immediately. Otherwise stepping is initiated upon entry to any of the functions fn1 fn2, ... The next two commands control this mode.

?soff
: turns stepping off.

?sc [n]
: (step counter) steps 'n' times, then enters a Lisp (break). 'n' defaults to one. if 'n' is the symbol 'inf' then steps forever without breaking. When in stepping mode, typing a <return> is equivalent to ?sc 1.

?state [sym1 val1 ...]
: prints/changes the state of tpl flags and variables. The variables listed by '?state' are the only ones which can be changed via this command. Sym1 is set to val1, etc.

?prt 'pop and retry': does a ?pop, followed by a retry of the command which caused the last break to be entered. This is one of the most commonly useful tpl commands, since it resumes computation, probably after a fix-up, from the last error.

?ld [file1 file2 ...]
: loads the given files, or re-loads the just previously loaded file if no arguments are supplied.

?fast sets up Lisp for fast execution, by: turning off debugging mode (?debug off), setting translink to 'on', and setting displace-macros to t. Debugging information will be lost when this mode is entered. These settings generally would be used during the running of compiled programs which are known to be correct and in which high-speed execution is important.

?pop pops up one break level. If at the top level, it has no effect.

?ret [val]
: returns 'val' from this break level. If the argument is missing, nil is returned. The value is returned to the function which produced the error, allowing it to continue. The break must have originated from invoking the break function or from the signalling of a continuable error. The argument 'val' is evaluated.

?zo views (Zooms) a portion of the Lisp stack. You may use ?up and ?dn to move the pointer to the current stack frame. Prior to using this you should execute the tpl command ?debug so that sufficient information is stored on the stack.

?dn [n]
: without arguments, moves the current frame pointer down one level and executes a ?zo. If n is given, it moves that number of frames down. The stack grows upward, so the oldest frames are on the bottom.

?up [n]
: is the same as ?dn, except the current frame pointer is moved in the up direction.

?ev symbol
: determines the value of symbol in the context of the current stack frame, as if the frames above the current one had not yet been created.

?pp "pretty prints" the current frame with neat indentation and without ellipsis. Ordinarily this would be used after ?zo is used to locate a frame of interest.

<eof>
: (a single character, without a '?' prefix) pops up one break level if it is typed to tpl from a keyboard input stream. Depending upon the catching of signals, if it is typed on the top level, may be used to exit from lisp. Lisp.

13.4. Tpl special symbols

The following are special symbols:

user-top-level
: may be bound to a function (i.e. a lambda-expression, or more likely a symbol which is the name of a defined function), which will be evaluated instead of (tpl) as the read-eval-print loop.

top-level-prompt
: if bound to a non-nil S-expression, will be used as the top-level prompt.

top-level-init
: if bound to a function, will be used to initialize tpl. Normally, the lisprc file is read, and the copyright notice and version number are printed.

top-level-print
: if bound to a function, will be used to print values returned by the read-eval part of the read-eval-print loop.

tpl-number-prompt
: if t, will cause tpl to print an index number with the '=>' prompt.

tpl-prinlevel
: the maximum nesting level to print lists. Beyond that point, lists are abbreviated to &.

tpl-prinlength
: the maximum length to print a list. Beyond that point, lists are abbreviated to &.

tpl-history-show
: the number of history items to show with the ?history command.

displace-macros

if t, then displace macros with their expansion. This will speed execution, occupy more space, and possibly interfere with debugging by replacing macro calls by possibly obscure expansions.

13.5. A sample session with TPL

```
      ; first we load in a factorial function:
=> ?ld fact
[load fact.l]
(fact)
; we 'prettyprint' the fact function
=> (pp fact)
(def fact
  (lambda (n)
    (cond ((= n 0) (bug)) ((times n (fact (sub1 n)))))))

t
; we somehow fail to notice that there is a bug in when n equals 0
; so we try it out:
=> (fact 10)
Error: eval: Undefined function  bug
Form: (fact 10)
; we could use showstack or backtrace to find out what is wrong, but
; for this example we decide that we want to use the more powerful
; ?zo (zoom) function. In order to use ?zo, we have to be in debugging
; mode before the error occurs. Since we aren't, we decide to turn
; on debugging and run the function again so it gets an error
c{1} ?debug
Debug is on
t
; the ?prt function pops up a break level and retries the function that
; caused the error. The line just below which says '=> (fact 10)' was
; printed by tpl. It was not typed by the user. tpl is showing the
; function it is retrying.
c{1} ?prt
=> (fact 10)
Error: eval: Undefined function  bug
Form: (fact 10)
; the error occurred again. Now that we are in debug mode, we can do
; a zoom
c{1} ?zo
Should I re-calc the stack(y/n):y
*** top ***
// current \
(bug)
(cond ((= n 0) (bug)) ((times n &)))
(fact (sub1 n))
(times n (fact (sub1 n)))
nil
; it shows that the current frame is the top frame and that is the
; evaluation of (bug).
; We can ask what the value of n is at this point:
c{1} ?ev n
```

```
0
; it is pretty clear that the problem is that the bug function is
; undefined. Before we correct it, we show a bit more of tpl.  Here
; we go down five frames:
c{1} ?dn 5
(cond ((= n 0) (bug)) ((times n &)))
(fact (sub1 n))
(times n (fact (sub1 n)))
(cond ((= n 0) (bug)) ((times n &)))
// current \
(fact (sub1 n))
(times n (fact (sub1 n)))
(cond ((= n 0) (bug)) ((times n &)))
(fact (sub1 n))
nil
; now we inquire as to n's value at this point in the execution:
c{1} ?ev n
2
; Now let us fix the bug. The function fact could be edited by using
; editf (see chapter 16), or we can define (bug) as returning 1.
c{1} (defun bug () 1)
bug
; Now we pop and retry. Notice that we didn't have to move the
; current frame to the top.
c{1} ?prt
=> (fact 10)
3628800
; this time it works.
=>
;
; sample session 2.
; things work slightly differently when fact is compiled
;
=> ?ld fact
[fasl fact.o]
(fact)
; we turn on debugging since we know that an error will occur
=> ?debug
Debug is on
t
=> (fact 10)
Error: Undefined function called from compiled code  bug
Form: (fact 10)
; look at the stack
c{1} ?zo
Should I re-calc the stack(y/n):y
*** top ***
// current \
a:(fact (0))
a:(fact (1))
a:(fact (2))
a:(fact (3))
nil
; The call to (bug) isn't visible on the stack, since undefined functions
; are detected in compiled code in a different manner.
; Notice that the frames are preceded by 'a:' and the arguments look
; unusual. This is an 'apply' form, which you may think of as a shorthand for
; (apply 'fact '(2)). This is how frames showing calls from compiled
; code look.
c{1} (defun bug () 1)
bug
; again we fix the bug and retry
c{1} ?prt
=> (fact 10)
3628800
; and it works.
=>
```

13.6. The File Subsystem

The FRANZ LISP file package helps support the residential environmental style of lisp programming in which most or all program editing is done within lisp itself, probably using *editf, editv, editp* to create and debug programs. Although ordinary data files are used by the file package to store the programmer's alterations to the function definitions and other data that persist between runnings of programs, management of those files is controlled by the FRANZ LISP system.

As an interactive session proceeds, the file package (in cooperation with the top level) tracks the changes the user makes to the lisp environment. Those changes are of three types: function (or macro) definitions, values of variables (symbols) altered, and properties of symbols altered. At any point the user can find out what has been changed by typing ?changed to the top level, which will print the information out in a table form.

Each item (function, value or property) may be associated with a file. The list printed by '?changed' will show the associated file for each changed item. In order to save a change, the user must request that the associated file be written out (using '?fileout', described below). If an item doesn't have an associated file, then one can be declared using '?add-function', '?add-var' or '?add-prop', depending on the type of item.

Command Summary for the File Package

?filein [name1 name2 ...]
: loads the named files using a read-eval loop, printing the names (not the values) being loaded. The files being read should have been written with ?fileout. If no files are named as arguments, a list of all previously loaded files is returned. The command

?fileout [name1 name2 ...]
: writes the given files if any of the items in the file have changed. With no arguments, all files that need updating are rewritten.

?changed
: reports on those data items which have changed but not stored on files, and the names of associated files.

?add-function filen fcn1 [fcn2 ...]
: adds 'fcn1', 'fcn2', etc. to the list of functions associated with the the file 'filen'. The file 'filen' should either not exist or should be the name of a file which has been loaded with ?filein.

?add-var filen var1 [var2 ...]
: adds the given symbols 'var1', 'var2', etc. to the list of symbols stored in the file

'filen'. The file 'filen' should either not exist or should be the name of a file which has been loaded with ?filein.

?add-prop
: adds symi's indi property to the list of properties stored in the file 'filen'. The file 'filen' should either not exist or should be the name of a file which has been loaded with ?filein.

?rem-function filen fcn1 [fcn2 ...]

?rem-var filen var1 [var2 ...]

?rem-prop filen (sym1 ind1) [(sym2 ind2) ...]
: remove the named items from filen. They do this by deleting the association of the item from the file. When the file is next written with '?fileout', the items will not be written out.

?whichfile fcn | var | (symbol ind) ...
: for each item (which can be a function, variable or (symbol ind)), prints the associated filename, if there is one. If a symbol is both a function and a variable (in different files), both associated files are printed.

?filestatus [filen1 filen2 ...]
: prints the names of the items in each file listed. If no filenames are given, prints a summary status report of all files.

Backup Variables in the File Package

There are several variables which the user might wish to alter to assist in backup maintenance:

file:backup-prepend
: is a string (or symbol) to prepend to the filename to generate a backup filename during a '?fileout'

file:backup-append
: is a string (or symbol) to append to the filename to generate a backup filename during a '?fileout'

13.7. File subsystem implementation notes

The file package maintains a database of knowledge about files. For each file it keeps track of the items stored in that file. The file package also maintains a list of items which have changed, called the changed-list.

Filein notes:

Filein recognizes three types of items: functions, variables and properties.

A *function* item has this form: (kwd functionname anything ...) where kwd is an element of the list which is the value of file:function-modifiers. The initial value of file:function-modifiers is (defun def defmacro). The user may wish to add something to this list to read in a file not created by '?fileout'. The '?fileout' function-printing function will only use the 'def' form, which provides a superset of the capabilities of the other forms.

A *variable* item has this form: (kwd variablename anything ...) where kwd is an element of the list which is the value of file:variable-modifiers. The initial value of file:variable-modifiers is (setq).

A *property* item has this form: (kwd symbol anything indicator) where kwd is an element of the list which is the value of file:property-modifiers. The initial value of file:property-modifiers is (defprop). Note that the symbol and indicator are not evaluated before they are added to the list of items, so 'putprop' is not a valid kwd to be added to file:property-modifiers.

Fileout notes:

Files created by ?fileout contain only a few types of forms (def, setq and defprop). If the file is edited externally from the lisp system and other forms are inserted (such as declares or comments), and then the file is filed in-and-out, the other forms will be lost. It is also important to keep forms syntactically correct (e.g. with parentheses balanced), because then forms following the error will not be read in to the lisp system. It is generally safe to edit or merge files to add, delete or alter *syntactically proper* definitions of the forms already known to the file package.

?fileout performs the following sequence of operations: it opens up a file in /tmp and writes all items in the file. As each item is written, it is also removed from the global changed-list if it was on that list. If a file with the same name as the one being written exists then ?fileout will preserve the previous file by changing its name, if the user has set one or both of the variables file:backup-prepend and file:backup-append. If both of these variables are nil, then a backup will not be done. If file:backup-prepend is non-nil, then its value should be a symbol or string which will be prepended to the filename in order to create the backup name. Likewise file:backup-append will be appended to the filename to create the backup name. If both variables are non-nil, then both will be used. Finally, the file in /tmp is renamed to the name of the file being filed out.

A caution is appropriate: suppose you start lisp and define the function 'solveit'. You would like to add this function to the file 'eqn.l' which you created earlier and which already contains a number of functions. Your first thought may be to type:
?add-function eqn.l solveit
Since the file 'eqn.l' exists on the disk but hasn't been loaded yet, the file package is ignorant of any functions other than 'solveit' associated with 'eqn.l'. Executing
?fileout eqn.l
would cause the contents of 'eqn.l' to be replaced with the definition of the single function 'solveit'. As a guard against this situation, the file package asks you if you want to abort the ?add-function operation when you mention an existing file which however has not been read-in. It is best to type 'yes' at this point, then
?filein eqn.l
and then
?add-function eqn.l solveit

CHAPTER 14

Miscellaneous Topics

14.1. Keyword Arguments

FRANZ LISP new offers an alternative function calling convention: keyword arguments. This feature is borrowed from Common Lisp and is completely compatible with Common Lisp keywords.

Keyword arguments are used in functions which are largely given as an interface to the user to some *package* which has a large number of selectable parameters. When calling a function of this type, it would be hard to remember the ordering of arguments, since this ordering is possibly random. Keyword arguments offer the ability to *tag* the parameters to a function so they can be given in any order and so they are easy to remember.

In Common Lisp, any symbol which contains a colon (:) as the first character is called a *keyword*, and is then treated very specially: keywords evaluate to themselves. This means :foo evaluates to :foo, where normal symbols would need a quote (') in front of them to evaluate this way. Currently, in FRANZ LISP, keywords do not exist, and they need the preceding quote to make them evaluate to themselves. In the near future this limitation will be lifted with the implementation of the packages facility.

A keyword is a tag for an argument to a function, and keyword-argument sequences come in pairs, in that order. To illustrate the use of keywords to tag arguments, consider the function *make-hash-table* (explained below) which takes up to four arguments. A possible invocation might be *(make-hash-table ':size 100)*. The :size keyword *tags* the argument 100 as the *size* parameter to the function make-hash-table. Also, note that the following forms *(make-hash-table ':size 10 ':test 'equal)* and *(make-hash-table ':test 'equal ':size 10)* are equivalent.

Whenever a keyword appears in a function definition in this manual, it will be named *without* the quote, and it is implied that the user must supply the quote until the packages facility has been installed.

14.2. Hash Tables

A hash table is an object that can efficiently map one object to another. Each hash table is a collection of entries, each of which associates a unique *key* with a *value*. There are functions to add, delete, and find entries based on a particular key. Finding a value in a hash table is relatively fast compared to looking up values in, for example, an assoc list or property list.

The hash table is *not* a true data type, but rather a type which is constructed from objects of the type **vector**. Because of this, the vector predicate returns a non-*nil* value when handed a hash table. It should be noted that using *vset* to set a element of the hash table will yield unpredictable results.

Adding a key to a hash table modifies the hash table, and is therefore a destructive operation.

There are two different kinds of hash tables: those that use the function *equal* for the comparing of keys, and those that use *eq*, the default. When a hash table is created, the type of comparator is set. If "eq" is chosen as the comparator, and a lookup of a key is being performed, then the given key is compared to the keys in the table using "eq".

Hashing provides an efficient basis for the construction of the *packages* facility and for various sorts of data retrieval techniques.

This hash table package is completely compatible with the one in Common Lisp.

14.2.1. Functions

(**makeht** 'x_size ['s_test])

RETURNS: A hash table with x_size hash buckets. If present, s_test is used as the test to compare keys in the hash table, the default being *eq*. Other valid values for s_test are *equal* or **nil** (to use the the default comparator *eq*).

NOTE: This function in not present in Common Lisp.

(**make-hash-table** :size :test :rehash-size :rehash-threshold)

RETURNS: A hash table object of some number of buckets, given by the :size argument. If the function to compare hash table keys is be something other than *eq*, then the :test argument should be used to set this (the choices are *eq*, *equal*, or **nil**).

NOTE: The :rehash-size and :rehash-threshold keywords are ignored at this time, and no rehashing is done. Please see the Keywords section in this chapter for an explanation of the colon.

(**hash-table-p** 'H_arg)

RETURNS: t if H_arg is a hash table.

NOTE: Hash tables are really vectors with the car of the their property list *eq*ual to *hash-table*.

(**gethash** 'g_key 'H_htab ['g_defval])

RETURNS: two values, first, the value associated with the key g_key in hash table H_htab, or **nil** if the key was not in the table, and then a Boolean value to indicate whether or not there was a match. If g_defval is given and there is no entry in the hash table, then g_defval is returned.

NOTE: *setf* may be used to set the value associated with a key.

(addhash 'g_key 'H_htab 'g_val**)**

RETURNS: g_key, after adding it with its value g_val to the hash table.

(remhash 'g_key 'H_htab**)**

RETURNS: **t** if there was an entry for g_key in the hash table H_htab, **nil** otherwise. In the case of a match, the entry and associated object are removed from the hash table.

(maphash 'u_fun 'H_htab**)**

RETURNS: nil.

NOTE: The function u_fun is applied to every element in the hash table H_htab. The function should expect two arguments: the key and value of an element. The mapped function should not add or delete objects from the table because the results would be unpredictable.

(clrhash 'H_htab**)**

RETURNS: the hash table cleared of all entries.

(hash-table-count 'H_htab**)**

RETURNS: the number of entries in H_htab. Given a hash table with no entries, this function returns zero.

```
; make a vanilla hash table using "eq" to compare items...
-> (setq black-box (makeht 20))
hash-table[26]
-> (hash-table-p black-box)
t
-> (hash-table-count black-box)
0
-> (setf (gethash 'anykey black-box) '(this list is the value))
anykey
-> (gethash 'anykey black-box)
(this list is the value)
-> (hash-table-count black-box)
1
-> (addhash 'composer black-box 'franz)
composer
-> (gethash 'composer black-box)
franz
-> (maphash '(lambda (key val) (msg "key=" key ",value=" value N))
black-box)
key=composer,value=franz
key=anykey,value=(this list is the value)
nil
-> (clrhash black-box)
hash-table[26]
-> (hash-table-count black-box)
0
-> (maphash '(lambda (key val) (msg "key=" key ",value=" value N))
black-box)
nil

; here is an example using "equal" as the comparator
-> (setq ht (makeht 10 'equal))
hash-table[16]
-> (setf (gethash '(this is a key) ht) '(and this is the value))
(this is a key)
-> (gethash '(this is a key) ht)
(and this is the value)
; the reader makes a new list each time you type it...
-> (setq x '(this is a key))
(this is a key)
-> (setq y '(this is a key))
(this is a key)
; these two lists are really different lists
; they are "equal" but not "eq"
-> (equal x y)
t
-> (eq x y)
nil
; since we are using "equal" to compare keys, we are OK...
-> (gethash x ht)
(and this is the value)
-> (gethash y ht)
(and this is the value)
```

14.3. Multiple Value Returns

Sometimes a function logically needs to return more than one value. A function returning a complex number might return the real and imaginary parts separately, using multiple value returns. Only those functions which expect multiple values can receive them, and thus the default is to return a single value. The mechanism for using multiple values is explained below.

There are special functions which must be used to produce and receive multiple value. If a called function produces multiple values and the calling function does not request them, then all but the first value are discarded. If no values are produced, then the caller receives **nil** for a value. The maximum number of multiple values which can be returned by a functions is bound to the global variable **multiple-values-limit**.

The multiple value facility is completely compatible with the one in Common Lisp.

Here are the functions to produce and receive multiple values:

(values ['g_arg1 ... 'g_argn])

RETURNS: g_arg1, or **nil** if given no arguments. The g_argi are returned as multiple values, needing one of the special forms below to receive them.

(values-list 'l_arg)

RETURNS: the car of l_arg, and the elements in the list l_arg as multiple values. This form is equivalent to (apply 'values 'l_arg).

(multiple-value-call 'u_fun 'g_form1 ['g_form2 ...])

RETURNS: the result of calling u_fun with the results of all g_formi as arguments.

(multiple-value-list 'g_form)

RETURNS: a list of the multiple values returned by g_form. This form is equivalent to (multiple-value-call #'list 'g_form).

(multiple-value-prog1 'g_form1 ['g_form2 ...])

RETURNS: the values produced by g_form1, after evaluating all g_formi.

(multiple-value-setq 'l_varlist 'g_form)

RETURNS: the first value returned by g_form after setting each variable in l_varlist to the corresponding value returned by g_form (the first variable gets the first value, and so on).

NOTE: If there are more variables than returned values, then the remaining variables are given the value of **nil**.

(multiple-value-bind 'l_varlist 'g_values-form 'g_form1 ['g_form2 ...])

RETURNS: the result of evaluating g_formi. The variables in l_varlist are bound to the values returned by g_values-form, and then all the g_formi are evaluated.

14.4. Miscellaneous Functions

(map-over-oblist 'u_fun)

RETURNS: nil..

NOTE: u_fun is applied to every element in the oblist. When packages are implemented, this function will disappear, and maphash will work on the oblist.

(dolist (s_var l_form g_resultform) g_form)

RETURNS: if present, g_resultform, **nil** otherwise. Dolist provides a mechanism to iterate over the elements of the list l_form, successively binding the elements to s_var, while executing the body of the loop g_form.

(dotimes (s_var i_countform g_resultform) g_form)

RETURNS: if present, g_resultform, **nil** otherwise. Dotimes provides a mechanism to iterate over a sequence of integers. First, i_countform is evaluated to produce an integer, and then evaluates g_form once for each integer from zero (inclusive) to i_countform (exclusive), in order, binding s_var to this integer.

(do l_vrbs l_test g_expl ...)

RETURNS: the value of the last form in the cdr of l_test, or a value explicitly given by a return evaluated within the do body.

NOTE: A feature has been added for Common Lisp compatibility. Each var-init-repeat form may be an atom, in which case it is bound to nil. Thus 'foo' may be regarded as an abbreviation for the var-init-form '(foo nil)'.

(do* l_vrbs l_test g_expl ...)

RETURNS: the value of the last form in the cdr of l_test, or a value explicitly given by a return evaluated within the do body.

NOTE: This is the Common Lisp *do** form. It is very similar to *do* except that: (1) The var-init-repeat forms are evaluated sequentially rather than simultaneously. (2) There is no analogue of the old-style maclisp *do*. In particular, l_vrbs must be a list of var-init-repeat forms.

(with-keywords 'l_keys 'l_keydefs 'g_form1 ['g_form2 ...])

NOTE: Please see the Keywords section in this chapter.

RETURNS: The result of evaluating the g_formi. The elements of the list l_keys represent the calling parameters to a function using keyword arguments as tags, and takes the form of *(keyword1 value1 keyword2 value2 ...)*. This is a list of the arguments given to a function. The l_keydefs define which keywords are valid, what variable the value will be bound to, and what the default value is is the keyword-value pair does not exist in l_keys. The g_form are forms which are evaluated after binding the variable in the l_keydefs list.

```
-> (setq x 100)
100
-> (dolist (x '(a b c d e f g) 'result) (msg x " "))
a b c d e f g result
-> x
100
-> (dotimes (x 10) (msg x ","))
0,1,2,3,4,5,6,7,8,9,nil
-> x
100

;; here is the definition of make-hash-table using
;; with-keywords:

(defun make-hash-table (&rest keys)
  (with-keywords keys ((:test test 'eq)
                       (:size size 20)
                       (:rehash-size dummy nil)         ; a no-op
                       (:rehash-threshold dummy nil))   ; a no-op
                 (makeht size test)))
```

(make-vector-float 'x_size)

RETURNS: a vector for storing x_size float values to be passed to C routines.

(vset-float 'v_vec 'x_index 'f_value)

RETURNS: f_value, after setting the x_index'th element of v_vec to f_value. v_vec should have been created by *make-vector-float*

(vref-float 'v_vec 'x_index)

RETURNS: the x_index'th element of v_vec. v_vec should have been created by *make-vector-float*

14.5. Sharp Sign Macro Syntax

The sharp sign macro (using the **#** character) in a part of the standard lisp reader. Among it's uses, are to have the reader evaluate expressions short hand notations, and reading numbers in other than the standard radix.

Sharp sign macros are invoked by a two character sequence, consisting of the sharp (or pound) sign (**#**), followed by an additional character, which will be discussed shortly. Here are the macros, listed by the two character sequences:

#' This is an abbreviation of *function*. *#'foo* is read as *(function foo)*.

#(read the following forms, up to a right parenthesis, into a lisp vector.

#,

#. The following form is evaluated before being returned from the read.

#\ If the form after the #\ is a one of *newline, space, rubout, page, tab, backspace, return, linefeed, vert, sharp,* then the character code for the above is read. And otherwise, the form must be a single character, and the form read is the character code for that character.

#| The characters read between this marker and the characters |# are discarded. This form does nest, so "#| #| |# |#" is valid.

#+

#- The following form is read (or not read) depending on whether (or not) the following form is on the *(status features)* list. This is how read-time conditionalization is done.

#o

#O read the following form as an octal number.

#x

#X read the following form as a hexidecimal number.

CHAPTER 15

The Lisp Stepper and FIXIT

Several handy debugging tools are described in detail in this chapter.

15.1. Simple Use Of Stepping

(step s_arg1...**)**

NOTE: The Lisp "stepping" package is intended to give the Lisp programmer a facility analogous to the Instruction Step mode of running a machine language program. The user interface is through the function (fexpr) step, which sets switches to put the Lisp interpreter in and out of "stepping" mode. The most common *step* invocations follow. These invocations are usually typed at the top-level, and will take effect immediately (i.e. the next S-expression typed in will be evaluated in stepping mode). The facilities of this package are similar to those in the 'tpl' system, but can be used separately. The capabilities of the two systems will be unified and expanded in the future.

```
(step t)        ; Turn on stepping mode.
(step nil)      ; Turn off stepping mode.
```

SIDE EFFECT: In stepping mode, the Lisp evaluator will print out each S-exp to be evaluated before evaluation, and the returned value after evaluation, calling itself recursively to display the stepped evaluation of each argument, if the S-exp is a function call. In stepping mode, the evaluator will wait after displaying each S-exp before evaluation for a command character from the console.

STEP COMMAND SUMMARY

<return>	Continue stepping recursively.
c	Show returned value from this level only, and continue stepping upward.
e	Only step interpreted code.
g	Turn off stepping mode. (but continue evaluation without stepping).
n <number>	Step through <number> evaluations without stopping
p	Redisplay current form in full (i.e. rebind prinlevel and prinlength to nil)
b	Get breakpoint
q	Quit
d	Call debug

15.2. Advanced Features

15.2.1. Selectively Turning On Stepping

If

(step foo1 foo2 ...)

is typed at top level, stepping will not commence immediately, but rather when the evaluator first encounters an S-expression whose car is one of *foo1, foo2*, etc. This form will then display at the console, and the evaluator will be in stepping mode waiting for a command character.

Normally the stepper intercepts calls to *funcall* and *eval*. When *funcall* is intercepted, the arguments to the function have already been evaluated but when *eval* is intercepted, the arguments have not been evaluated. To differentiate the two cases, when printing the form in evaluation, the stepper prints intercepted calls to *funcall* with "f:". Calls to *funcall* are normally caused by compiled Lisp code calling other functions, whereas calls to *eval* usually occur when Lisp code is interpreted. To step through only calls to eval, use: *(step e)*

15.2.2. Stepping With Breakpoints

Step is turned off for the duration of error breaks, but not by explicit use of the break function. Executing *(step nil)* inside a error loop will turn off stepping globally, i.e. within the error loop, and after return the return from the break loop.

15.3. Overhead of Stepping

If stepping mode has been turned off by *(step nil)*, there is no execution overhead for having the stepping packing in your Lisp. If one stops stepping by typing "g", every call to eval incurs a small overhead--several machine instructions, corresponding to the compiled code for a simple cond and one function pushdown. Running with *(step foo1 foo2 ...)* can be more expensive, since a 'member' computation of the car of the current form into the list *(foo1 foo2 ...)* is required at each call to eval.

15.4. Evalhook and Funcallhook

For 'step' and potentially other user-written functions to gain control of the evaluation process, hooks were installed in the FRANZ LISP interpreter. In fact there are two hooks and they have been strategically placed in the two key functions in the interpreter: *eval* (which controls execution of interpreted code) and *funcall* (which controls compiled code if *(sstatus translink nil)* has been executed). The hook in *eval* is compatible with MacLisp, but there is no MacLisp equivalent of the hook in *funcall*.

To arm the hooks two forms must be evaluated: *(*rset t)* and *(sstatus evalhook t)*. Once that is done, *eval* and *funcall* do a special check when they are invoked.

If *eval* is given a form to evaluate, say *(foo bar)*, and the symbol 'evalhook' is non-nil, say its value is 'ehook', then *eval* will lambda-bind the symbols 'evalhook' and 'funcallhook' to nil and will call ehook, passing *(foo bar)* as the argument. It is ehook's responsibility to evaluate *(foo bar)* and return its value. Typically ehook will call the function 'evalhook' to evaluate *(foo bar)*. Note that 'evalhook' is a symbol whose function binding is a system function described in Chapter 4, and whose value binding, if non-nil, is the name of a user written function (or a lambda expression, or a binary object) which will gain control whenever eval is called. 'evalhook' is also the name of the *status* tag which must be set for all of this to work.

If *funcall* is called on a function, say foo, and a set of already evaluated arguments, say barv and bazv, and if the symbol 'funcallhook' has a non nil value, say 'fhook', then *funcall* will lambda-bind 'evalhook' and 'funcallhook' to nil and will call fhook with arguments barv, bazv and foo. Thus fhook must be a lexpr since it may be given any number of arguments. The function to call, foo in this case, will be the *last* of the arguments given to fhook. It is fhook's responsibility to do the function call and return the value. Typically fhook will call the function *funcallhook* to do the funcall. This is an example of a funcallhook function which just prints the arguments on each entry to funcall and the return value.

```
-> (defun fhook n (let ((form (cons (arg n) (listify (1- n))))
                        (retval))
                            (patom " calling ")(print form)(terpr)
                            (setq retval (funcallhook form 'fhook))
                            (patom " returns ")(print retval)(terpr)
                            retval))
fhook
-> (*rset t) (sstatus evalhook t) (sstatus translink nil)
-> (setq funcallhook 'fhook)
calling (print fhook)          ;; now all compiled code is traced
fhookreturns nil
calling (terpr)

returns nil
calling (patom "-> ")
-> returns "-> "
calling (read nil Q00000)
(array foo t 10)               ;; to test it, we see what happens when
returns (array foo t 10)       ;; we make an array
calling (eval (array foo t 10))
calling (append (10) nil)
returns (10)
calling (lessp 1 1)
returns nil
calling (apply times (10))
returns 10
calling (small-segment value 10)
calling (boole 4 137 127)
returns 128
 ... there is plenty more ...
```

15.5. The FIXIT Debugger

FIXIT is a debugging environment for FRANZ LISP written and documented by David S. Touretzky of Carnegie-Mellon University for MacLisp, and adapted to FRANZ LISP by Mitch Marcus of Bell Labs. One of FIXIT's goals is to get a program being tested running again as quickly as possible. The user is assisted in making changes to his functions "on the fly", i.e. in the midst of execution, and then computation is resumed.

To enter the debugger type *(debug).* The debugger goes into its own read-eval-print loop. Like the top-level, the debugger understands certain special commands. One of these is help, which prints a list of the available commands. The basic idea is that you are somewhere in a stack of calls to eval. The command "bka" is probably the most appropriate for looking at the stack. There are commands to move up and down. If you want to know the value of "x" as of some place in the stack, move to that place and type "x" (or (cdr x) or anything else that you might want to evaluate). All evaluation is done as of the current stack position. You can fix the problem by changing the values of variables, editing functions or expressions in the stack etc. Then you can continue from the current stack position (or anywhere else) with the "redo" command. Or you can simply return the right answer with the "return" command.

When it is not immediately obvious why an error has occurred or how the program got itself into its current state, FIXIT comes to the rescue by providing a powerful debugging loop in which the user can:

- examine the stack
- evaluate expressions in context
- enter stepping mode
- restart the computation at any point

The result is that program errors can be located and fixed more rapidly.

The debugger can only work effectively when extra information is kept about forms in evaluation by the Lisp system. Evaluating *(*rset t)* tells the Lisp system to maintain this information. If you are debugging compiled code you should also be sure that the execute *(sstatus translink nil)*.

(debug [s_msg])

NOTE: Within a program, you may enter a debug loop directly by putting in a call to *debug* where you would normally put a call to *break*. Also, within a break loop you may enter FIXIT by typing *debug*. If an argument is given to debug, it is treated as a message to be printed before the debug loop is entered. Thus you can put *(debug |just before loop|)* into a program to indicate what part of the program is being debugged.

FIXIT Command Summary

TOP	go to top of stack (latest expression)
BOT	go to bottom of stack (first expression)
P	show current expression (with ellipsis)
PP	show current expression in full
WHERE	give current stack position
HELP	types the abbreviated command summary found in /lisp/lib/fixit.ref. H and ? work too.
U	go up one stack frame
U n	go up n stack frames
U f	go up to the next occurrence of function f
U n f	go up n occurrences of function f
UP	go up to the next user-written function
UP n	go up n user-written functions ...the DN and DNFN commands are similar, but go down ...instead of up.
OK	resume processing; continue after an error or debug loop
REDO	restart the computation with the current stack frame. The OK command is equivalent to TOP followed by REDO.
REDO f	restart the computation with the last call to function f. (The stack is searched downward from the current position.)
STEP	restart the computation at the current stack frame, but first turn on stepping mode. (Assumes the stepper is loaded.)
RETURN e	return from the current position in the computation with the value of expression e.
BK..	print a backtrace. There are many backtrace commands, formed by adding suffixes to the BK command. "BK" gives a backtrace showing only user-written functions, and uses ellipsis. The BK command may be suffixed by one or more of the following modifiers:
..F..	show function names instead of expressions
..A..	show all functions/expressions, not just user-written ones
..V..	show variable bindings as well as functions/expressions
..E..	show everything in the expression, i.e. don't use ellipsis
..C..	go no further than the current position on the stack Some of the more useful combinations are BKFV, BKFA, and BKFAV.
BK.. n	show only n levels of the stack (starting at the top). (BK n counts only user functions; BKA n counts all functions.)
BK.. f	show stack down to first call of function f
BK.. n f	show stack down to nth call of function f

15.5.1. Interaction with *trace* FIXIT knows about the standard Franz trace package, and tries to make tracing invisible while in the debug loop. However, because of the way *trace* works, it may sometimes be the case that the functions on the stack are really un*intern*ed atoms that have the same name as a traced function. (This only happens when a function is traced WHEREIN another one.) FIXIT will call attention to *trace's* hackery by printing an appropriate tag next to these stack entries.

15.5.2. Interaction with *step* The *step* function may be invoked from within FIXIT via the STEP command. FIXIT initially turns off stepping when the debug loop is entered. If you step through a function and get an error, FIXIT will still be invoked normally. At any time during stepping, you may explicitly enter FIXIT via the "D" (debug) command.

15.5.3. Multiple error levels FIXIT will evaluate arbitrary Lisp expressions in its debug loop. The evaluation is not done within an *errset,* so, if an error occurs, another invocation of the debugger can be made. When there are multiple errors on the stack, FIXIT displays a barrier symbol between each level that looks something like <------------UDF-->. The UDF in this case stands for UnDefined Function. Thus, the upper level debug loop was invoked by an undefined function error that occurred while in the lower loop.

CHAPTER 16

The Lisp Editor

16.1. Introduction

Many people use standard text editors to edit their Lisp programs. However there are also Lisp ''structure-oriented'' embedded editors which are particularly handy for the editing of Lisp programs and data. These operate in a rather different fashion, namely within a Lisp environment. Such an editor is handy for rapid fixes and re-evaluating of tests without exiting from the Lisp system. For example, you can fix a bug and then continue your computation from a break-point. The editor has its own command structure which includes the ability to evaluate arbitrary Lisp expressions.

The Lisp editor ''editf'' and its related components in FRANZ LISP differ from file/text editors in that editor commands directly change the internal structure of Lisp expressions rather than an external character representation. In particular, it is not possible for the Lisp editor to create an expression with unbalanced parentheses because such expressions cannot occur in the internal representation of a Lisp object. This editor modifies the structure of existing Lisp objects but does not automatically update any copies of the objects on files. See, for example, the function ''pp'' in chapter 5, for writing functions to files.

This editor is based on the InterLisp editor and has an almost identical command syntax.

16.2. Tutorial

Suppose that we wish to define a function foo which adds five to its argument if it is a number, and returns nil otherwise. We might type the following (incorrect) expression into the interpreter:

```
-> (defun foo (x)      ; incorrect
        ((numberp x) (plus x 5))
        (t nil))
foo
```

Executing foo will cause an error because the conditional function *cond* has been left out. We can correct it by editing the function foo:

```
-> (editf foo)
edit
#
```

We are now in edit mode, with the attention of the editor focused on the expression which defines *foo*. To print the expression on the screen, type:

```
#p
```

```
(lambda (x) (& & ) (t nil))
```

This is not exactly what was typed in. *defun* is really a macro which expands into something involving *def* and *lambda*, so that is why the *lambda* is there. The comment has been omitted and the spacing is different. The reason for these differences is that we are editing a Lisp object, and not the characters which were typed to define the Lisp object. The symbol "&" is just a shorthand for a more complicated subexpression. To see the full expression, type:

```
#?
(lambda (x) ((numberp x) (plus x 5)) (t nil))
```

This is the current expression being edited. To insert cond before the third expression in the current expression, type:

```
#(-3 cond)
(lambda (x) cond (& & ) (t nil))
```

Now we need a pair of parentheses. The editor requires that they be entered as a pair. To insert a left parenthesis before the third element of the current expression and a matching right parenthesis at the end, type:

```
#(li 3)
(lambda (x) (cond &  &  &  & ))
```

The expression appears even more abbreviated as the default print function only shows parenthesis nesting up to a level of two. For the full expression, type:

```
#?
(lambda (x) (cond ((numberp x) (plus x 5)) (t nil)))
```

This definition for *foo* will work, so we can save the change and return to Lisp by typing:

```
#ok
foo
-> (foo 20)
25
-> (foo 'not-a-number)
nil
->
```

Now suppose that we wish to change *foo* so that it adds ten instead of adding five. We reenter the editor:

```
-> (editf foo)
edit
#?
(lambda (x) (cond ((numberp x) (plus x 5)) (t nil)))
```

The current expression only has three elements and "5" is not one of them, so we cannot change "5" directly. Typing "3" causes the editor to focus attention on the third element, and to consider that to be the current expression.

```
#3
(cond ((numberp x) (plus x 5)) (t nil))
#2
((numberp x) (plus x 5))
#2
(plus x 5)
```

The following command replaces the third element with a 10.

```
#(3 10)
(plus x 10)
```

Typing "0" (zero) takes us to a higher level:

```
#0
((numberp x) (plus x 10))
#0
(cond ((numberp x) (plus x 10)) (t nil))
#0
(lambda (x) (cond ((numberp x) (plus x 10)) (t nil)))
```

Suppose that we wish to change *foo* so that it returns "not-a-number" if the argument is not a number. A quick way to find *nil* in the current expression is to type:

```
#f nil
```

The current expression is a valid Lisp object, but it is called the "tail" of an expression because a left parenthesis would be misleading. The following commands replace *nil*, check the result, and exit the editor.

```
#(1 'not-a-number)
#^
(lambda (x) (cond &  &  &  & ))
#?
(lambda (x) (cond ((numberp x) (plus x 10)) (t 'not-a-number)))
#ok
->
```

Variable values and property lists can also be edited. The following example illustrates assigning a value and a property list to a variable, and then using the editor to make modifications.

```
-> (setq foo '(this is a chair))
(this is a chair)
-> (putprop 'foo 'blue 'color)
color
-> foo
(this is a chair)
-> (get 'foo 'color)
blue
-> (editv foo)
edit
#p
(this is a chair)
#(4 pillow)
pillow
#p
(this is a pillow)
#ok
foo
-> (editp foo)
edit
#p
(color blue)
#(2 red)
(color red)
#ok
foo
-> (get 'foo 'color)
red
```

While within the editor, you can reverse the most recent change, type the command *undo*. The command *!undo* undoes all changes made during the editing session.

16.3. Editor Functions

(editf s_x1 ...)

SIDE EFFECT: Edits a function with the name s_x1. Any additional arguments are optional commands to the editor.

RETURNS: s_x1.

NOTE: If s_x1 is not an editable function, editf generates a "fn not editable" error.

(editv s_var [g_com1 ...])

SIDE EFFECT: Edits values in a manner similar to the way editf edits functions. The value of the variable can be changed by subsequent editing commands.

RETURNS: the name of the variable whose value was edited.

(editp s_x)

SIDE EFFECT: Edits property lists.

RETURNS: the atom whose property list was edited.

(editfns s_x [g_coms1 ...])

SIDE EFFECT: Performs the same editing operations on several functions. The symbol s_x is the function or list of functions, and the following arguments are the editing commands. Evaluation of editfns will map down the list of functions, print the name of each function, and call the editor (via editf) on each function.

RETURNS: nil.

EXAMPLE: (editfns foofns (r fie fum)) will change every fie to fum in each of the functions in the list called foofns.

NOTE: The call to the editor is errset protected, so that if the editing of one function causes an error, editfns will proceed to the next function. In the above example, if one of the functions did not contain a fie, the r command would cause an error, but editing would continue with the next function.

(editracefn s_com)

NOTE: This is available to help the user debug complex edit macros, or subroutine calls to the editor. It is initially an undefined function, to be defined by the user. Whenever the value of editracefn is non-nil, the editor calls the function editracefn before executing each command (at any level), giving it that command as its argument.

(editfindp x pat nil)

NOTE: Allows a program to use the editor find command as a pure predicate from outside the editor. It searches for the pattern *pat* in the expression *x*.

RETURNS: t if the editor command *f pat* would succeed, *nil* otherwise.

16.3.1. The Edit Chain

The edit-chain is a list of which the first element is the expression you are now editing ("current expression"), the next element is what would become the current expression if you were to type a 0, etc., until the last element which is the expression that was passed to the editor.

EDIT CHAIN COMMAND SUMMARY

mark. Adds the current edit chain to the front of the list marklst.

_ . Makes the new edit chain be (car marklst).

(_ pattern). Ascends the edit chain looking for a link which matches pattern.

__ . A double underscore is similar to a single underscore (_) but also erases the mark.

/. Makes the edit chain be the value of unfind. Unfind is set to the current edit chain by each command that makes a "big jump", i.e., a command that usually performs more than a single ascent or descent, namely ^, _, __, !nx, all commands that involve a search, e.g., f, lc, !!, below, et al and / and /p themselves. If the user types f cond, and then f car, / would take him back to the cond. Another / would take him back to the car, etc.

/p. Restores the edit chain to its state as of the last print operation. If the edit chain has not changed since the last printing, /p restores it to its state as of the printing before that one. If the user types p followed by 3 2 1 p, /p will return to the first p, i.e., would be equivalent to 0 0 0. Another /p would then take him back to the second p.

(\# g_com1 ...)

RETURNS: what the current expression would be after executing the edit commands com1 ... starting from the present edit chain, generating an error if any of comi cause errors. The current edit chain is never changed.

EXAMPLE: (i r (quote x) (\# (cons ..z))) replaces all x's in the current expression by the first cons containing a z.

16.4. Printing Commands

PRINTING COMMAND SUMMARY

p Prints current expression in abbreviated form. *(p m)* prints mth element of current expression in abbreviated form. *(p m n)* prints mth element of current expression as though printlev were given a depth of n. *(p 0 n)* prints the current expression as though printlev were given a depth of *n*. *(p foo)* will search for the first occurrence of *foo* and then print it.

? . prints the current expression as though printlev were given a depth of 100.

pp . pretty-prints the current expression.

*pp**. is like pp, but forces comments to be shown.

16.5. Scope of Attention

Attention-changing commands allow you to look at a different part of a Lisp expression you are editing. The sub-structure upon which the editor's attention is centered is called "the current expression". Changing the current expression means shifting attention and not actually modifying any structure.

SCOPE OF ATTENTION COMMAND SUMMARY

n (n> 0) . Makes the nth element of the current expression be the new current expression.

-n (n> 0). Makes the nth element from the end of the current expression be the new current expression.

0. Makes the next higher expression be the new correct expression. If the intention is to go back to the next higher left parenthesis, use the command !0.

up . Unless the current expression is a tail, *up* changes the current expression to the one which has the previous current expression as its first element. Tails are unchanged. (A tail is an expression which starts with "..." when printed with the *p* command.)

!0. Goes back to the next higher left parenthesis.

^ . Makes the top level expression be the current expression.

nx. Makes the current expression be the next expression. It will not go through an unmatched right parenthesis, so it generates an error if the current expression is the last

(nx n) n> 0 equivalent to n consecutive *nx* commands.

!nx . Makes current expression be the next expression at a higher level. Goes through any number of right parentheses to get to the next expression. It always gives a different result from *nx.*

bk. Makes the current expression be the previous expression in the next higher expression.

(nth n) n> 0. Makes the list starting with the nth element of the current expression be the current expression.

(nth $) . This generalized *nth* command locates $, and then backs up to the current level, where the new current expression is the tail whose first element contains, however deeply, the expression that was the terminus of the location operation.

!!, as in *(pattern !! . $)*. Searches for an expression or tail which starts with *pattern* and ends with *$.* For example, *(cond !! return)* finds a *cond* that contains a *return,* at any depth.

(below com x) . This ascends to higher levels searching for *com* and then changes the current expression to the one which is *x* levels below *com.* The default value of *x* is 1. For example *(below cond)* will cause the *cond* clause containing the current expression to become the new current expression.

(nex x). same as *(below x)* followed by nx. For example, if you are deep inside of a selectq clause, you can advance to the next clause with *(nex selectq).*

nex. The atomic form of *nex* is useful if you will be performing repeated executions of *(nex x)*. By simply marking the chain corresponding to x, you can use *nex* to step through the sublists.

16.6. Pattern and Search Commands

In many of the editor commands it is possible to specify a pattern to direct an operation to a subexpression or change the attention of the editor. This section describes the types of patterns and searches.

PATTERN SPECIFICATION SUMMARY

A pattern *pat* matches with x if:

- *pat* is *eq* to x. In this case, x may not be a tail, so (a b) will not match ... a b).

- x is a list, (car *pat*) matches (car x), and (cdr *pat*) matches (cdr x).

- *pat* is &.

- *pat* is a number and equal to x.

- (car *pat*) is the atom **any**, (cdr *pat*) is a list of patterns, and one of those patterns matches x.

- *pat* is a literal atom or string, and (nthchar *pat*-1) is @, then *pat* matches with any literal atom or string which has the same initial characters as *pat*, e.g. ver@ matches with verylongatom, as well as "verylongstring".

- if (car *pat*) is the atom --, *pat* matches x if (a) (cdr *pat*)=nil, i.e. *pat*=(--), e.g., (a --) matches (a) (a b c) and (a . b) in other words, -- can match any tail of a list. (b) (cdr *pat*) matches with some tail of x, e.g. (a -- (&)) will match with (a b c (d)), but not (a b c d), or (a b c (d) e). however, note that (a -- (&) --) will match with (a b c (d) e). in other words, -- will match any interior segment of a list.

- if (car *pat*) is the atom ==, *pat* matches x if and only if (cdr *pat*) is *eq* to x. (This pattern is for use by programs that call the editor as a subroutine, since any non-atomic expression in a command typed in by the user obviously cannot be *eq* to existing structure.)

- *pat* has !!! for its car, and either its cdr matches with x or x is a tail which would match if it had a left parenthesis. For example, searching for a match with (!!! b c) will succeed on (a (b c)) as well as on (a b c).

SEARCH COMMAND SUMMARY

f pattern. Finds the next instance of *pattern*. If no pattern is given then the last pattern is used.

(f pattern n). Finds the next instance of *pattern*. (Here, n stands for *next*, and not an integer.)

(f pattern t). Similar to f pattern, except, for example, if the current expression is (cond ..), f cond will look for the next cond, but (f cond t) will not.

(f pattern n) n> 0. Finds the nth place that pattern matches. If the current expression is (fool foo2 foo3), (f foo@ 3) will find foo3.

(f pattern) or (f pattern nil). only matches with elements at the top level of the current expression. If the current expression is *(prog nil (setq x (cond & &)) (cond &) ...)* f (cond --) will find the cond inside the setq, whereas (f (cond --)) will find the top level cond, i.e., the second one.

(fs pattern1 ... patternn) . Is equivalent to *f pattern1* followed by *f pattern2* ... followed by *f patternn*, so that if a search fails, the edit chain is left at the place where the previous pattern matched.

(f= expression x) . Searches for a structure *eq* to *expression.*

(orf pattern1 ... patternn). Searches for an expression that is matched by either *pattern1* or ... *patternn.*

bf pattern . This backwards find searches for the first previous occurrence of the pattern. If the current expression is the top-level expression, then the entire expression is searched in reverse print order. For example, if the current expression is *(prog nil (setq x (setq y (list z))) (print x))* , then *f list* followed by *bf setq* will change the current expression to *(setq y (list z))*, as will *f print* followed by *bf setq.*

(bf pattern t). This is similar to the above backwards find. Search always includes current expression, i.e., starts at end of current expression and works backward, then ascends and backs up, etc.

16.7. Location Specifications

Many editor commands use a method of specifying position called a location specification. The meta-symbol $ is used to denote a location specification. $ is a list of commands interpreted as described above. $ can also be atomic, in which case it is interpreted as *(list $).* A location specification is a list of edit commands that are executed in the normal fashion with the following exception. All commands not recognized by the editor are interpreted as though they had been preceded by *f.* The location specification *(cond 2 3)* specifies the third element in the first clause of the next cond.

The *if* command and the \# function provide a way of using in location specifications arbitrary predicates applied to elements in the current expression.

LOCATION COMMAND SUMMARY

$ In descriptions of the editor, the meta-symbol $ is used to denote a location specification. $ is a list of commands interpreted as described above. $ can also be atomic.

(lc . $) . Provides a way of explicitly invoking the location operation. (lc cond 2 3) will perform a search for a cond clause and then change the current expression to the third element of the cond clause.

(lcl . $) . Same as lc except search is confined to current expression. To find a cond containing a return, one might use the location specification (cond (lcl return) /) where the / would reverse the effects of the lcl command, and make the final current expression be the cond.

(second . $) . same as (lc . $) followed by another (lc . $) except that if the first succeeds and second fails, no change is made to the edit chain.

(third . $) . Similar to second.

16.8. Structure Modification Commands

All structure modification commands are undoable. See section 16.11 for a description of undoing commands.

In insert, delete, replace and change, if $ is nil (empty), the corresponding operation is performed on the current edit chain, i.e. (replace with (car x)) is equivalent to (! (car x)). For added readability, here is also permitted, e.g., (insert (print x) before here) will insert (print x) before the current expression (but not change the edit chain). It is perfectly legal to ascend to insert, replace, or delete. For example (insert (*return*) after ^ prog -1) will go to the top, find the first prog, and insert a (*return*) at its end, and not change the current edit chain.

The a, b, and ! commands all make special checks in e1 thru em for expressions of the form (\# . coms). In this case, the expression used for inserting or replacing is a copy of the current expression after executing coms, a list of edit commands. (insert (\# f cond -1 -1) after3) will make a copy of the last form in the last clause of the next cond, and insert it after the third element of the current expression.

STRUCTURE MODIFICATION COMMAND SUMMARY

(n) n>1 deletes the corresponding element from the current expression.

(n e1 ... em) n,m>1, replaces the nth element in the current expression with e1 ... em.

(-n e1 ... em) n,m>1 inserts e1 ... em before the n element in the current expression.

(n e1 ... em) (the letter "n" for "next" or "nconc", not a number) m>1 attaches e1 ... em at the end of the current expression.

(a e1 ... em). inserts e1 ... em after the current expression (or after its first element if it is a tail).

(b e1 ... em). inserts e1 ... em before the current expression. To insert foo before the last element in the current expression, perform -1 and then (b foo).

(! e1 ... em). replaces the current expression by e1 ... em. If the current expression is a tail then replace its first element.

(r x y) replaces each occurrence of *x* with *y* in the current expression. The term *x* can be an atom, a list, or a location specification.

(sw n m) switches the nth and mth elements of the current expression. For example, if the current expression is (list (cons (car x) (car y)) (cons (cdr y))), (sw 2 3) will modify it to be (list (cons (cdr x) (cdr y)) (cons (car x) (car y))). (sw car cdr) would produce the same result.

delete or (!) . deletes the current expression, or if the current expression is a tail, deletes its first element.

(delete . $). does a (lc . $) followed by delete. current edit chain is not changed.

(insert e1 ... em before . $). similar to (lc. $) followed by (b e1 ... em).

(insert e1 ... em after . $). similar to insert before except uses a instead of b.

(insert e1 ... em for . $). similar to insert before except uses ! for b.

(replace $ with e1 ... em). here $ is the segment of the command between replace and with.

(change $ to e1 ... em). same as replace with.

EXTRACTION AND EMBEDDING COMMAND SUMMARY

(xtr . $) . Replaces the original current expression with the expression that is current after performing (lcl . $).

(mbd x) . If x is a list, substitutes the current expression for all instances of the atom * in x, and replaces the current expression with the result of that substitution. If x is atomic, (mbd x) is the same as (mbd (x *)).

(extract $1 from $2) . This is an editor command which replaces the current expression with one of its subexpressions (from any depth). ($1 is the segment between extract and from.) For example, if the current expression is (print (cond ((null x) y) (t z))) then following (extract y from cond), the current expression will be (print y). (extract 2 -1 from cond), (extract y from 2), (extract 2 -1 from 2) will all produce the same result.

(embed $ in . x) . Replaces the current expression with a new expression which contains it as a subexpression. ($ is the segment between embed and in.) Some examples: (embed print in setq x), (embed 3 2 in *return*), (embed cond 3 1 in (or * (null x))).

MOVE AND COPY COMMAND SUMMARY

(move $1 to com . $2) . ($1 is the segment between move and to.) where com is before, after, or the name of a list command, e.g., :, n, etc. If $2 is nil, or (here), the current position specifies where the operation is to take place. If $1 is nil, the move command allows the user to specify some place the current expression is to be moved to. If the current expression is (a b d c), (move 2 to after 4) will make the new current expression be (a c d b).

(mv com . $) . is the same as (move here to com . $).

(copy $1 to com . $2) is like move except that the source expression is not deleted.

(cp com . $). is like mv except that the source expression is not deleted.

16.9. Parentheses Moving Commands

The commands presented in this section permit modification of the list structure itself, as opposed to modifying components. Their effect can be described as inserting or removing a single left or right parenthesis, or pair of left and right parentheses. Some people find that use of only 'bi' and 'bo' to be less confusing and quite adequate for use instead of the 4 additional commands.

PARENTHESES MOVING COMMAND SUMMARY

(bi n m) . This "both in" command inserts parentheses before the nth element and after the mth element in the current expression. example: if the current expression is (a b (c d e) f g), then (bi 2 4) will modify it to be (a (b (c d e) f) g). (bi n) : same as (bi n n). example: if the current expression is (a b (c d e) f g), then (bi -2) will modify it to be (a b (c d e) (f) g).

(bo n). This "both out" command removes both parentheses from the nth element. example: if the current expression is (a b (c d e) f g), then (bo d) will modify it to be (a b c d e f g).

(li n) . This "left in" command inserts a left parenthesis before the nth element (and a matching right parenthesis at

the end of the current expression). example: if the current expression is (a b (c d e) f g), then (li 2) will modify it to be (a (b (c d e) f g)).

(lo n) . This "left out" command removes a left parenthesis from the nth element. all elements following the nth element are deleted. example: if the current expression is (a b (c d e) f g), then (lo 3) will modify it to be (a b c d e).

(ri n m) . This "right in" command moves the right parenthesis at the end of the nth element in to after the mth element. inserts a right parenthesis after the mth element of the nth element. The rest of the nth element is brought up to the level of the current expression. example: if the current expression is (a (b c d e) f g), (ri 2 2) will modify it to be (a (b c) d e f g).

(ro n) . This "right out" command moves the right parenthesis at the end of the nth element out to the end of the current expression. removes the right parenthesis from the nth element, moving it to the end of the current expression. all elements following the nth element are moved inside of the nth element. example: if the current expression is (a b (c d e) f g), (ro 3) will modify it to be (a b (c d e f g)).

Certain commands can be made to made to operate on several contiguous elements of a list by using the to or thru command in their respective location specifications. These commands are *to, thru, extract, embed, delete, replace,* and *move.* to and thru can also be used directly with *xtr* (which takes after a location specification), as in *(xtr (2 thru 4))* (from the current expression).

TO AND THRU COMMAND SUMMARY

($1 to $2) . same as thru except last element not included.

($1 to). same as ($1 thru -1)

($1 thru $2) . If the current expression is (a (b (c d) (e) (f g h) i) j k), following (c thru g), the current expression will be ((c d) (e) (f g h)). If both $1 and $2 are numbers, and $2 is greater than $1, then $2 counts from the beginning of the current expression, the same as $1. in other words, if the current expression is (a b c d e f g), (3 thru 4) means (c thru d), not (c thru f). in this case, the corresponding bi command is (bi 1 $2-$1+1).

($1 thru). same as *($1 thru -1).*

16.10. Undoing Commands

Each command that causes structure modification automatically adds an entry to the front of a list called *undolst.* The undo command undoes the most recent such command based on information in *undolst.*

UNDO COMMAND SUMMARY

undo . the undo command undoes most recent, structure modification command that has not yet been undone, and prints the name of that command, e.g., mbd undone. The edit chain is then exactly what it was before the 'undone' command had been performed.

!undo . undoes all modifications performed during this editing session, i.e., this call to the editor.

unblock . removes an undo-block. If executed at a non-blocked state, i.e., if undo or !undo could operate, types not blocked.

test adds an undo-block at the front of *undolst.* Note that By using *test* together with *!undo,* the user can perform a number of changes, and then undo all of them with a single *!undo* command.

?? prints the entries on undolst. The entries are listed most recent entry first.

16.11. Commands that Evaluate

These commands allow you to execute arbitrary Lisp expressions, perhaps including calling a function you are editing! All the changes you have made are "in place" in the interpreted version of the function under edit.

EVALUATION COMMAND SUMMARY

e. when typed in as a single atomic command, passes the next s-expression to the Lisp reader and evaluates and prints it. Other uses of the symbol 'e' are unaffected: (i.e., (insert d before e) will treat e as a pattern) *(e x)* evaluates x and prints the result. (e x t) is the same as (e x) but does not print.

(i c x1 ... xn) same as (c y1 ... yn) where yi=(eval xi). example: (i 3 (cdr foo)) will replace the 3rd element of the current expression with the cdr of the value of foo. (i n foo (car fie)) will attach the value of foo and car of the value of fie to the end of the current expression. (i f= foo t) will search for an expression eq to the value of foo. If c is not an atom, it is evaluated as well. (The coms and comsq commands below provide more general ways of computing commands.)

(coms x1 ... xn). Each xi is evaluated and its value executed as a command. For example, (coms (cond (x (list 1 x)))) will replace the first element of the current expression with the value of x if non-nil, otherwise do nothing. (Note that nil as a command does nothing.)

(comsq com1 ... comn). Executes com1 ... comn and used mainly useful in conjunction with the coms command. For example, suppose the user wishes to compute an entire list of commands for evaluation, as opposed to computing each command one at a time as does the coms command. He would then write (coms (cons 'comsq x)) where x computed the list of commands, e.g., (coms (cons 'comsq (get foo 'commands)))

16.12. Commands that Test

TESTING COMMAND SUMMARY

(if x) Generates an error unless the value of (eval x) is non-nil. Thus an error is generated if either (eval x) causes an error or if (eval x) is nil.

(if x coms1) Evaluates x and if it is non-nil, executes coms1. Otherwise, generates an error.

(if x coms1 coms2) Evaluates x and if it is non-nil, executes coms1. If (eval x) causes an error or is equal to nil, coms2 is executed.

(lp . coms). repeatedly executes coms, a list of commands, until an error occurs. *(lp f print (n t))* will attach a t at the end of every print expression. (lp f print (if (\# 3) nil ((n t)))) will attach a t at the end of each print expression which does not already have a second argument. (i.e. the form (\# 3) will cause an error if the edit command 3 causes an error, thereby selecting ((n t)) as the list of commands to be executed. The if could also be written as (if

(cddr (\#)) nil ((n t))).).

(lpq . coms) same as lp but does not print n occurrences.

(orr coms1 ... comsn) . orr begins by executing coms1, a list of commands. If no error occurs, orr is finished. otherwise, orr restores the edit chain to its original value, and continues by executing coms2, etc. If none of the command lists execute without errors, i.e., the orr "drops off the end", orr generates an error. Otherwise, the edit chain is left as of the completion of the first command list which executes without error.

16.13. Editor Macros

Many of the more sophisticated branching commands in the editor, such as orr, if, etc., are most often used in conjunction with edit macros. The macro feature permits the user to define new commands and thereby expand the editor's repertoire. (However, built in commands always take precedence over macros, i.e., the editor's repertoire can be expanded, but not modified.) Macros are defined by using the m command. If a macro is redefined, its new definition replaces its old.

(m c . coms) defines c as an atomic command, where c is an atom and coms is a list. Executing c is then the same as executing the list of commands coms. see the next paragraph for an example. Macros can also define list commands, i.e., commands that take arguments. (m (c) (arg[1] ... arg[n]) . coms) c an atom. m defines c as a list command. Executing (c e1 ... en) is then performed by substituting e1 for arg[1], ... en for arg[n] throughout coms, and then executing coms. a list command can be defined via a macro so as to take a fixed or indefinite number of 'arguments'. The form given above specified a macro with a fixed number of arguments, as indicated by its argument list. If the of arguments. (m (c) args . coms) c, args both atoms, defines c as a list command. executing (c e1 ... en) is performed by substituting (e1 ... en), i.e., cdr of the command, for args throughout coms, and then executing coms.

(m bp bk up p) will define bp as an atomic command which does three things, a bk, an up, and a p. note that macros can use commands defined by macros as well as built in commands in their definitions. For example, suppose z is defined by (m z -1 (if (null (\#)) nil (p))), i.e. z does a -1, and then if the current expression is not nil, a p. now we can define zz by (m zz -1 z), and zzz by (m zzz -1 -1 z) or (m zzz -1 zz). We could define a more general bp by (m (bp) (n) (bk n) up p). (bp 3) would perform (bk 3), followed by an up, followed by a p. The command second can be defined as a macro by (m (2nd) x (orr ((lc . x) (lc . x)))).

Note that for all editor commands, 'built in' commands as well as commands defined by macros, atomic definitions and list definitions are completely independent. In other words, the existence of an atomic definition for c in no way affects the treatment of c when it appears as car of a list command, and the existence of a list definition for c in no way affects the treatment of c when it appears as an atom. In particular, c can be used as the name of either an atomic command, or a list command, or both. In the latter case, two entirely different definitions can be used. Note also that once c is defined as an atomic command via a macro definition, it will not be searched for when used in a location specification, unless c is preceded by an f. (insert -- before bp) would not search for bp, but instead perform a bk, an up, and a p, and then do the insertion. The corresponding also holds true for list commands.

(bind . coms) This is an edit command which is useful mainly in macros. It binds three dummy variables #1, #2, #3, (initialized to nil), and then executes the edit commands coms. Note that these bindings are only in effect while the commands are being executed, and that bind can be used recursively; it will rebind #1, #2, and #3 each time it is invoked.

usermacros is a Lisp variable which contains a list of the user-defined editing macros with their definitions. These macros remain in effect from one editing session to another. you can save your macros for another Lisp session by saving usermacros on a disk file.

editcomsl is a Lisp variable which contains a list of the "list commands" recognized by the editor. (These are the commands such as *li* whose execution takes the form (command arg1 arg2 ...).)

16.14. Miscellaneous Editor Commands

This section contains a descriptions of those editing functions which can be called from the lisp top level. These include functions for merely entering the editor as well as some which perform some editing tasks and return to the top level.

MISCELLANEOUS EDITOR COMMAND SUMMARY

ok. Exits from the editor.

nil. Unless preceded by f or bf, is always a null operation.

tty . Calls the editor recursively. The user can then type in commands, and have them executed. The tty command is completed when the user exits from the lower editor (with ok or stop). The tty command is extremely useful. It enables the user to set up a complex operation, and perform interactive attention-changing commands part way through it. For example the command (move 3 to after cond 3 p tty) allows the user to interact, in effect, within the move command. He can verify for himself that the correct location has been found, or complete the specification "by hand". In effect, tty says "I'll tell you what you should do when you get there."

stop . Exits from the editor with an error. This is mainly for use in conjunction with tty commands that the user wants to abort. Since all of the commands in the editor are errset protected, the user must exit from the editor via a command. The stop command provides a way of distinguishing between a successful and unsuccessful (from the user's standpoint) editing session.

tl. Calls (top-level). To return to the editor just use the *return* top-level command.

repack. Permits the 'editing' of an atom or string.

(repack $) Does (lc . $) followed by repack, e.g. (repack this@).

(makefn form args n m) n,m > 0. Makes (car form) an expr with the nth through mth elements of the current expression with each occurrence of an element of (cdr form) replaced by the corresponding element of args. The nth through mth elements are replaced by form.

(makefn form args n). Same as (makefn form args n n).

(s var) . Sets var (using setq) to the current expression. If the current expression is a tail, the appropriate left parenthesis is generated.

(s var . $). Performs the location command (lc . $) and then sets var to the new current expression. For example, (s foo -1 1) will set foo to the first element in the last element of the current expression.

APPENDIX A

Index to FRANZ LISP Functions

(\# g_com1 ...) 16-5
(*array 's_name 's_type 'x_dim1 ... 'x_dim*n*) 2-17
(*break 'g_pred 'g_message) 4-2
(*catch 'ls_tag g_exp) 4-3
(*makhunk 'x_arg) 2-20
(*mod 'x_dividend 'x_divisor) 3-7
(*process 'st_command ['g_readp ['g_writep]]) 6-4
(*process-receive 'st_command) 6-5
(*process-send 'st_command) 6-5
(*quo 'i_x 'i_y) 3-2
(*rplacx 'x_ind 'h_hunk 'g_val) 2-20
(*rset 'g_flag) 6-6
(*throw 's_tag 'g_val) 4-16
(**/** ['x_arg1 ...]) 3-2
(**1+** 'x_arg) 3-1
(**1−** 'x_arg) 3-2
(**<** 'fx_arg1 'fx_arg2) 3-3
(**<&** 'x_arg1 'x_arg2) 3-3
(**>** 'fx_arg1 'fx_arg2) 3-3
(**>&** 'x_arg1 'x_arg2) 3-3
(**Divide** 'i_dividend 'i_divisor) 3-2
(**Emuldiv** 'x_fact1 'x_fact2 'x_addn 'x_divisor) 3-2
(**I-throw-err** 'l_token) 4-12
(* ['x_arg1 ...]) 3-2
(**=** 'fx_arg1 'fx_arg2) 3-4
(**=&** 'x_arg1 'x_arg2) 3-4
(**−** ['x_arg1 ...]) 3-1
(**+** ['x_arg1 ...]) 3-1
(**abs** 'n_arg) 3-6
(**absval** 'n_arg) 3-6
(**acos** 'fx_arg) 3-4
(**add** ['n_arg1 ...]) 3-1
(**add-syntax-class** 's_synclass 'l_properties) 7-10
(**add1** 'n_arg) 3-1
(**addhash** 'g_key 'H_htab 'g_val) 14-3
(**aexplode** 's_arg) 2-14
(**aexplodec** 's_arg) 2-14
(**aexploden** 's_arg) 2-14
(**allocate** 's_type 'x_pages) 6-1
(**alphalessp** 'st_arg1 'st_arg2) 2-12
(**and** [g_arg1 ...]) 4-1
(**append** 'l_arg1 'l_arg2) 2-1
(**append1** 'l_arg1 'g_arg2) 2-2
(**apply** 'u_func 'l_args) 4-1
(**arg** ['x_numb]) 4-2
(**argv** 'x_argnumb) 6-1
(**array** s_name s_type x_dim1 ... x_dim*n*) 2-17

(**arraycall** s_type 'as_array 'x_indl ...) 2-18
(**arraydims** 's_name) 2-18
(**arrayp** 'g_arg) 2-17
(**arrayp** 'g_arg) 2-8
(**arrayref** 'a_name 'x_ind) 2-18
(**ascii** 'x_charnum) 2-11
(**asin** 'fx_arg) 3-4
(**assoc** 'g_arg1 'l_arg2) 2-21
(**assq** 'g_arg1 'l_arg2) 2-21
(**atan** 'fx_arg1 'fx_arg2) 3-4
(**atom** 'g_arg) 2-8
(**attach** 'g_x 'l_l) 2-5
(**baktrace**) 6-1
(**bcdad** 's_funcname) 2-25
(**bcdp** 'g_arg) 2-8
(**bignum-leftshift** bx_arg x_amount) 3-5
(**bignum-to-list** 'b_arg) 2-3
(**bigp** 'g_arg) 2-8
(**boole** 'x_key 'x_v1 'x_v2 ...) 3-5
(**boundp** 's_name) 2-12
(**break** [g_message ['g_pred]]) 4-2
(**c..r** 'lh_arg) 2-4
(**car** 'l_arg) 2-4
(**caseq** 'g_key-form l_clause1 ...) 4-2
(**catch** g_exp [ls_tag]) 4-3
(**cdr** 'l_arg) 2-4
(**cfasl** 'st_file 'st_entry 'st_funcname ['st_disc ['st_library]]) 5-1
(**chdir** 's_path) 6-1
(**close** 'p_port) 5-2
(**clrhash** 'H_htab) 14-3
(**command-line-args**) 6-2
(**comment** [g_arg ...]) 4-4
(**concat** ['stn_arg1 ...]) 2-11
(**concatl** 'l_arg) 2-11
(**cond** [l_clause1 ...]) 4-4
(**cons** 'g_arg1 'g_arg2) 2-1
(**copy** 'g_arg) 2-25
(**copyint*** 'x_arg) 2-25
(**copysymbol** 's_arg 'g_pred) 2-11
(**cos** 'fx_angle) 3-4
(**cprintf** 'st_format 'xfst_val ['p_port]) 5-2
(**cpy1** 'xvt_arg) 2-25
(**cvttofranzlisp**) 4-4
(**cvttointlisp**) 4-4
(**cvttomaclisp**) 4-4
(**cvttoucilisp**) 4-4
(**cxr** 'x_ind 'h_hunk) 2-20
(**debug** [s_msg]) 15-5
(**debug** s_msg) 4-4
(**debugging** 'g_arg) 4-5
(**declare** [g_arg ...]) 4-5
(**def** s_name (s_type l_argl g_exp1 ...)) 4-5
(**defcmacro** s_name l_arg g_exp1 ...) 4-5
(**defmacro** s_name l_arg g_exp1 ...) 4-5
(**defprop** ls_name g_val g_ind) 2-23

(defun s_name [s_mtype] ls_argl g_expl ...) 4-5
(defvar s_variable ['g_init]) 4-6
(delete 'g_val 'l_list ['x_count]) 2-5
(delq 'g_val 'l_list ['x_count]) 2-5
(deref 'x_addr) 6-2
(desetq sl_pattern1 'g_exp1 [... ...]) 2-14
(diff ['n_arg1 ...]) 3-1
(difference ['n_arg1 ...]) 3-1
(do l_vrbs l_test g_exp1 ...) 14-6
(do l_vrbs l_test g_exp1 ...) 4-6
(do s_name g_init g_repeat g_test g_exp1 ...) 4-7
(do* l_vrbs l_test g_exp1 ...) 14-6
(dolist (s_var l_form g_resultform) g_form) 14-6
(dotimes (s_var i_countform g_resultform) g_form) 14-6
(drain ['p_port]) 5-2
(dremove 'g_val 'l_list ['x_count]) 2-5
(dsubst 'g_x 'g_y 'l_s) 2-6
(dtpr 'g_arg) 2-3
(dtpr 'g_arg) 2-8
(dumplisp s_name) 6-2
(editf s_x1 ...) 16-4
(editfindp x pat nil) 16-5
(editfns s_x [g_coms1 ...]) 16-4
(editp s_x) 16-4
(editracefn s_com) 16-4
(editv s_var [g_com1 ...]) 16-4
(environment [l_when1 l_what1 l_when2 l_what2 ...]) 4-8
(environment-lmlisp [l_when1 l_what1 l_when2 l_what2 ...]) 4-8
(environment-maclisp [l_when1 l_what1 l_when2 l_what2 ...]) 4-8
(eq 'g_arg1 'g_arg2) 2-9
(eqstr 'g_arg1 'g_arg2) 2-10
(equal 'g_arg1 'g_arg2) 2-10
(err ['s_value [nil]]) 4-8
(error ['s_message1 ['s_message2]]) 4-8
(errset g_expr [s_flag]) 4-8
(eval 'g_val ['x_bind-pointer]) 4-9
(eval-when l_time g_exp1 ...) 6-2
(evalframe 'x_pdlpointer) 4-9
(evalhook 'g_form 'su_evalfunc ['su_funcallfunc]) 4-10
(evenp 'x_arg) 3-3
(exec s_arg1 ...) 4-10
(exece 's_fname ['l_args ['l_envir]]) 4-10
(exit ['x_code]) 6-2
(exp 'fx_arg) 3-6
(explode 'g_arg) 2-14
(explodec 'g_arg) 2-14
(exploden 'g_arg) 2-14
(expt 'n_base 'n_power) 3-6
(fact 'x_arg) 3-6
(fake 'x_addr) 6-2
(fasl 'st_name ['st_mapf ['g_warn]]) 5-2
(fclosure 'l_vars 'g_funobj) 2-25
(fclosure-alist 'v_fclosure) 2-25
(fclosure-function 'v_fclosure) 2-25
(fclosure-list l_list u_function [...]) 8-8

(**fclosurep** 'v_fclosure)....2-25
(**filepos** 'p_port ['x_pos])....5-3
(**filestat** 'st_filename)....5-3
(**fillarray** 's_array 'l_itms)....2-19
(**fix** 'n_arg)....3-6
(**fixp** 'g_arg)....3-3
(**flatc** 'g_form ['x_max])....5-3
(**flatsize** 'g_form ['x_max])....5-3
(**float** 'n_arg)....3-6
(**floatp** 'g_arg)....3-3
(**fork**)....6-2
(**freturn** 'x_pdl-pointer 'g_retval)....4-10
(**frexp** 'f_arg)....4-10
(**fseek** 'p_port 'x_offset 'x_flag)....5-3
(**funcall** 'u_func ['g_arg1 ...])....4-11
(**funcallhook** 'l_form 'su_funcallfunc ['su_evalfunc])....4-11
(**function** u_func)....4-11
(**gc**)....6-3
(**gcafter** s_type)....6-3
(**gensym** ['s_leader])....2-11
(**get** 'ls_name 'g_ind)....2-23
(**get_pname** 's_arg)....2-13
(**getaccess** 'a_array)....2-17
(**getaddress** 's_entry1 's_binder1 'st_discipline1 [... ... ...])....2-26
(**getaux** 'a_array)....2-17
(**getchar** 's_arg 'x_index)....2-13
(**getcharn** 's_arg 'x_index)....2-13
(**getd** 's_arg)....2-13
(**getdata** 'a_array)....2-17
(**getdelta** 'a_array)....2-17
(**getdisc** 'y_bcd)....2-21
(**getdisc** 'y_func)....4-11
(**getentry** 'y_bcd)....2-21
(**gethash** 'g_key 'H_htab ['g_defval])....14-2
(**getl** 'ls_name 'l_indicators)....2-23
(**getlength** 'a_array)....2-17
(**getsyntax** 's_symbol)....7-10
(**go** g_labexp)....4-11
(**greaterp** ['n_arg1 ...])....3-3
(**haipart** bx_number x_bits)....3-4
(**hash-table-count** 'H_htab)....14-3
(**hash-table-p** 'H_arg)....14-2
(**hashtabstat**)....6-3
(**haulong** bx_number)....3-4
(**hunk** 'g_val1 ['g_val2 ... 'g_val*n*])....2-19
(**hunk-to-list** 'h_hunk)....2-20
(**hunkp** 'g_arg)....2-8
(**hunksize** 'h_arg)....2-20
(**if** 'g_a 'g_b 'g_c ...)....4-12
(**if** 'g_a 'g_b)....4-12
(**if** 'g_a **then** 'g_b [...] [**elseif** 'g_c **then** 'g_d ...] [**else** 'g_e [...])....4-12
(**if** 'g_a **then** 'g_b [...] [**elseif** 'g_c **thenret**] [**else** 'g_d [...])....4-12
(**implode** 'l_arg)....2-11
(**include** s_filename)....6-3
(**include-if** 'g_predicate s_filename)....6-4

(includef 's_filename)......6-4
(includef-if 'g_predicate s_filename)......6-4
(infile 's_filename)......5-3
(insert 'g_object 'l_list 'u_comparefn 'g_nodups)......2-6
(intern 's_arg)......2-12
(kwote 'g_arg)......2-26
(last 'l_arg)......2-4
(lconc 'l_ptr 'l_x)......2-24
(ldiff 'l_x 'l_y)......2-5
(length 'l_arg)......2-3
(lessp ['n_arg1 ...])......3-3
(let l_args g_exp1 ... g_exprn)......4-13
(let* l_args g_exp1 ... g_expn)......4-13
(lexpr-funcall 'g_function ['g_arg1 ...] 'l_argn)......4-13
(list ['g_arg1 ...])......2-1
(list-to-bignum 'l_ints)......2-3
(listarray 'sa_array ['x_elements])......2-18
(listify 'x_count)......4-13
(listp 'g_arg)......2-3
(listp 'g_arg)......2-8
(load 's_filename ['st_map ['g_warn]])......5-4
(log 'fx_arg)......3-6
(lsh 'x_val 'x_amt)......3-5
(lsubst 'l_x 'g_y 'l_s)......2-7
(macroexpand 'g_form)......2-26
(make-hash-table :size :test :rehash-size :rehash-threshold)......14-2
(make-vector-float 'x_size)......14-7
(makeht 'x_size ['s_test])......14-2
(makereadtable ['s_flag])......5-4
(makhunk 'xl_arg)......2-20
(maknam 'l_arg)......2-11
(maknum 'g_arg)......6-4
(makunbound 's_arg)......2-14
(map 'u_func 'l_arg1 ...)......4-13
(map-over-oblist 'u_fun)......14-6
(mapc 'u_func 'l_arg1 ...)......4-13
(mapcan 'u_func 'l_arg1 ...)......4-14
(mapcar 'u_func 'l_arg1 ...)......4-14
(mapcon 'u_func 'l_arg1 ...)......4-14
(maphash 'u_fun 'H_htab)......14-3
(maplist 'u_func 'l_arg1 ...)......4-14
(marray 'g_data 's_access 'g_aux 'x_length 'x_delta)......2-17
(max 'n_arg1 ...)......3-6
(member 'g_arg1 'l_arg2)......2-10
(memq 'g_arg1 'l_arg2)......2-10
(merge 'l_data1 'l_data2 'u_comparefn)......2-6
(mfunction t_entry 's_disc)......4-14
(min 'n_arg1 ...)......3-7
(minus 'n_arg)......3-2
(minusp 'g_arg)......3-3
(mod 'i_dividend 'i_divisor)......3-7
(msg [l_option ...] ['g_msg ...])......5-4
(multiple-value-bind 'l_varlist 'g_values-form 'g_form1 ['g_form2 ...])......14-5
(multiple-value-call 'u_fun 'g_form1 ['g_form2 ...])......14-5
(multiple-value-list 'g_form)......14-5

(**multiple-value-prog1** 'g_form1 ['g_form2 ...]) 14-5
(**multiple-value-setq** 'l_varlist 'g_form) 14-5
(**nconc** 'l_arg1 'l_arg2 ['l_arg3 ...]) 2-7
(**ncons** 'g_arg) 2-1
(**neq** 'g_x 'g_y) 2-10
(**new-vector** 'x_size ['g_fill ['g_prop]]) 2-15
(**new-vectori-byte** 'x_size ['g_fill ['g_prop]]) 2-15
(**new-vectori-long** 'x_size ['g_fill ['g_prop]]) 2-15
(**new-vectori-word** 'x_size ['g_fill ['g_prop]]) 2-15
(**not** 'g_arg) 2-10
(**nreconc** 'l_arg 'g_arg) 2-8
(**nreverse** 'l_arg) 2-8
(**nth** 'x_index 'l_list) 2-4
(**nthcdr** 'x_index 'l_list) 2-4
(**nthchar** 's_arg 'x_index) 2-13
(**nthelem** 'x_arg1 'l_arg2) 2-4
(**null** 'g_arg) 2-10
(**numberp** 'g_arg) 3-2
(**numbp** 'g_arg) 3-2
(**nwritn** ['p_port]) 5-5
(**oblist**) 4-14
(**oddp** 'x_arg) 3-3
(**onep** 'g_arg) 3-3
(**opval** 's_arg ['g_newval]) 6-4
(**or** [g_arg1 ...]) 4-15
(**outfile** 's_filename ['st_type]) 5-5
(**patom** 'g_exp ['p_port]) 5-5
(**plist** 's_arg) 2-13
(**plist** 's_name) 2-22
(**plus** ['n_arg1 ...]) 3-1
(**plusp** 'n_arg) 3-3
(**pntlen** 'xfs_arg) 5-5
(**portp** 'g_arg) 5-6
(**pp** [l_option] s_name1 ...) 5-6
(**pp-form** 'g_form ['p_port]) 5-6
(**princ** 'g_arg ['p_port]) 5-6
(**print** 'g_arg ['p_port]) 5-6
(**probef** 'st_file) 5-6
(**process** s_pgrm [s_frompipe s_topipe]) 6-5
(**product** ['n_arg1 ...]) 3-2
(**prog** l_vrbls g_exp1 ...) 4-15
(**prog1** 'g_exp1 ['g_exp2 ...]) 4-15
(**prog2** 'g_exp1 'g_exp2 ['g_exp3 ...]) 4-15
(**progn** 'g_exp1 ['g_exp2 ...]) 4-15
(**progv** 'l_locv 'l_initv g_exp1 ...) 4-15
(**ptime**) 6-5
(**ptr** 'g_arg) 2-26
(**purcopy** 'g_exp) 4-14
(**purep** 'g_exp) 4-16
(**putaccess** 'a_array 'su_func) 2-19
(**putaux** 'a_array 'g_aux) 2-19
(**putd** 's_name 'u_func) 4-16
(**putdata** 'a_array 'g_arg) 2-19
(**putdelta** 'a_array 'x_delta) 2-19
(**putdisc** 'y_func 's_discipline) 2-21

(**putlength** 'a_array 'x_length) 2-19
(**putprop** 'ls_name 'g_val 'g_ind) 2-23
(**quote** g_arg) 2-26
(**quote!** [g_qform_i] ...[! 'g_eform_i] ... [!! 'l_form_i] ...) 2-2
(**quotient** ['n_arg1 ...]) 3-2
(**random** ['x_limit]) 3-7
(**ratom** ['p_port ['g_eof]]) 5-7
(**read** ['p_port ['g_eof]]) 5-7
(**readc** ['p_port ['g_eof]]) 5-7
(**readlist** 'l_arg) 5-7
(**remainder** 'i_dividend 'i_divisor) 3-7
(**rematom** 's_arg) 2-12
(**remhash** 'g_key 'H_htab) 14-3
(**remob** 's_symbol) 2-12
(**remove** 'g_x 'l_l) 2-6
(**removeaddress** 's_name1 ['s_name2 ...]) 5-7
(**remprop** 'ls_name 'g_ind) 2-23
(**remq** 'g_x 'l_l ['x_count]) 2-6
(**replace** 'g_arg1 'g_arg2) 2-26
(**reset**) 6-5
(**resetio**) 5-7
(**return** ['g_val]) 4-16
(**reverse** 'l_arg) 2-8
(**rot** 'x_val 'x_amt) 3-6
(**rplaca** 'lh_arg1 'g_arg2) 2-5
(**rplacd** 'lh_arg1 'g_arg2) 2-5
(**rplacx** 'x_ind 'h_hunk 'g_val) 2-20
(**sassoc** 'g_arg1 'l_arg2 'sl_func) 2-21
(**sassq** 'g_arg1 'l_arg2 'sl_func) 2-21
(**scons** 'x_arg 'bs_rest) 2-27
(**segment** 's_type 'x_size) 6-6
(**selectq** 'g_key-form [l_clause1 ...]) 4-16
(**set** 's_arg1 'g_arg2) 2-13
(**set-in-fclosure** 'v_fclosure 's_symbol 'g_newvalue) 2-25
(**setarg** 'x_argnum 'g_val) 4-16
(**setf** g_refexpr 'g_value) 2-27
(**setplist** 's_atm 'l_plist) 2-14
(**setplist** 's_atm 'l_plist) 2-22
(**setq** s_atm1 'g_val1 [s_atm2 'g_val2 ... ...]) 2-13
(**setsyntax** 's_symbol 's_synclass['ls_func]) 7-10
(**setsyntax** 's_symbol 's_synclass ['ls_func]) 5-7
(**shell**) 6-6
(**showstack**) 6-6
(**signal** 'x_signum 's_name) 6-6
(**signp** s_test 'g_val) 2-9
(**sin** 'fx_angle) 3-4
(**sizeof** 'g_arg) 6-6
(**sload** 's_file) 5-7
(**small-segment** 's_type 'x_cells) 6-6
(**sort** 'l_data 'u_comparefn) 2-27
(**sortcar** 'l_list 'u_comparefn) 2-27
(**sqrt** 'fx_arg) 3-7
(**sstatus appendmap** g_val) 6-7
(**sstatus automatic-reset** g_val) 6-7

(**sstatus chainatom** g_val) 6-7
(**sstatus dumpcore** g_val) 6-7
(**sstatus evalhook** g_val) 6-7
(**sstatus feature** g_val) 6-7
(**sstatus ignoreeof** g_val) 6-8
(**sstatus nofeature** g_val) 6-8
(**sstatus translink** g_val) 6-8
(**sstatus uctolc** g_val) 6-8
(**sstatus** g_type g_val) 6-7
(**status ctime**) 6-8
(**status feature** g_val) 6-8
(**status features**) 6-9
(**status isatty**) 6-9
(**status localtime**) 6-9
(**status syntax** s_char) 6-9
(**status undeffunc**) 6-9
(**status version**) 6-9
(**status** g_code) 6-8
(**step** s_arg1...) 15-1
(**sticky-bignum-leftshift** 'bx_arg 'x_amount) 3-5
(**store** 'l_arexp 'g_val) 2-19
(**stringp** 'g_arg) 2-9
(**sub1** 'n_arg) 3-2
(**sublis** 'l_alst 'l_exp) 2-22
(**subpair** 'l_old 'l_new 'l_expr) 2-7
(**subst** 'g_x 'g_y 'l_s) 2-6
(**substring** 'st_string 'x_index ['x_length]) 2-13
(**substringn** 'st_string 'x_index ['x_length]) 2-13
(**sum** ['n_arg1 ...]) 3-1
(**symbolp** 'g_arg) 2-9
(**symeval** 's_arg) 2-12
(**symeval-in-fclosure** 'v_fclosure 's_symbol) 2-25
(**sys:access** 'st_filename 'x_mode) 6-9
(**sys:chmod** 'st_filename 'x_mode) 6-9
(**sys:getpid**) 6-9
(**sys:link** 'st_oldfilename 'st_newfilename) 6-9
(**sys:time**) 6-9
(**sys:unlink** 'st_filename) 6-9
(**tab** 'x_col ['p_port]) 5-8
(**tailp** 'l_x 'l_y) 2-3
(**tconc** 'l_ptr 'g_x) 2-24
(**terpr** ['p_port]) 5-8
(**terpri** ['p_port]) 5-8
(**throw** 'g_val [s_tag]) 4-16
(**time-string** ['x_seconds]) 6-10
(**times** ['n_arg1 ...]) 3-2
(**top-level**) 6-10
(**trace** [ls_arg1 ...]) 11-1
(**traceargs** s_func [x_level]) 11-3
(**tracedump**) 11-3
(**tyi** ['p_port]) 5-8
(**tyipeek** ['p_port]) 5-8
(**tyo** 'x_char ['p_port]) 5-8
(**type** 'g_arg) 2-9
(**typep** 'g_arg) 2-9

(uconcat ['stn_arg1 ...]) 2-11
(untrace [s_arg1 ...]) 11-4
(untyi 'x_char ['p_port]) 5-8
(unwind-protect g_protected [g_cleanup1 ...]) 4-16
(valuep 'g_arg) 2-9
(values ['g_arg1 ... 'g_argn]) 14-5
(values-list 'l_arg) 14-5
(vector ['g_val0 'g_val1 ...]) 2-15
(vectori-byte ['x_val0 'x_val2 ...]) 2-15
(vectori-long ['x_val0 'x_val2 ...]) 2-15
(vectori-word ['x_val0 'x_val2 ...]) 2-15
(vectorip 'v_vector) 2-9
(vectorp 'v_vector) 2-9
(vget 'Vv_vect 'g_ind) 2-16
(vprop 'Vv_vect) 2-16
(vputprop 'Vv_vect 'g_value 'g_ind) 2-16
(vref 'v_vect 'x_index) 2-15
(vref-float 'v_vec 'x_index) 14-7
(vrefi-byte 'V_vect 'x_bindex) 2-15
(vrefi-long 'V_vect 'x_lindex) 2-15
(vrefi-word 'V_vect 'x_windex) 2-15
(vset 'v_vect 'x_index 'g_val) 2-16
(vset-float 'v_vec 'x_index 'f_value) 14-7
(vseti-byte 'V_vect 'x_bindex 'x_val) 2-16
(vseti-long 'V_vect 'x_lindex 'x_val) 2-16
(vseti-word 'V_vect 'x_windex 'x_val) 2-16
(vsetprop 'Vv_vect 'g_value) 2-16
(vsize 'Vv_vect) 2-16
(vsize-byte 'V_vect) 2-16
(vsize-word 'V_vect) 2-16
(wait) 6-10
(with-keywords 'l_keys 'l_keydefs 'g_form1 ['g_form2 ...]) 14-6
(xcons 'g_arg1 'g_arg2) 2-1
(zapline) 5-8
(zerop 'g_arg) 3-3

APPENDIX B

Special Symbols

The values of these symbols have a predefined meaning. Some values are counters, while others are simply flags whose value the user can change to affect the operation of the Lisp system. In all cases, only the value cell of the symbol is important; the function cell is not. The value of some of the symbols (like **ER%misc**) are functions. What this means is that the value cell of those symbols either contains a lambda expression, a binary object, or symbol with a function binding.

The values of the special symbols are:

$gccount$ — The number of garbage collections which have occurred.

$gcprint — If bound to a non nil value, then, after each garbage collection and subsequent storage allocation, a summary of storage allocation is printed.

$ldprint — If bound to a non nil value, then, during each *fasl* or *cfasl*, a diagnostic message is printed.

ER%all — The function that is the error handler for all errors. (See Chapter §10)

ER%brk — The function that is the handler for the error signal generated by the evaluation of the *break* function. (See Chapter §10).

ER%err — The function that is the handler for the error signal generated by the evaluation of the *err* function. (See Chapter §10).

ER%misc — The function that is the handler of the error signal generated by one of the unclassified errors. (See Chapter §10). Most errors are unclassified at this point.

ER%tpl — The function that is the handler to be called when an error has occurred which has not been handled. (See Chapter §10).

ER%undef — The function that is the handler for the error signal generated when a call to an undefined function is made.

^w — When it is bound to a non-nil value, this prevents output to the standard output port (poport) from reaching the standard output (usually a terminal). Note that ^w is a two character symbol and should not be confused with ^W which is how control-w is denoted. The value of ^w is checked when the standard output buffer is flushed, which occurs after a *terpr*, *drain*, or when the buffer overflows. This is most useful in conjunction with ptport described later. System error handlers rebind ^w to nil when they are invoked to ensure that error messages are not lost. (This was introduced for Maclisp compatibility.)

defmacro-for-compiling — This has an effect during compilation. If it is non-nil, it causes macros defined by defmacro to be compiled and included in the object file.

environment — The operating system environment in assoc list form.

errlist — When a *reset* is done, the value of errlist is saved away and control is thrown to the top level. *Eval* is then mapped over the saved away value of this list.

errport — This port is initially bound to the standard error file.

evalhook — The value of this symbol, if bound, is the name of a function to handle evalhook traps (see §14.4)

float-format — The value of this symbol is a string that is the format to be used by print to print flonums. See the documentation on the operating system function printf for a list of allowable formats.

funcallhook — The value of this symbol, if bound, is the name of a function to handle funcallhook traps. (See Chapter §14.4).

gcdisable — If it is non-nil, then garbage collections are not done automatically when a collectable data type runs out.

ibase — This is the input radix used by the Lisp reader. It may be either eight or ten. Numbers followed by a decimal point are assumed to be decimal regardless of what ibase is.

linel — The line length used by the pretty printer, pp. This should be used by *print* but it is not at this time.

multiple-values-limit — The maximum number of multiple values that can be returned. This is a read-only variable.

nil — This symbol represents the null list and, thus, can be written (). Its value is always nil. Any attempt to change the value results in an error.

piport — Initially bound to the standard input (usually the keyboard). A read with no arguments reads from piport.

poport — Initially bound to the standard output (usually the terminal console). A print with no second argument writes to poport. See also: ^w and ptport.

prinlength — If this is a positive fixnum, then the *print* function prints no more than prinlength elements of a list or hunk and further elements abbreviated as '...'. The initial value of prinlength is nil.

prinlevel — If this is a positive fixnum, then the *print* function prints only prinlevel levels of nested lists or hunks. Lists below this level are abbreviated by '&' and hunks below this level are abbreviated by a '%'. The initial value of prinlevel is nil.

ptport — Initially bound to nil. If bound to a port, then all output sent to the standard output is also sent to this port as long as this port is not also the standard output since this would cause a loop. Note that ptport does not get a copy of whatever is sent to poport if poport is not bound to the standard output.

readtable — The value of this is the current readtable. It is an array, but you should NOT try to change the value of the elements of the array using the array functions. This is because the readtable is an array of bytes and the smallest unit the array functions work with is a full word (4 bytes). You can use *setsyntax* to change the values and *(status syntax ...)* to read the values.

t — This symbol always has the value t. It is possible to change the value of this symbol for short periods of time, but you are strongly advised against it.

top-level — In a Lisp system without /lisp/lib/tpl.l loaded, after a *reset* is done, the Lisp system *funcall's* the value of top-level if it is non-nil. This provides a way for you to introduce your own top level interpreter. When /lisp/lib/tpl.l is loaded, it sets top-level to tpl and changes the *reset* function so that once tpl starts, it cannot be replaced by changing top-level. tpl does provide a way of changing the top level however, and that is through user-top-level.

user-top-level — If this is bound, then after a *reset* the top level function *funcall's* the value of this symbol rather than going through a read eval print loop.

APPENDIX C

The Garbage Collector

The FRANZ LISP storage management "garbage collector" is invoked automatically whenever a collectable data type's current allocation is exhausted. All data types are collectable except for strings. After a garbage collection finishes, the collector calls the function *gcafter*, which should be a lambda of one argument. The argument passed to *gcafter* is the name of the data type that ran out and which caused the garbage collection. It is *gcafter*'s responsibility to allocate more pages of free space. The default *gcafter* makes its decision based on the percentage of space still in use after the garbage collection. If there is a large percentage of space still in use, *gcafter* allocates a larger amount of free space than if only a small percentage of space is still in use. The default *gcafter* also prints a summary of the space in use if the variable *$gcprint* is non-nil. The summary always includes the state of the list and fixnum space, and includes an additional type if that type caused the garbage collection. The type that provoked the garbage collection is preceded by an asterisk.

Tektronix®
COMMITTED TO EXCELLENCE

MANUAL CHANGE INFORMATION

PRODUCT 4400P30 LISP PROGRAMMERS REFERENCE CHANGE REFERENCE C1/285

MANUAL PART NO. 070-5607-00 DATE 2-1-85

TEXT CHANGES

This is a page replacement package.

Remove the appropriate pages from your manual and insert the attached pages. Keep this cover sheet in the Change Information section at the very back of this manual for a permanent record.

REVISED: Pages 13-1 through 13-6.

ADDED: Pages iv, v, vi, 13-7, and 13-8.