TEK PROGRAMMERS
REFERENCE

Part No. 070-5607-01
Product Group 07

4400P30
FRANZ LISP

Tektronix
COMMITTED TO EXCELLENCE

MANUAL REVISION STATUS

PRODUCT: 4400P30 Franz Lisp Programming Language

This manual supports the following versions of this product: Version 42

REV DATE	DESCRIPTION
JAN 1985	Original Issue
FEB 1985	Revised: Pages 13-1 through 13-6. Added: Pages iv, v, vi, 13-7, and 13-8.
JUL 1985	Completely revised and rewritten.

Contents

CHAPTER 1

FRANZ LISP

1.1. Introduction

This document is a reference manual for the FRANZ LISP system. It is not a LISP primer or introduction to the language. It assumes that you are familiar with at least one dialect of LISP, preferably a member of the MacLISP / Common LISP family.

A recommended text for learning LISP, with specific reference to FRANZ LISP is *LISPcraft* by Robert Wilensky, published by W. W. Norton (1984).

This chapter describes the data types of FRANZ LISP and the conventions used in the description of the FRANZ LISP. functions. In an attempt to be concise, we use a shorthand given in §1.3. It is very important that these conventions be read for a full understanding of subsequent sections. You should refer to the table in that section if the data types of function arguments are in question.

1.2. Data Types

FRANZ LISP has a collection of data types for system implementation and programming. This section describes each type briefly, 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 value (conceptually, at least) pushed on a stack, 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 value is popped off the stack.

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 is 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 identified from input and referred to on (printed) output.

The function *intern* is the usual way of creating a symbol, but the functions *concat*, *maknam*, and their derivatives also create symbols. Usually, symbols are a member of a package, but not always (see chapter 19 for a description of packages).

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
home package	symbol-package		package

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*", "*void-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^{29} to $2^{29}-1$. Small fixnums (-1024 to 1023) are stored in a special table so they needn't be allocated each time one is needed.

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.

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 on most machines, more on others). 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 *infile or outfile*). During initialization, FRANZ LISP binds the symbol **piport** to a port attached to the standard input stream. This port prints as #<port stdin>. There are ports connected to the standard output and error streams, which print as #<port stdout> and #<port 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, except for compatability reasons. 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.

[†]The current algorithms for integer arithmetic operations return (in certain cases) a result as a bignum although this could just barely be represented as a fixnum.

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 the user, 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 that for a list cell element but faster than for an array element. 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* (dotted pair) 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.

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

1.2.13. package Chapter 17 describes the FRANZ LISP package system. The ideas, adopted from Common LISP, are helpful for the development of large systems in a modular fashion.

1.2.14. closure The fclosure functional object is introduced in chapter 2, but described in greater detail in chapter 8.

1.2.15. 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 initial letter or letters give the allowable type or types for that argument according to this table.

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
H	hash table
a	array
e	value
p	port (or nil)
h	hunk
k	package
cl	closure

In the first line of a function description we provide a template showing you how to call the function from the top level, including appropriate quote marks. (In FRANZ LISP, the form 'foo is the same as (quote foo).) For example, cons could be described via (cons

'g_first 'g_rest), allowing us to say the result is the list cell x where (car x) = g_first, and (cdr x) = g_rest. Referring to the previous table, we see that the 'g' prefix means 'general' or 'any type'. The suffixes first and rest are chosen to be mnemonically useful. Those arguments preceded by a quote mark are evaluated before the function is applied. This allows us to refer to the result of the cons without saying "the result of evaluating its first argument ..." etc.

Occasionally a function will utilize some special order of evaluation (or non-evaluation) but if the arguments are generally evaluated, we will still use this notation.

When an argument is not quoted in the function description line, it is usually because no evaluation is done on that argument. Rarely, however, the argument is evaluated but at a time 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.

1.4. Some History

The original version of FRANZ LISP was created as a research tool for symbolic and algebraic manipulation, artificial intelligence, and programming languages at the University of California at Berkeley. As FRANZ LISP grew, it adopted numerous features of MacLISP and LISP Machine LISP (Zetalisp). Substantial compatibility with other LISP dialects (Interlisp, UCILISP) is achieved by means of support packages and compiler switches. Beginning in 1984, Franz Inc.'s Common LISP compatibility features were introduced making this version of FRANZ LISP a unique combination of a proven LISP system with the essential support for development and delivery of Common LISP programs.

The kernel of FRANZ LISP is written almost entirely in the programming language C, with much of the support written in (compiled) LISP. For run-time efficiency, small portions of the kernel are written in assembly language.

FRANZ LISP's distinctive features include its capability of running very large LISP programs in a timesharing environment, its excellent facilities for arrays and user-defined structures, its user-controlled reader with character and word macro capabilities, and its ability to interact directly with compiled LISP, C, Fortran, and Pascal code in most implementations.

CHAPTER 2

Data Structure Access

The functions described in this chapter allow you to create and manipulate the various types of lisp data structures. Refer to §1.2 for a brief overview of the data structures available in 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 ...]**)**
(list* ['g_arg1 ...]**)**

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

NOTE: *List** differs from list in that the last argument is cons'd on to the list.

EXAMPLE: (list 'x 'y '(z w)) is (x y (z w))
(list* 'x 'y '(z w)) is (x y z w)

(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.

(append1 '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 append1
-> (append1 '(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)
(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.

(litatom 'g_arg)

RETURNS: t if g_arg is a string or symbol, not a number.

(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.

(keywordp 's_sym)

RETURNS: t if the argument is a symbol and is in the keyword package.

(packagep 'k_package)

RETURNS: t if the argument is a package.

(hash-table-p 'H_arg)

RETURNS: t if H_arg is a hash table, nil otherwise.

NOTE: A hash table is implemented using vectors, and the function *typep* returns **vector** when given a hash table.

(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 $\leq$ 0
e	g_val $=$ 0
n	g_val $\neq$ 0
ge	g_val $\geq$ 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**)**
(nequal 'g_x 'g_y**)**

RETURNS: (for *neq,*) t if g_x is not *eq* to g_y, otherwise nil. (for *nequal,*) t if g_x is not *equal* 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, package-cell, 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 ...])
(strcat ['stn_arg1 ...])

RETURNS: A symbol whose print name (or in the case of *strcat*, a string) is the result of concatenating the print names, string characters, or numerical representations of the stn_arg*i*.

NOTE: If no arguments are given, a symbol with a null pname is returned. *concat* interns (see intern) the symbol in the current package; the function *uconcat* does the same thing but does not intern the symbol.

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

(concatl 'l_arg)

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

(implode 'l_arg)
(implodes 'l_arg)
(maknam 'l_arg)

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

RETURNS: The symbol whose print name (or in the case of *implodes*, the string) that 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.

(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 ['k_package])

RETURNS: s_arg

SIDE EFFECT: s_arg is installed in the package k_package or (by default) the current package.

NOTE: When a symbol is interned, a mapping between the external print-name and the symbol itself is established. Most symbols are interned so that when the user types in say, *foo,* this is the same symbol as previously used. It is possible to change an existing *foo* to be uninterned; in that case when a token *foo,* is read in, a new symbol, unrelated to the old one except by its coincidentally matching print-name, is created. The two symbols will ordinarily have different values, etc. One can refer to the old *foo* only by having an internal pointer to it. As far as the reader is concerned, there's no *foo* like the old *foo.*

(remob 's_symbol)

RETURNS: s_symbol

SIDE EFFECT: s_symbol is removed from the current package. Historically, the name "remob" comes from "remove from the object list".

(rematom 's_arg)

RETURNS: t if s_arg is indeed an atom.

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

NOTE: This function does *not* check to see if s_arg is accessible. 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 with short lifetimes.

The following functions are recommended for use in the orderly management of generated symbols:

(newsym 's_name)
(gensym 's_name)

RETURNS: a new uninterned symbol whose name will be s_name with an integer concatenated to the right.

SIDE EFFECT: The integer used (by default initially 0) will be incremented at each use (and can be initialized to some other value by *initsym,* described below.

NOTE: *Gensym,* the original system function, uses only the first letter of s_name, and manufactures a symbol with that prefix and a six-digit suffix which is the number of times gensym has been called.

(oldsym 's_name)

RETURNS: the last symbol returned from *newsym* in the series generated from s_name. If s_name has not been used, the return value is just s_name.

(allsym 'sl_arg)

RETURNS: a list of symbols generated by newsym.

NOTE: If sl_arg is a symbol, all newsymed symbols with that prefix will be in the list returned. If sl_arg is a pair: (name number), then only symbols beginning with the name and generated with suffix equal to number or higher will be listed.

(initsym 'l_list1 ...)

WHERE: each l_listi is of the form (symbol fixnum).

RETURNS: a list of generated symbols serving as the initialization of symbol generators.

SIDE EFFECT: Future calls to newsym may be affected by this initialization.

EXAMPLE: -> (initsym '(foo 10) '(bar 3))
(foo10 bar3) -> (newsym 'bar) bar4

(remsym 'sl_list1 ...)

WHERE: each sl_listi is a symbol or a list.

SIDE EFFECT: If sl_listi is a symbol, all the generated symbols with that prefix are uninterned. If sl_listi is a pair (name number), uninterning is done on symbols beginning with the name and with suffix equal to the given number or higher.

(symstat 's_name1 ...)

RETURNS: a list of pairs: (name number) for each s_name1, where s_namei is presumably a prefix for generated symbols, the number in the pair is the last number used in a gensym with that prefix. If the name has not been used at all the second element of the pair will be nil.

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.

(str= 't_string1 't_string2)

RETURNS: t if t_string1 is equal to t_string2.

NOTE: An error is signalled if the arguments are not strings.

2.3.3. symbol and string accessing

(symeval 's_arg)
(symbol-value 's_arg)

RETURNS: The value of symbol s_arg. *Symbol-value* is the Common Lisp name for this function.

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)
(symbol-name 's_arg)

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

NOTE: *Symbol-name* is the Common Lisp name for this function.

(getd 's_arg)
(symbol-function 's_arg)

RETURNS: the function definition of s_arg. If there is no function definition, *getd* returns nil; *symbol-function,* the Common Lisp function signals an error in this case.

NOTE: The function definition may turn out to be an array header. You might wish to use *fboundp* to see if a function definition exists in a Common Lisp world, rather than *getd.*

(symbol-package 's_name)

RETURNS: the contents of the package cell for the symbol s_name. This will be nil if there is no package associated with s_name.

(fboundp 's_arg)

RETURNS: t if s_arg is bound to a function.

(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.

(char-index 't_string 'stx_char)
(char-rindex 't_string 'stx_char)

RETURNS: the index of stx_char in t_string, from the beginning of the string, in the case of char-index, and the end of the string in the case of char-rindex. If stx_char is a fixnum it will be treated as the ascii code for the character. If stx_char is a symbol or string, the first character of the print name or the string is used.

(substrp 't_string1 'string2)

RETURNS: t if t_string1 is contained in t_string2, nil otherwise.

(string 'st_symbol-or-string)

RETURNS: a string, given a string or symbol. In case of a symbol arg it returns the symbol's print name.

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.

(explode 'g_arg)
(explodec 'g_arg)
(exploden 'g_arg)
(escape-exploden 'g_arg)
(qualify-explode 'g_arg)
(qualify-explodec 'g_arg)
(qualify-exploden 'g_arg)
(qualify-escape-exploden 'g_arg)

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

NOTE: There are a set of functions called "aexplode" which mirror the above functions, except that they only take symbols as arguments.

NOTE: If the function name ends in a "n", then the function returns a fixnum representation of the character instead of the character itself. Also, the functions beginning with "qualify" return a list of characters or fixnums which include package qualifiers.

NOTE: The functions *explode*, *escape-exploden*, *qualify-explode* and *qualify-escape-exploden* return a list of characters or fixnums that *print* would use to print the argument, which include all necessary escape characters. The functions *explodec*, *exploden*, *qualify-explodec* and *qualify-exploden* return a list of characters or fixnums that *patom* would use to print the argument (that is, no escape characters).

```
=> (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.

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]])
(new-vectori-float 'x_size ['g_fill ['g_prop]])
(new-vectori-double '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, 4 for new-vectori-long or new-vectori-float, or 8 for new-vectori-double. 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.

NOTE: new-vectori-float and new-vectori-double are intended to be used in passing float and double arrays to C routines.

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 ...])
(vectori-float ['f_val0 'f_val2 ...])
(vectori-double ['f_val0 'f_val2 ...])

RETURNS: A **vectori** with as many data elements as there are arguments. The arguments are required to be fixnums, except in the case of float and double, where they must be flonums. 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)
(vrefi-float 'V_vect 'x_lindex)
(vrefi-double '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)
(vsize-float 'V_vect)
(vsize-double '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 comprosed of bytes or words.

2.4.3. vector modification

(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)
(vseti-float 'V_vect 'x_lindex 'f_val)
(vseti-double 'V_vect 'x_lindex 'f_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. 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.

2.5.1. hash table functions

(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 parameters are ignored at this time, and no automatic rehashing is done.

NOTE: For an explanation of keyword arguments, see section 8.2.

(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. As is the standard with a multiple-value return, if only one value is expected, the first is used.

NOTE: *setf* may be used to set the value associated with a key.

(addhash 'g_key 'H_htab 'g_val)

RETURNS: g_key, after installing 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. It is, however, acceptable to use remhash or setf of gethash on the entry currently being mapped over.

(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 (makehash-table :size 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 (make-hash-table :size 10 :test #'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)
```

2.6. 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.

2.6.1. 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 whose meanings are indicated in chapter 9 (see also § 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.

2.6.2. array predicate

2.6.3. 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_ind1 ... **)**

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)).
```

2.6.4. 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.

2.7. 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.

2.7.1. 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.

2.7.2. 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.

2.7.3. 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

2.8. 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.

2.9. 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.

2.9.1. 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)
(rassq '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. Rassq is similar to assq, with the 'r' indicating the reversal that it looks at cdr's instead of car's in l_arg2.

(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.

2.9.2. 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**)**
(symbol-plist 's_name**)**

RETURNS: The property list of s_name.

NOTE: *Symbol-plist* is the Common Lisp name for this function.

(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
```

2.9.3. 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)
```

2.9.4. 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.

(fclosure-list 'l_vars1 'g_fcnobj1 [... ...])

RETURNS: a list of fclosures where variables with the same name are shared between fclosures.

NOTE: *fclosure-list* creates a set of fclosures which share variables. If you create fclosures with the *fclosure* function, they will not share closed-over variables. The following example shows the difference between *fclosure-list* and a pair of *fclosure* calls.

```
; intialize x and then make two closures over it
=> (setq x 10)
10
=> (setq cla (fclosure '(x) '(lambda (y) (setq x y))))
#<fclosure 2>
; This fclosure will have the same initial value as the one above but
; it will not share the value of x
=> (setq clb (fclosure '(x) '(lambda () (print x))))
#<fclosure 2>
; To demonstrate that the closures don't share the value of x
; we call the first closure function:
=> (funcall cla 20)
(funcall cla 20)
20
; and then see that the value of the second has not been affected:
=> (funcall clb)
10nil

; now we create two closures with fclosure-list
; since both closures are over x, the value of x will be shared
=> (setq clc (fclosure-list '(x) '(lambda (y) (setq x y))
                                 '(x) '(lambda () (print x))))
(#<fclosure 2> #<fclosure 2>)
; to demonstrate this, we call the first one to set the value of x
=> (funcall (car clc) 15)
15
; ... and then call the second one to print the value of x
=> (funcall (cadr clc))
15nil
=>
```

(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.

2.10. 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 and Logical Functions

This chapter describes FRANZ LISP's functions for manipulation of numeric quantities. Often the same function is known by several names for the sake of compatibility with several lisp dialects. For example, *add* is also *plus* and *sum*. However, you should avoid using functions with names such as + and * in this version of FRANZ LISP unless their arguments are fixnums. The + function in Common Lisp has been defined to allow mixed mode arguments (an incompatibility with pre-existing lisps such as FRANZ LISP) and corresponds to the FRANZ LISP function *plus*. The mapping of + to *plus* (etc.) may be easily achieved via the package system for Common Lisp compatibility. Note that FRANZ LISP compiler can take advantage of 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.

(*invmod 'x_number 'x_modulus)

RETURNS: This function returns the inverse of x_number in the finite field of the integers modulo the odd prime x_modulus. The result is expressed as a positive or negative fixnum of magnitude less than x_modulus/2.

NOTE: This is useful in algebraic manipulation and number theory calculations.

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.

(primep 'x_arg)

RETURNS: t iff x_arg is a prime integer in the range of fixnums.

NOTE: This is intended to be fast and convenient for, for example, hash-table construction.

3.3. Trignometric Functions

Some of these functions are taken from the host math library, and therefore have the same accuracy and other performance characteristics.

(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)
(integer-length '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. Common Lisp's integer-length differs from haulong in that if the argument is negative, the result is equal to the least integer greater than or equal to the base two logarithm of 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

In this section numerous functions based on the *boole* function are defined for Common Lisp compatibility. Many are merely macro-expanded into calls to *boole,* and in the current implementation are restricted to fixnum-length integers.

(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)
(ash '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. Functionally "arithmetic shift" *ash,* is the same as *lsh.* 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.

(logand ['n_arg1 ...])

RETURNS: the (numeric) result of the bitwise logical *and* of the n_argi's.

NOTE: If there are no arguments, -1 is returned.

(logior ['n_arg1 ...])

RETURNS: the (numeric) result of the bitwise logical *inclusive or* of the n_argi's.

NOTE: If there are no arguments, zero is returned.

(logxor ['n_arg1 ...])

RETURNS: the (numeric) result of the bitwise logical *exclusive or* of the n_argi's.

NOTE: If there are no arguments, zero is returned.

(logeqv ['n_arg1 ...])

RETURNS: the (numeric) result of the bitwise logical *equivalence* of the n_argi's.

NOTE: If there are no arguments, -1 is returned. Equivalence is the same as *exclusive nor.*

(lognot 'n_arg)

RETURNS: the (numeric) result of the bitwise logical *not* of n_arg. (The logical complement of n_arg).

Additional Common Lisp logical operators which require exactly two fixnum arguments are as follow:

(logandc1 'n_arg1 'n_arg2)

RETURNS: (logand (lognot n_arg1) n_arg2)

(logandc2 'n_arg1 'n_arg2)

RETURNS: (logand n_arg1 (lognot n_arg2))

(logbitp 'x_index 'n_number)

RETURNS: t if the bit in n_number with index *index* is

(logcount 'n_number)

RETURNS: the number of one bits in n_number if n_number is positive; the number of zero bits if n_number is negative.

(logiorc1 'n_arg1 'n_arg2)

RETURNS: (logior (lognot n_arg1) n_arg2)

(logiorc2 'n_arg1 'n_arg2)

RETURNS: (logior n_arg1 (lognot n_arg2))

(lognand 'n_arg1 'n_arg2)

RETURNS: (lognot (logand n_arg1 n_arg2))

(lognor 'n_arg1 'n_arg2)

RETURNS: (lognot (logor n_arg1 n_arg2))

(logtest 'n_arg1 'n_arg2)

RETURNS: (not (zerop (logand n_arg1 n_arg2)))

NOTE: Logtest is true if any one's are in corresponding locations in the two arguments.

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

4.1. Introduction

This chapter describes the special functions, or forms of FRANZ LISP. While Lisp is generally thought of as very simple, in fact serious programming in Lisp uses a large helping of these special forms. What makes them special is that they generally do not conform to the usual function evaluation methodology.

4.2. Functions

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

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

NOTE: If u_func is a lambda, then the length of *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.

For application of a lambda (or lexpr), apply will take the optional g_arg's and just push them on the stack (like funcall). The LAST argument, *l_args* should be a list which will be spread on the stack. This version of apply (Opus 42.03 and later) is upward compatible with Common Lisp. If the argument is a nlambda or macro, it still must be given only one argument.

```
; 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)
```

(apropos 'st_arg**)**

RETURNS: nil

NOTE: All packages are searched for symbols whose print name contains the substring st_arg. Matching symbols are printed, along with information about their function definition and current value.

(apropos-list 'st_arg**)**

RETURNS: a list of symbols which is the result of seaching all packages for symbols which have a print name containing the substring st_arg.

(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 *?pop* or *reset*.

(caseq 'g_key-form l_clause1 ...)
(case '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 run-time 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 nothing but return a constant value. You should be caution to avoid using this in a place where the value might be used.

(cond [l_clause1 ...])
(when pred form1 ...)
(unless pred form1 ...)
(if pred form1 [form2])

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

NOTE: Cond 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. The forms *when, unless,* and *if* are expanded into *cond's* as follows:
(when p a b ...) = (cond (p a b ...))
(unless p a b ...) = (cond ((not p) a b ...))
(if p a b) = (cond (p a) (t b))

(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

(defsubst s_name l_llist g_form [...])

RETURNS: s_name.

NOTE: defsubst is like defun (below), except that a compiler macro is defined (see defcmacro above) which expands to the definition of s_name as a lambda. It's use is to allow an easy method of defining functions which are to be macro expanded out at compile time. Note, also, that the & lambda list parameters, which are valid to defun, are not to defsubst.

(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: Some of these options exist for Maclisp compatibility. Defun rearranging the information you provide and then invokes the 'def' procedure. For example, the MacLisp 'fexpr' is automatically converted to a FRANZ LISP nlambda. A MacLisp s_mtype of expr is simply the same thing as FRANZ LISP's lambda. The s_type of macro exists in the same form. If ls_arg1 is a non-nil symbol, then the type is assumed to be lexpr and ls_arg1 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 four types of optional parameters that can occur in ls_argl: *&optional*, *&rest*, *&aux* and *&key*. For an explanation of use of these forms, see section 8.2.
An additional form accepted by defun is provided to set up property lists. To place the value g_value on the property list of symbol s_name under indicator s_ind, use the form
(defun (s_name s_ind) g_value)

```
; 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 then (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_exp1 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.

NOTE: For Common Lisp compatability, the var-init-repeat form may be an atom, which is equivalent to specifing a g_init of nil.

(do s_name g_init g_repeat g_test g_exp1 ...**)**

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_exp1.

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

(6) Go to step 3.

RETURNS: Nil.

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

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.

(**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.

```
; 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
=> (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
=>
```

(**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.

(cli:error 's_format-string ['args])

WHERE: s_format-string is a command string to the format-program, and the arg(s) are passed to format.

RETURNS: Nothing. This function signals a fatal error.

NOTE: The prefix "cli" must be used normally because it is part of the Common LISP package, and conflicts with the FRANZ LISP function of the same name.

(cerror 's_continue-format-string 's_error-format-string ['arg ...])

RETURNS: nil, if the program is continued after the error.

SIDE EFFECT: This function (which is intended for use with continuable errors) signals an error and enters a break loop. The program may be continued after resolution of the error by typing ?ret to the break loop. The two strings given are intended as control strings to the format function to construct a continuation message and an error message. This is entirely compatible with the Common LISP *cerror* function. Explanations of use can be found in *Common LISP the Language.*

(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 §15.4, 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*.

(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.

(getenv 'st_name)

RETURNS: the value of looking up st_name in current environment. Not all operating systems have an environment.

(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.

(help sx_arg**)**

RETURNS: nil.

NOTE: a portion of the online FRANZ LISP manual is printed. If sx_arg is an function, then manual description of that function is printed. If sx_arg is a number or "b" or "c", then print that chapter or appendix. If sx_arg is "tc", then a table of contents is printed.

(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_exp1 ... 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_exp1 ... 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 every interned symbol in the current lisp environment.

NOTE: The name of this function is historical, and means "object list". It used to be the case that all symbols lived on the same entity called the oblist or obarray (for object array).

(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.

(pop 'l_stack ['g_into])

RETURNS: The top element on the stack l_stack. If g_into is given, a *setf* of g_into is done with the top element as the value. See *push* below.

(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.

(push 'g_element 'l_stack)
(pushnew 'g_element 'l_stack)

RETURNS: l_stack.

NOTE: l_stack is a fifo stack, and push *cons*'es g_element into l_stack. pushnew only pushes the element if it is not already there. See *pop* above.

(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.

4.3. Multiple Value Returns

Sometimes a function logically needs to return more than one value, but in most cases only one of them is used and it is therefore considered uneconomical to "cons-up" a structure for the unusual case. A function performing a division might returning the quotient and then as a second part, number might return the remainder. A neat way to do this is by 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 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. With the exception of the first, they can be accessed only by using one of the special forms below to receive them.

(values-list 'l_arg)

RETURNS: the car of l_arg; but if received appropriately will exhibit the elements in the list l_arg as multiple values. This form is equivalent to (apply 'values 'l_arg).

EXAMPLE: (values-list '(1 2 3)) is equivalent to (values 1 2 3)

(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 *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.

CHAPTER 5

Input/Output

5.1. Introduction

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 *close*d. 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.

FRANZ LISP has borrowed a convenient shorthand notation from the Unix operating system 'C' shell concerning naming files. If a file name begins with ~ (tilde), and the symbol **tilde-expansion** is bound to something other than nil, then FRANZ LISP expands the file name. It takes the string of characters between the leading tilde, and the first slash as a user-name. Then, that initial segment of the filename is replaced by the home directory of the user. The null username is taken to be the current user.

On Unix systems, having gone to the effort of searching the password file, FRANZ LISP remembers the user directory, in case it gets asked to do so again. Tilde-expansion is performed in most places in which file names are expect, and in particular, the following functions: *cfasl, chdir, fasl, ffasl, fileopen, infile, load, outfile, probef, sys:access, sys:unlink.*

The programmer should be careful to note which of the functions reference file names and which refer to ports.

5.2. Functions

(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.

(fileopen 's_filename 's_mode])

RETURNS: a port for reading or writing based on the file s_filename. The s_mode is one of **r**, **w**, or **a**, (for read, write, append), or **r+**, **w+**, or **a+** each of which permits both reading and writing on a port provided that *fseek* is done between changes in "direction". In the case of writing, the file will be created if it does not already exist. The directory search-path for file name resolution is *not* used.

(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.

(help sx_arg)

RETURNS: nil.

NOTE: a portion of the online FRANZ LISP manual is printed. If sx_arg is an function, then manual description of that function is printed. If sx_arg is a number or "b" or "c", then print that chapter or appendix. If sx_arg is "tc", then a table of contents is printed.

(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. For objects that can't be read back in, the convention is used that a sharp-sign (#) prefixes the data printed. For example,

The lisp objects *other, bcd, vector, hash-table, package, array,* and *port* now print as "#<...>". Reading the value of printing one of the above objects is an error.

```
=> *package*
#<package user>
=> (getd 'car)
#<bcd 0x16B2E lambda>
=> (make-hash-table)
#<hash-table 20>
=> Standard-Input
#<port $stdin>
=> (vector 10)
#<vector 1>
=> (marray nil nil nil 10 nil)
#<array 10>
=> (getd 'format)
#<bcd 0xD4800 lambda>    ; known to be compiled code, which is of type other
=> (fake #xd4800)
#<other 0xD4800>
```

(sprintf 't_control ['arg1 ...])

RETURNS: the formatted string corresponding to the returned value of the C library sprintf function.

NOTE: The \ (backslash) escapes do NOT work. T_control must be a string.

(wide-print-list 'g_exp [:port 'p_where] [:left-margin 'x_wheretostart])

RETURNS: Nil

SIDE EFFECT: Prints the expression g_exp to to the port p_where, starting at the column number x_wheretostart. *wide-print-list* prints as many entities on one line as will fit assuring that no lisp object other than a list is broken across a line boundary. In particular, atoms and numbers remain on one line.

(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.

(readline ['p_port])

RETURNS: a string containing all characters in the stream p_port up to, but not including, the next newline. The null string is returned if the first character is a newline.

(charcnt 'p_port)

RETURNS: The number of characters left on the current line in p_port.

sys:readdir
(readdir ['t_dirname])

RETURNS: A list of strings, one for each file in the named directory.

NOTE: t_dirname is a string representing the name of a directory. It may be ".", meaning return a list of the current working directory (This is the default if the argument is omitted.) All files are listed except "." and "..". No tilde-expansion is done on the argument.

(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.

(**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.

(**tilde-expand** 'st_name)

RETURNS: st_name will all ~ 's expanded with absolute pathnames in their place.

NOTE: this function is not available in all versions of FRANZ LISP.

(**truename** 'p_port)

RETURNS: The name of the file to which p_port refers.

(**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.

(y-or-n-p ['t_message])

RETURNS: t if an answer of "y" is read from the user, nil otherwise. If t_message is given, then prompt the user with this string before reading.

NOTE: piport and poport are used as the querying ports.

(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.

5.3. Format

Format is a output formatter somewhat in the style of the *fprintf* 'C' run-time library program. The arguments to *format* include an output port, a control string, and the values to be printed. The values passed to *format* are printed according to the directives in the control string.

In the simplest case, the *format* control string has no directives and is passed no variables. In this case, *format* just prints:

(format t "Hello world") ==> "Hello world" nil

As we see from the previous example, *format* prints some expression, and then *returns* a value *nil*. If, instead of *t*, (which indicates the standard output), we had used *nil* as the value for the port, *format* would have *printed* nothing, but *returned* a symbol, *|Hello world|*. Values returned by *format* are uninterned.

(format 'p_port 's_ctrl ['g_arg ...])

RETURNS: *nil* if p_port is non-*nil* otherwise returns a *symbol* determined by s_ctrl and the arguments.

WHERE: p_port is one of: *t* (the user's terminal), an output port opened with *outfile* (or *fileopen*), or *nil* (which causes *format* to return a symbol).

s_ctrl is a control string, enclosed in double quotes ("). It may also contain *directives*, prefaced by a tilde (~), as explained, below.

The number of arguments should correspond to the number of values expected by the control string. All arguments are processed according to the number of directives contained in the control string. Extra arguments are discarded. When an

insufficient number of arguments are given, default values may be inserted.

SIDE EFFECT: when p_port is non-*nil*, prints a string to p_port.

In the example, above, the control string was a simple string, containing no directives and there were no arguments to the string. The remainder of this section will deal with more complex control strings which contain directives which are used to manipulate the output. Directives are invoked using the directive prefix, ~ , (a tilde). In addition, there are a number of *modifiers*, which can be used to further manipulate the output. These modifiers may be in the form of a special character, such as the atsign ("@"), or the colon (":"), or, they may simply be a combination of numbers, indicating, perhaps, the number of significant digits in a numeric expression. Finally, there is a control character, which is flagged by the tilde signifies the directive to be followed. In some cases, directives may only apply to the next item to be printed, and as such, will appear alone in the text of the control string. In other cases, directives may involve more complex strings, and may require a matching "closing" directive (much like the closing right parenthesis of a list). All of these variations will be described and illustrated in the following text. In general, however, the form of the control string is (with the optional end-directive in brackets):

"~ *<prefix><modifier><directive> string...[~ <end-directive>]*"

In certain cases, more than one modifier may be used for a single directive. In such cases, individual modifiers may be separated by a comma. For example, the directive ~ **d** causes the argument to be printed in decimal format. However, ~ ***n*d** causes the argument to be printed to *n* decimal places (using a space as a pad character), while ~ ***n,m*d** will cause the integer to be printed to *n* decimal places using the character whose decimal representation is *m* as the pad character. (In the previous discussion, *n* and *m* are *prefixes*, in contradistinction to *modifiers* such as the atsign and the colon.)

```
; the ~ % outputs a carriage return.

=> (format t "Today's daily number is: ~ d ~ %" 1024)
Today's daily number is: 1024
nil
; the number will be printed to four places but the pad character
; will be a space.
=> (format t "Today's daily number is:~ 4d ~ %" 304)
Today's daily number is:  304
nil
; 48 is the decimal representation of a zero (0)
=> (format t "Today's daily number is: ~ 4,48d ~ %" 304)
Today's daily number is: 0304
nil
=>
```

In some cases, the *prefix* parameter will indicate a character to be printed, such as the padding character in the case of ~ **d**, above. Since *format* expects a decimal number, ascii characters should either be specified by their decimal values (e.g., 48 = "0"), or using a quote sign (e.g., "~ 4,'0d") In addtion, the *prefix* character can be the letter **v** or the sharpsign character, **#**. The **v** prefix directs format to use the value of the current argument as the prefix to the directive. This syntax is illustrated in description of the ~ **q** directive. The other option, **#**, substitutes the *number* of remaining arguments for the prefix character. The use of the sharpsign is illustrated in a number of examples, below.

5.3.1. format directives

The following is a description of the *format* directives have been modeled after those in MacLISP and ZetaLISP. The symbol *arg* will be used to indicate the corresponding argument to the directive.

~ **a** prints any argument, *arg* as it would be printed by *princ* (without escaping). ~ *n***a** prints the argument filling it to width *n* with spaces.

~ **s** is identical to ~ **a** except that the print function is *prin1*.

~ **d** as described in the previous example, prints the decimal representation of *arg* (without a decimal point). If *arg* is not a number, it is printed in ~ **a** format. ~ *n***d** prints *arg* to *n* spaces, the padding character is indicated by ~ *mincolw,padchar***d** where padchar is the decimal representation of the padding character (default is a space but a useful value is decimal 48 which is an ascii 0).

~ **o** similar to ~ **d** except that it prints the argument in octal form (radix 8). Note that *arg* must be a number. See also ~ **r**. ~ *n***f**
prints arg to a precision of *n* digits (minimum 2, because of the decimal point). The decimal pointed is floated to be consistent with standard scientific notation so that extremely large numbers and extremely small number are printed in exponential notation. If *arg* is not a number it is printed in ~ **a** format. Unlike the previous directives, the prefix is not *mincol*, rather, it is total number of digits printed (*n-1* decimal places).]

~ **e** [*arg* is printed in exponential notation. The prefix case is identical to ~ **f** indicative of the number of significant digits. Unlike ~ **f**, *all* values for *arg* are printed in exponential notation.]

~ **r** a general purpose number formatter. The full form of this directive is ~ *radix,mincolw,padchar,commachar* **r** where each of these prefix values has the same meaning as in the ~ **d** directive. The *radix* prefix causes the argument to be printed in that base, i.e, ~ **10r** is equivalent to ~ **d**. Non-numeric arguments are printed using ~ **a**.

If no prefix is given, the numerical argument is printed as text, i.e, if the argument is "4", then ~ **r** prints "four" (cardinal value), ~ **:r** prints "fourth" (ordinal value), and ~ **@r** prints IV (Roman numeral). [~ **:@r** prints IIII (old-style Roman numeral).]

```
=> (format t "The ~ :r one on the right.~ %" 1)
The first one on the right.
nil
; ~ % prints a carriage return.
=> (format t "Total value of door number ~ r: ~ r dollars!~ %" 3 12332)
Total value of door number three: twelve thousand three hundred thirty-two dollars!
nil
; the next example illustrates the use of the ~ * directive which
; allows the user to evaluate the arguments out of order.
=> (format t "The octal representation of ~ d is ~ :*~ 8r. ~ %" 18)
The octal representation of 18 is 22.
nil
```

~ **c** prints *arg* (which should be a fixnum representing an ascii character), in a more human-readable form. ~ **c** prints the non-printable (control) characters represented by the fixnum argument in a slashified octal form. ~ **:c** prints the non-printable characters as an upper case character prefixed by a carat (e.g.,

"^C"); some characters such as decimal 32 (a space) is printed in a text form (e.g., "space"). The ~ **@c** form prints the character in LISP readable form (slashified). In all cases, the printable characters are printed in their keyboard form.

[The ~ **c** allows the character to be displayed along with any control bits which may be set. In ZetaLISP, an alternate character set may be used to indicate non-printing characters such as control characters (using the ~ **c** directive) while characters printed with the ~ **:c** directive will have the control bits spelled out. For example, a character may print out as "Control-Meta-X". Mouse characters will print out "Mouse-<button number>-<number of clicks>". This is not, yet, available in FRANZ LISP.

~ **%** outputs a carriage-return. In most UNIX modes, the operating system will generate a line-feed automatically upon receipt of the carriage-return. A carriage-return may also be inserted directly into the control string, but this can be used to create more readable code. If it becomes necessary to insert a carriage-return, which should not be printed, into code to make it more readable, a simple tilde prefixing the carriage-return will "escape" it. A tilde in front of a space will prevent the space from being echoed on the output. The form ~ *n*% prints *n* carriage returns.

~ **x** outputs a space. ~ *n'.25'*x outputs *n* spaces.

~ **&** calls for a "fresh line", meaning that if the cursor is not in the first column of the current line, output a carriage return, else do nothing.

~ **|** outputs a formfeed on a line oriented output. ~ *n*| outputs n formfeeds. ~ **:|** will output an operation to clear the screen on a screen-oriented output (this requires that the screen has been initialized in a manner described in the next section), otherwise a formfeed will be output.

~ ~ outputs a tilde. If ~ *n*~ is used, *n* tildes are printed.

~ **t** outputs a relative tab (5 spaces).[~ *n,m'.25'*t will cause spaces to be printed to the absolute column *n*, or if it is past *n*, to column *n+mk* where *k* is the smallest unit of increment (1 column on a terminal, 1 pixel on a bitmap). If the colon flag is used, the unit values for *n,m,* and *k* will be in characters (columns) on a standard terminal and in units of pixels on a bitmapped display. If the colon flag is not used, the units are in columns. The syntax is ~ *n,m*:t. Absolute tabs (as opposed to relative tabs), are only possible when the screen mode is such that the cursor position is always known. In ZetaLISP, this means that **:read-cursorpos** and **:set-cursorpos** must be in operation. Currently, there is no equivalent in FRANZ LISP and only relative tabs are allowed.]

~ the tilde character can also be used to "escape" characters which appear in the control string but which should not be printed. Most commonly these characters consist of the space and the carriage return. For example, in some circumstances the control string may stretch out over two or more lines, and the user may wish to insert carriage returns to make more readable code. This can be done if the carriage return is the character immediately following the tilde. Similarly, a single space immediately following a tilde will not be printed.

~ **p** Modifies text to include plural syntax. For example, if *arg* is not "1", then the character "s" is printed. ~ **@p** prints "ies" if the argument is not one, else it prints "y". The colon modifier causes the *last* argument, rather than the *current* argument to be used. *arg* need not be a number.

```
=> (format t "Yes, we have ~ a banana~ :p[~ ~ %" "no")
Yes, we have no bananas!
nil
; the "~ [" directive is explained in the following section
=> (format t "Here ~ [~ 1;is~ :;are~ ]~ :*~ r pupp~ :@p.~ %" 3)
Here are three puppies.
nil
```

~ q This directive provides a means of calling functions from within *format* control strings. The name of the function should be *arg*. The called function can determine if the ":" or "@" modifiers have been used be referencing the global variables *colon-flag* and *atsign-flag*. Arguments may be passed to the called function with the **v** modifier. The *value* returned by the function is ignored by *format* (i.e., not printed).

```
=> (defun test (arg)
     (printout t "The value of atsign-flag is: " atsign-flag t)
     (printout t "The value of colon-flag is: " colon-flag t)
     (printout t "The value of arg is: " arg t))
test

=> (format t "~ @q" "dummy" 'test)
The value of atsign-flag is: t
The value of colon-flag is: nil
The value of arg is: dummy
nil
```

5.3.1.1. argument processing

The following directives control the application of arguments to the control string. Normally, the arguments are applied in order, according to the requirements of the directives. In circumstances, it is desirable to alter that order, and these directives provide a means of doing so. We have alread seen, in the last example, a way of applying a single argument to more than one directive using the ~ * directive. The complete list of directives includes the following.

~ * Causes the current argument to be ignored. The form ~ *n** causes the next *n* arguments to be ignored. The last argument processed is ~ :*; the *n*th previous argument is ~ *n*:*. This directive is used for processing of arguments relative to the current argument; to use absolute argument addressing use ~ *n* **g**. The effect of this directive in iterative operations (described below) is described in the next section.

~ *n* g
Causes the *n*th argument to become the current argument. All other argument references will be relative to argument *n*. Arguments are *zero* indexed, therefore ~ **0g** will go to the first argument. The use of this directive in iterative

operations is detailed, below.

5.3.1.2. conditional evaluation and iterative forms

The following directives allow the user to specify alternate control strings which can be evaluated in a particular sequence or on a conditional basis. In general, the form of these directives is a block of text beginning with one of ~ [, ~ {, or ~ < and ending with a corrsponding directive (~], ~ }, or ~ >). *Clauses* or *items* in the block are demarcated by the separator directive ~ ;. In some instances a *default* control string can be included as the last element of the block and prefixed by a special form of the clause separator: ~ :;. In most cases, if no default is specified and none of the conditions are satisfied, no alternative is printed.

5.3.1.2.1. alternate strings

~ [*str0*~ ;*str1*~ ;...~ ;*strn*~]

The ~ [...~] directive contains a *zero*-ordered list of alternatives, separated by the ~ ; directive. The single argument, *arg*, is the *index* of the list ("0" prints the first element, "1" the second, and so on). Unless a default is given (using ~ :;), no alternative is printed if the value for *arg* is out of range. The ~ **p** example illustrates a simple use of ~ [...~].

~ :[*false*~ ;*true*~]

This directive will select the *false* string if *arg* evaluates to *nil*, else it will select the *true* string.

~ @[*true*~]

An alternative to the previous directive; the control string *true* is only used if *arg* is non-*nil*. If the value of *arg* is *nil*, it is discarded and the argument pointer is advanced to the next argument. If *arg* evaluates to *t*, then the pointer is not advanced and *arg* will be the current argument for the next directive.

~ [~ *tag01,tag02,...;str0*~ *tag11,tag12,...;str1...*~]

[The most complicated of the alternative string directives†. The *tagn* strings are comma separated lists of values which may serve as a tag for the corresponding *str*. If *arg* is *eq* to one of the *tags*, the corresponding *str* is used. If a colon is used in the separator character (e.g., ~ *tag00,tag01*:;*str0*), then the tags list should be a list of *pairs* of values, representing a *range* of values for arg. For example, ~ [~ '0,'9:;numeral~] will use the string "numeral" if the value for *arg* is between "0" and "9".]

5.3.1.2.2. iteration strings

In certain circumstances, *format* may be called, repeatedly, to process data which is in the same form, for example, the elements of an association list or property list. To illustrate, let us assume that we wanted a formatted way of listing the elements of the property list of some symbol. We might try the following:

---

†Note, for example, that the syntax of the separator directive is slightly different from that in the previous cases. In the current example, the *tag* is actually part of the separator, and so, the *tilde* **precedes** the *tag*, which is followed either by a colon, or a colon-semicolon pair.

```
=> (putprop 'Clyde 'elephant 'type)
elephant
=> (putprop 'Clyde 'grey 'color)
grey
=> (putprop 'Clyde 'African 'subtype)
African
=> (plist 'Clyde)
(subtype African color grey type ...)
=> (format t "Symbol:~ a~ %~ { property:~ 10s value:~ 10s~ %~ }" 'Clyde
(plist 'Clyde))
Symbol: Clyde
 property: subtype    value: African
 property: color      value: grey
 property: type       value: elephant
nil
=>
```

~ {*str*~ }

This directive demarcates the iteration construct which takes one argument, *arg*, a list. The control string is reiterated until all of the elements of the list have been used as arguments to the control string. The number of directives requiring arguments in the control string will be the number of elements fetched from the list argument in a single iteration. If the control string requires *no* arguments, *format* will reiterate *ad infinitum*. Prior to each iteration, the number of remaining elements is checked; iteration is terminated if the list is null.

The ~ {*str*~ } directive can accept both a prefix and a postfix value. ~ *n*{*str*~ } guarantees that the iteration will occur no more than *n* times. If the directive is terminated with colon form, ~ :}, the iteration will occur *at least* once, even if the list is null (however a prefix of "0" will override this and iteration will not occur).

~ :{*str*~ }

Instead of a list, the argument should be a list of sublists (for example, an *assoc* list). On each iteration the *next* sublist will be used, even if there are elements remaining in the current sublist

```
=> (setq listpos '((car 1)(cadr 2)(caddr 3)(cadddr 4)))
((car 1) (cadr 2) (caadr 3) (caaadr 4))
=> (format t "~ :{function:~ 10a position:~ 10a~ %~ }" listpos)
function: car       position: 1
function: cadr      position: 2
function: caddr     position: 3
function: cadddr    position: 4
nil
=>
```

~ @{*str*~ }
Similar to the the forms described, above, but instead of recursively processing the current argument, this form processes the list of all of the remaining arguments, recursively.

~ @:{*str*~ }
A combination of the previous two forms; all of the remaining arguments are processed and each one should be a list which will be used as the argument to *str* on that iteration.

```
=> (format t "Ships spotted at: ~ @:{ <~ d , ~ d>~ } " '(2 2) '(4 8) '(3 5)
Ships spotted at: <2 , 2> <4 , 8> <3 , 5> <7 , 7> nil
=>
```

~ {~ }
If the directive does not contain a control string, the next argument is taken to be the control string. Arguments following the string argument are processed by the iteration. Evaluation of the control string occurs prior to the first iteration.

~ <...~ >
[~ *mincolw,colincr,minpad,padchar*<*text*~ > causes *text* to be justified within a field which is, at least *mincolw* wide. If the column width given would be insufficient to print *text*, *mincolw* is adjusted, upward, by *colincr*. The remaining prefixes are explained, elsewhere. The number of columns is determined by the number of fields in *text*; fields are separated by ~ ;. If no prefixes appear, the leftmost field is left-justified; the rightmost field is right justified (if there is only one field it is right justified). The : modifier causes spacing to be added before the first field is printed; the @ causes spaces to be added to the last field.

If the first clause of the ~ <...~ > directive is terminated with a ~ :; a special evaluation occurs in which the remaining forms are first evaluated and the the form of the output determined. Then, if the form of the padded text will fit on the current line according to the pre-determined padding, it will be output and the first clause discarded, else, the first clause will be output first, followed by the padded text. For example, to output a list so that the the each line after the first is indented 5 spaces and no element runs over the current column boundaries we can use the string (note the use of the "5" prefix†):

```
"~ %;; ~ {~ <~ %;; ~ 5:; ~ s~ >~ ^,~ }.~ %"
```

~ ^ In all of the cases described so far, an error will be caused if *format* runs out of arguments. The ~ ^ directive causes the current evaluation to be halted if there are no more arguments to be processed. Within one of iterative and alternate clauses it will cause the entire clause to be aborted, without error. If a prefix parameter is given, then the iteration is stopped if the prefix is zero (hence ~ ^ is equivalent to ~ #^). If two parameters are given, the iteration is stopped if they are *equal*, for example, ~ **1,#**^ will

†A second prefix can be given which will become the maximum line length.

halt iteration if only one argument remains. If three prefixes are specified, termination occurs if the second is between the first and the third in order of ascendency. Both **#** and **v** are acceptable prefixes.

When **~ ^** is used within a **~ :{** directive, only the *current* iteration is halted; remaining iterations begin with the next argument. To escape from the entire construct, use **~ :^**.

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 except for port or hunk.

RETURNS: x_pages.

SIDE EFFECT: FRANZ LISP attempts to allocate x_pages of type s_type. If there are fewer than x_pages of memory available for allocation, no space is allocated and an error occurs. The storage that is allocated is not given to the caller. Rather, it is added to the free storage list of s_type. By contrast, the functions *segment* and *small-segment* allocate blocks of storage and return it to the caller.

NOTE: You cannot allocate additional ports, and although you can allocate more hunk space, you must specify which size, e.g. you can allocate more hunk0, hunk1, ..., hunk6 space. (Hunkn holds up to 2^(n+1) items).

(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 for compatibility with 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**)**
(gcbefore s_type**)**

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

NOTE: The function *gcbefore* is called before the execution of the garbage collector when space of type s_type is exhausted. The flag *gcdisable* will be bound to t for the duration of the call. The argument to *gcafter* which is called by the garbage collector after the garbage collection is the same data type s_type. Usually the function *gcafter* 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.

(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).

On the Tektronix 4404 there are two shells: /bin/shell and /bin/script. If the user assigns a value to the operating system's *LispSHELL* environment variable, then that value will be used for the 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 communicating 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 (100*ths* of a second on the Tektronix 4404). 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 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).* or with *feature-present* described below.

(feature-present 'g_exp**)**

RETURNS: T or nil depending upon the evaluation of the g_exp in the context of the features present in the Lisp.

NOTE: Feature-present is used as follows:

```
(feature-present 'f)              [ == (memq 'f (status features)) ]
(feature-present '(not f1))
(feature-present '(and f1 f2))
(feature-present '(or f1 f2))
```

g_exp evaluation is calculated as follows. If the feature *fi* is on the (status features) list, replace it with a non-nil value otherwise with nil. Then perform the specified operations ('and', 'not', or 'or'), returning the value.

EXAMPLE:

```
=> (status features)
(long-filenames mvr hasht |68k| string Franz franz)
=> (feature-present 'mvr)
(mvr hasht |68k| string Franz franz
=> (feature-present '(not mvr))
nil
=> (feature-present '(or vax 68k))
t
=> (feature-present '(and mvr hasht))
t
```

(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 42.12"

(sys:chmod 'st_filename 'xl_mode)

WHERE: st_filename is the name of the file whose mode will be set and xl_mode is the mode to set.

RETURNS: t if successful. If unsuccessful (e.g. if the user doesn't have the required permissions or if the file doesn't exist), then an error is signalled.

NOTE: xl_mode can be a fixnum representing the mode or it can be a list of symbolic mode descriptors. Use of a fixnum mode is discouraged as it is operating system dependent.

Only the low 7 bits of the fixnum are important. The bits have this meaning:

1	user read permission
2	user write permission
4	user execute permission
8	other read permission
16	other write permission
32	other execute permission
64	other execute permission
128	set user id upon execution

In order to set a combination of permissions you add the individual permission numbers. For example to get user read, user write and other read permission: 1+2+8 = 11.

An alternative to the fixnum mode is a list of symbolic modes, each symbolic mode describing one or more permission bits.

Each symbolic mode is a string or symbol whose name is s or has the form [uoa]=[rwx]. The brackets are meta-syntax denoting the fact that one or more of the

enclosing characters can be present. The brackets themselves are not typed. The meaning of the symbolic mode 's' is that the set user id bit should be set. In [uoa]=[rwx], the 'u' and 'o' stand for user and other, and 'a' stands for all, meaning both 'u' and 'o' (on operating system which support group permission, the 'g' option is provided and 'a' also includes group permissions). 'r' stands for read permission, 'w' for write permission and 'x' for execute permission. To specify a symbolic bit, you must specify both a class (user, other and/or all) and a set of bits (read, write and/or execute). The symbolic mode is a list of these symbolic mode descriptors.

If the global symbol 'tilde-expansion' is non-nil, then the filename will be expanded to include the path-name to the conventional directory.

EXAMPLE: To set the mode of the file foo to be read/write by the user who owns the file, but only readable by others, use the following:
(sys:chmod 'foo '(u=rw o=r))
or
(sys:chmod 'foo '(u=r u=w o=r))
or
(sys:chmod 'foo '(a=r u=w)).

(sys:access 'st_filename 'x_mode**)**

RETURNS: True if the file has the given mode.

(sys:getpid)
(sys:link 'st_oldfilename 'st_newfilename**)**
(sys:time)
(sys:unlink 'st_filename**)**

NOTE: This set of functions is intended to provide symbolic access to various operating system features in a reasonably portable way. Tilde-expansion of file-names is performed on all filename arguments if the global variable "tilde-expansion" is non-nil.

(sys:getuid)

RETURNS: the user identification number of the user running lisp.

NOTE: This number uniquely identifies the user on a machine.

(sys:getpwuid 'x_uid**)**

RETURNS: a vector containing the information from the password file on the user whose id is x_uid. It returns nil if no user with that id exists.

The vector has the following form: index 0: the user's name (a string) index 1: the user id (a fixnum) index 2: the group id (a fixnum; nil if this notion is unimplemented). index 3: the user's login directory (a string).

(sys:getpwnam 'st_name)

RETURNS: the same information as sys:getpwuid but it searches for a given entry in the password file using the user's name, not the user's id.

(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 should be used rather than *(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 invoked.

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

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

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

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

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

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

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 raw readtable:none
csingle-splicing-macro standard readtable:none
escape-when-unique

vinfix-macro raw readtable:none
cinfix-macro standard readtable:none
escape-always
separator

vsingle-infix-macro raw readtable:none
csingle-infix-macro standard readtable:none
escape-when-unique

villegal raw readtable:^@-^G^N-^Z^\-^_rubout
cillegal standard readtable:^@-^G^N-^Z^\-^_rubout
escape-always
separator

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 *tconc* 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. If s_synclass is *macro* or *splicing*, then ls_func is the associated function.

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

Program Forms

8.1. Valid Function Objects

There are many different program forms in FRANZ LISP. These objects can occupy the function field of a symbol object and be "called" by other functions. The traditional forms in earlier versions of FRANZ LISP have been augmented by the Common-Lisp-required modified lambda expressions, and by closures. Various constructions on top of these (such as Flavors, described in chapter 19) are also useful. Table 8.1 at the end of this chapter shows all of the possibilities, how to recognize them, and where to look for documentation.

8.2. Functions

The basic technique for defining a function in Lisp is to use *def* or *defun* and use normal lambda binding. For example

```
(def foo (lambda(x y) ...))
  or equivalently
(defun foo (x y) ...)
```

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.

FRANZ LISP provides an nlambda function-type which is often used for functions that are invoked at top level, and give the program more control over evaluation of arguments. Some built-in functions which evaluate their arguments in special ways are defined in FRANZ LISP as nlambdas (for example *cond*, *do*, and *or*). When an nlambda function is called, the list of **un**evaluated arguments is lambda-bound to the single formal parameter of the nlambda function. For example

---

```
(def foo (nlambda (l) l)).
=> (foo a b c)
(a b c)
```

FRANZ LISP also supplies lexprs, which allow for a arbitrary number of evaluated arguments. 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. For example,

```
(def foo (lexpr (nargs) (list nargs (arg 1) (arg 2) (arg 3)))
=> (foo 'a 'b 'c)
(3 a b c)
```

When a function is compiled, *special* declarations may be needed to preserve the behavior of functions using its lambda-bound arguments as "free" or "global" variables. It is important in the case of such access to dynamic variable to declare the corresponding formal parameter *special* (see §12.3.2.2).

Lambda and lexpr functions both compile into a binary objects which have the same formal calling discipline, lambda.

In order to provide compatibility with Common Lisp functions, it is possible to use the expanded lambda-expression in a function definition as indicated below:

```
(defun function-name ({req-var}*
    [&optional {opt-var | (opt-var [initform [opt-svar]])}*]
    [&rest rest-var]
    [&key {key-var | ({key-var | (keyword key-var)} [initform [key-svar]])}*
    [&allow-other-keys]]
    [&aux {aux-var | (aux-var [initform])}*])
{form}*)
```

This admittedly daunting specification follows Common Lisp, and the description given in G. L. Steele's *Common Lisp the Language* (Digital Press, 1984) [referred to as CLTL]. On a first reading, you may prefer to skip to the examples at the end of this section. before embarking on the detailed explanation which follows immediately, as is based on CLTL.

Each element of the formal parameter specification list, or the lambda-list (that is, the list following the function name above) is either a *parameter specifier* or a lambda-list *signifier*. Lambda-list signifiers begin with the ampersand character (&).

In all cases a var or svar must be a symbol, the name of a variable; each keyword must be a keyword symbol which begins with a colon, such as :start. An initform may be any form.

A lambda-list has five parts, any or all of which may be empty:

Specifiers for the *required* parameters: These are all the names (req-vars) up to the first lambda-list signifier; if there is no lambda-list signifier, then all the specifiers are for required parameters.

Specifiers for &optional parameters: If the lambda-list signifier &optional is present, the optional names are those following the lambda-list signifier &optional up to the next lambda-list signifier or the end of the list.

A specifier for a &rest parameter: The lambda-list signifier &rest, if present, must be followed by a single name (rest-var), which in turn may be followed by another lambda-list signifier or the end of the lambda-list.

Specifiers for &key parameters: If the lambda-list signifier &key is present, all names (key-vars) up to the next lambda-list signifier or the end of the list are &key names. The keyword names may optionally be followed by the lambda-list signifier &allow-other-keys.

Specifiers for &aux variables: These are not really parameters. If the lambda-list signifier &aux is present, all names (aux-vars) after it are *auxiliary variables.*

When the function represented by the lambda-expression is applied to arguments, the arguments and parameters are processed in order from left to right. In the simplest and most usual case, only required parameters are present in the lambda-list; each is specified simply by a name *req-var* for the parameter variable. When the function is applied, there must be exactly as many arguments as there are parameters, and each parameter is bound to one argument.

In the more general case, if there are *n* required parameters (*n* may be zero), there must be at least *n* arguments, and the required parameters are bound to the first *n* arguments. The other parameters are then processed using any remaining arguments.

If &optional parameters are specified, then each one is processed as follows. If any unprocessed arguments remain, then the parameter variable *opt-var* is bound to the next remaining argument, just as for a required parameter. If no arguments remain, however, then the *initform* part of the parameter specifier is evaluated, and *opt-var* is bound to the resulting value (or to nil if no *initform* appears in the parameter specifier). If another variable name *opt-svar* appears in the specifier, it is bound to t if an argument was available, and to nil if no argument remained (and therefore *initform* had to be evaluated). The variable *opt-svar* is called a *supplied-p* parameter; it is bound not to an argument but to a value indicating whether or not an argument had been supplied for its associated parameter.

After all &optional parameter specifiers have been processed, then there may or may not be a &rest parameter. If there is a &rest parameter, it is bound to a list of all as-yet-unprocessed arguments. (If no unprocessed arguments remain, the &rest parameter is bound to nil.) If there is no &rest parameter and there are no &key parameters, then there should be no unprocessed arguments (it is an error if there are).

Next, any &key parameters are processed. For this purpose the same arguments are processed that would be made into a list for a &rest parameter. (Indeed, it is permitted to specify both &rest and &key. In this case the remaining arguments are used for both purposes; that is, all remaining arguments are made into a list for the &rest parameter, and are also processed for the &key parameters. This is the only situation in which an argument is used in the processing of more than one parameter specifier.) If &key is specified, there must remain an even number of arguments; these are considered as pairs, the first argument in each pair being interpreted as a keyword name and the second as the corresponding value. It is an error for the first object of each pair to be anything but a keyword.

In each keyword parameter specifier must be a name *key-var* for the parameter variable. If an explicit *keyword* is specified, then that is the keyword name for the parameter. Otherwise the name *key-var* serves to indicate the keyword name, in that a keyword with the same name (in the keyword package) is used as the keyword. Thus

```
(defun foo (&key radix (type 'integer)) ...)
```

means exactly the same as

```
(defun foo (&key ((:radix radix)) ((:type type) 'integer)) ...)
```

The keyword parameter specifiers are, like all parameter specifiers, effectively processed from left to right. For each keyword parameter specifier, if there is an argument pair whose keyword name matches that specifier's keyword name (that is, the names are *eq*),

then the parameter variable for that specifier is bound to the second item (the value) of that argument pair. If more than one such argument pair matches, it is not an error; the leftmost argument pair is used. If no such argument pair exists, then the *initform* for that specifier is evaluated and the parameter variable is bound to that value (or to nil if no *initform* was specified). The variable *key-svar* is treated as for ordinary *optional* parameters: it is bound to t if there was a matching argument pair, and to nil otherwise.

It is an error if an argument pair has a keyword name not matched by any parameter specifier, unless at least one of the following two conditions is met:

- &allow-other-keys was specified in the lambda-list.

- Among the keyword argument pairs is a pair whose keyword is :allow-other-keys and whose value is not nil.

If either condition occurs, then it is not an error for an argument pair to match no parameter specified, and the argument pair is simply ignored (but such an argument pair is accessible through the &rest parameter if one was specified). The purpose of these mechanisms is to allow sharing of argument lists among several functions and to allow either the caller or the called function to specify that such sharing may be taking place.

After all parameter specifiers have been processed, the auxiliary variable specifiers (those following the lambda-list keyword &aux) are processed from left to right. For each one, the *initform* is evaluated and the variable *aux-var* bound to that value (or to nil if no *initform* was specified). Nothing can be done with &aux variables that cannot be done with the special form *let**, and the following two forms are equivalent:

```
(defun foo (x y &aux (a 1) (b 2) c) ...)
  and
(defun foo (x y) (let* ((a 1) (b 2) c) ...)).
```

Whenever any *initform* is evaluated for any parameter specifier, that form may refer to any parameter variable to the left of the specifier in which the *initform* appears, including any supplied-p variables, and may rely on the fact that no other parameter variable has yet been bound (including its own parameter variable).

Once the lambda-list has been processed, the forms in the body of the lambda-expression are executed. These forms may refer to the arguments to the function by using the names of the parameters. On exit from the function, either by a normal return of the function's value(s) or by a non-local exit, the parameter bindings are no longer in effect. (The bindings are not necessarily permanently discarded, for a binding can later be reinstated if a *closure* over that binding was created and saved before the exit occurred).

Examples of &optional and &rest parameters:

```
(defun baz (x &optional (y 2)) (+ x (* y 3)))
(baz 4 5) ==> 19
(baz 4) ==> 10
(defun foo (&optional (x 2 y) (z 3 w) &rest q) (list x y z w q))
(foo) ==> (2 nil 3 nil nil)
(foo 6) ==> (6 t 3 nil nil)
(foo 6 3) ==> (6 t 3 t nil)
(foo 6 3 8) ==> (6 t 3 t (8))
(foo 6 3 8 9 10 11) ==> (6 t 3 t (8 9 10 11))
```

Examples of &key parameters:

```
(defun fum (u &key c d) (list u c d))
(fum 1) ==> (1 nil nil)
```

(fum 1 :c 6) ==> (1 6 nil)
(fum 1 :d 8) ==> (1 nil 8)
(fum 1 :c 6 :d 8) ==> (1 6 8)
(fum 1 :d 8 :c 6) ==> (1 6 8)
(fum :u :d 8 :c 6) ==> (:u 6 8)
(fum :u :c :d) ==> (:u :d nil)

Examples of mixtures:

(defun fie (a &optional (b 3) &rest x &key c (d a)) (list a b c d x))
(fie 1) ==> (1 3 nil 1 nil)
(fie 1 2) ==> (1 2 nil 1 nil)
(fie :c 7) ==> (:c 7 nil :c nil)
(fie 1 6 :c 7) ==> (1 6 7 1 (:c 7))
(fie 1 6 :d 8) ==> (1 6 nil 8 (:d 8))
(fie 1 6 :d 8 :c 9 :d 10) ==> (1 6 9 8 (:d 8 :c 9 :d 10))

All symbols whose names begin with & are conventionally reserved for use as lambda-list signifiers and should not be used as variable names. (Note: In CLTL, the phrase "lambda-list keyword" is confusingly used for signifier.)

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-definition facilities. The FRANZ LISP language's macro facility can be used to incorporate popular features of other languages into Lisp. For example, there are macro packages that allow you to create records (as in Pascal) and refer to elements of those records by the field names. The *struct* and *defstruct* packages imported from Maclisp does this. Another popular use for macros is to create more readable control structures which expand into *cond*, *or*, and *and*.

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's 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. protection against re-evaluation

Defmacro in FRANZ LISP takes a &protect option which allows the programmer to declare that one or more of the arguments should be protected against re-evaluation.

Consider, for example:
(defmacro printtwice (x) '(progn (print ,x) (print ,x))).

If we execute (printtwice "foo") It would print "foo""foo", probably just what we intended, since it expands to
(progn (print "foo")(print "foo"))

Something more mysterious happens if we execute
(setq i 0)
(printtwice (setq i (1+ i))). The expression printed is 12 instead of 11. In this case the printtwice macro expands to
(progn (print (setq i (1+ i))) (print (setq i (1+ i))))

which increments *i* twice.

Any argument to a macro may be protected against re-evaluation by using the &protect keyword as follows:
(defmacro printtwice (x &protect (x)) '(progn (print ,x) (print ,x)))

The macro expansion will then evaluate x once. If the evaluation of x could possibly have side effects (i.e. x is not an atom), the evaluation will be done once, and the value lambda-bound. This makes it simple to use a macro to replace a lambda. In compiled code there is no time penalty in using &protect if the protected arguments are atoms.

8.3.4. 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
```

8.3.5. sharp sign character macros

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.

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 forms are evaluated.

#,

#. The following form is evaluated at read time. See discussion below. read.

#\ If the form after the #\ is a one of *newline, space, rubout, page, tab, backspace, return, linefeed, vert, sharp, lpar, or rpar* 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. See discussion below.

#o

#O read the following form as an octal number.

#x

#X read the following form as a hexidecimal number.

#/c returns the fixnum representation of the character *c*.

8.3.5.1. 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

8.3.5.2. 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. Closures and Fclosures

A closure is a type of functional object useful for implementing advanced control structures and data access mechanisms. Its 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. This environment can then be re-used. Closures in FRANZ LISP provide the same functionality as closures in the MIT Lisp Machine Lisp (Zetalisp) design. Closures are also used as the implementation basis for *flavors* (described in chapter 19).

The form (**closure** *var-list function*), where *var-list* is a list of variables and *function* is any function, creates and returns a closure. When this closure is applied to some arguments, all of the value cells of the variables on *var-list* are saved away, and the value cells that those variables had *at the time* **closure** *was called* are made to be the value cells of the symbols. Then *function* is applied to the arguments. (This paragraph is somewhat complex, but it completely describes the operation of closures)

Fclosures, a simplified version of closures may be available in your version of FRANZ LISP. In later releases, fclosures are preserved (although with a different and more efficient underlying implementation strategy) for compatibility.

Fclosures are related to closures in this way:
(fclosure '(a b) 'foo) <==> (let ((a a) (b b)) (closure '(a b) 'foo))

We proceed with an example in the next section.

8.4.1. an example

```
++ lisp
Franz Lisp, Opus 42.03
=>(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 (closure '(x) 'code))
closure
=> (tester z 3)
(in 0)(in 1)(in 2)(out 3)(out 3)(out 3)
nil
=>x
0
```

The function *closure* creates a new object called a closure, containing a functional object including an environment in which variables (symbols) and the values exist. In the given example, the closure functional object is the function 'code'. The environment set of symbols and values contains the symbol 'x' and 0, the value of 'x' when the closure was created.

When a closure is funcall'ed:

1) The Lisp system lambda-binds the symbols in the closure to their values in the closure.
2) It continues the funcall on the functional object of the closure.
3) Finally, it un-lambda binds the symbols in the closure and at the same time stores the current values of the symbols in the closure.

Notice that the closure is saving the value of the symbol 'x'. Each time a closure is created, new space is allocated for saving the values of the symbols. Thus, if you execute closure again, over the same function, you can have two independent counters:

```
=> (setq zz (closure '(x) 'code))
closure
=> (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

(closure 'l_vars 'g_funcobj**)**

RETURNS: a closure.

WHERE: 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.)

NOTE: 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:

```
(closure '(a b) #'(lambda (x) (plus x (setq a (plus a 1)))))
```

Here, the #' construction is a special kind of quote which, if run through the compiler indicates that this should be compiled as a function rather than a quoted symbolic expression.

Other functions imported from Lisp Machine Lisp are

(symeval-in-closure 'cl_a 's_x**)**

RETURNS: the binding of the symbol s_x in the closure cl_a. You can look around inside a closure with this.

(set-in-closure 'cl_a 's_symbol 'g_x**)**
This sets the binding of s_symbol in the environment of the closure cl_a to g_x.

(let-closed argument-list function-body**)**
This is best described by an example:

EXAMPLE: (let-closed ((a 5) b (c 'x)) (function (lambda() ...)))
expands into
(let ((a 5) b (c 'x))
(closure '(a b c) function (lambda() ...)))))

8.5. Functional Objects table

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
closure	vector with *vprop* *eq* to closure	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"	18.
foreign function	binary with discipline of "function"	18
foreign integer function	binary with discipline of "integer-function"	18
foreign real function	binary with discipline of "real-function"	18
foreign C function	binary with discipline of "c-function"	18
foreign double function	binary with discipline of "double-c-function"	18
foreign structure function	binary with discipline of "vector-c-function"	18
foreign void function	binary with discipline of "void-c-function"	18
array	array object	9
package	package	17

Table 8.1

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. The vector size, as reported by the *vsize* functions, return the number of lisp objects in the vector.

9.5.2. property For other lisp objects which are implemented using vectors, the property list is reserved for internal use. For example, a package object has a special property list, which indicates how it is to be printed. Normally, property lists for vectors follow the same rules as do symbols. Also, 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.

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 *tpl-err-tpl-fcn,* which is the default value of **ER%tpl.** Thus, when all other handlers have ignored an error, *tpl-err-tpl-fcn* takes over. It prints out the error message and goes into a read-eval-print loop. The other standard error handler is *tpl-err-all-fcn.* 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 *untrace*s 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 before the compiler encounters a call to them This can be done either via the declare directive, or will be done implicitly by the compiler when compiling the definition of an nlambda.

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 primary advantage of a local function is that its entry and exit protocol is simpler and faster. Short functions can be speed up considerably by declaring them localf's. Bacuse the local functions are not known outside the file, these functions can have the same names as functions defined in another file, without a name conflict. This can be a benefit or a hazard. Compiled local functions cannot be used from the functions funcall or apply.

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.

```
++ list 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 ".o", (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, it may be used to exit from lisp. This is ^D on many machines.

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-eofs
if bound to a fixnum, used as the number of end-of-files to read at the top level read-eval-print loop before exiting. The default value is 5. It is still true that the top level will exit is an eof is read and the input device in not a tty.

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

Advanced Structured Programming: Defstruct

14.1. Introduction

Chapter 2 described the basic data types of FRANZ LISP objects and the functions which access and manipulate them. In most advanced LISP applications, however, these simple types are used to create more complex forms which may combine or even nest these data types to form a *structure*. There are no built-in functions for manipulating these user-defined structures; they must be created by the user. However, this implementation of FRANZ LISP contains a number of facilities for assisting the user in the creation and maintenance of structures.

14.2. Setf and Defsetf

Consider, for a moment the list (call it *alist*):

(foo bar).

We refer to *foo* as the *car* of the list; *bar* is the *cadr*. The LISP function which returns the *car* of a list is the *function car*. In the above example, *foo* can be thought of either as the result of evaluating *(car alist)* or as *the car of alist* where *car of alist* is used as another *name* for *foo*.

Another way of looking at this is to think of each LISP data type as having a function which *accesses* it and a function which *modifies* it. However, the **function** which accesses a data type can also be thought of as the **name** of object which is accessed. The simplest case of this occurs in variable binding. For example:

(setq var 'value)

assigns the value of *var* to **point to** *value*. The *access* form is *var* itself, since typing "var" to the top-level will cause it to return *'value*. The name of the variable is the form by which it is accessed. It would be useful to be able to eliminate the need for a separate modifier function by using the access function to *modify* the data type as well. The function *setf* accomplishes this.

This chapter to the FRANZ LISP manual describes additions by contributors from MIT, Berkeley, and University of Pittsburgh.

(setf g_accessfn*1* 'g_val*1*)

WHERE: g_accessfn is the complete function call which would access the desired data and g_val is the value to which it will be assigned.

SIDE EFFECT: *setf* is a macro which replaces itself with a call to the proper update function.

RETURNS: whatever the value of the macro expansion returns.

In the simplest case, then, *setf* can be used to replace the function *setq*. But consider, for a moment, some other data types. Returning to the example of the list, above, we referred to the first element of the list as the *car* of the list. The function which would update this value is the function *rplaca*. But with *setf* we could use *car*, instead.

```
; we can use setf as we would setq:
=> (setf alist '(foo bar))
(foo bar)

; now, to replace the car of alist:
=> (setf (car alist) 'baz)
(baz bar)

; is it a miracle? no, not really:
=> (macroexpand '(setf (car alist) 'baz))
(rplaca alist 'baz)
```

Lest one think that *setf* is all done with magic, it should be pointed out that the objects which can be updated by *setf* must be predefined using the macro *defsetf*. For instance, to make the access function *car* known to *setf*, we must define it:

(defsetf car (e v) '(rplaca ,(cadr e) ,v))

(defsetf s_fname l_setfvars 'g_body)

RETURNS: a *lambda* form which describes the expansion of the *setf*.

WHERE: s_fname is the name of the *access* function, *l_setfvars* is a list of the arguments to the *setf* call, and g_body which is the body of the *lambda* expression to be created.

SIDE EFFECT: *Defsetf* is a macro whose argument list is in the same form as *defun*[†]. The only substantial difference between *defsetf* and *defun* is that in the case of *defsetf* the *lambda* form is not defined at the top-level[‡], but rather as the value of the property *setf-expand* on the the property list of s_fname.

In order to better understand the workings of *defsetf*, it is useful to examine the implementation of *setf*. The following example illustrates some of the "inner workings" of the *setf* evaluation:

[†]The defining forms of a *lambda* expression are described in Chapter 8 in the section on *defun*. Since *defsetf* is, essentially, a *defun*, all of the *defun* defining forms will work for *defsetf*.

[‡]Meaning the *lambda* expression created by *(defsetf s_fname l_vars 'g_body)* will not become the function binding of s_fname.

```
; as in the example, above

=> (defsetf car (arg1 arg2) '(rplaca ,(cadr arg1) ,arg2))
(lambda (arg1 arg2) (list 'rplaca (cadr arg1) arg2))

; now, if we examine the property list of car:
=> (pp-form (plist 'car))
(setf-expand (lambda (arg1 arg2)
             '(rplaca ,(cadr arg1) ,arg2)))
nil
```

In evaluating *(setf g_accessfn 'g_val)* the following algorithm is followed:

(1) If g_accessfn is atomic and a *symbol*, then g_accessfn is *setq*ed to g_val.

(2) If g_accessfn is non-atomic and the *car* of g_accessfn is a symbol, then the property list of g_accessfn is searched for the indicator *setf-expand* and the property value (presumably a *lambda* expression created by *defsetf) is then apply*'d to g_val.

(3) If g_accessfn is non-atomic and the *car* of g_accessfn is a symbol representing some combination of *car* and *cdr*, then an update function is created and applied to g_val†.

(4) If g_accessfn is non-atomic and the *car* of g_accessfn is a cmacro or macro, then the macro is expanded and *setf* reapplied to the result.

(5) If g_accessfn is non-atomic and the *car* of g_accessfn is a function, then *setf* is reapplied using the function binding of the *car* of g_accessfn.

(6) If no other condition is satisfies, *setf* g_accessfn is is continually reapplied (or expanded), until either *setf* quits or the situation is resolved.

For convenience, a number of *setf* macros have already been defined in FRANZ LISP. These are shown in the following table. In addition, *setf* is can be used to update structures defined by *defstruct*.

setf form	update form	object type
(setf (arg argnum) newval)	(setarg argnum newval)	nlambda
(setf (arraycall type arr key) newval)	(store arexp newval)	array
(setf (cxr key obj) newval)	(rplacx key obj newval)	hunk
(setf (get obj key) newval)	(putprop obj newval key)	property list
(setf (gethash obj key)newval)	(puthash obj newval key)	hash array
(setf (gethash-equal key obj) newval)	(puthash-equal obj newval key)	hash array
(setf (nth keyex list) newval)	(rplaca (nthcdr keyex list) newval)	list
(setf (nthcdr keyex list) newval)	(rplacd (nthcdr (sub1 keyex) list) newval)	list
(setf (nthelem keyex list) newval)	(rplaca (nthcdr (sub1 keyex) list) newval)	list
(setf (plist obj) newval)	(setplist obj newval)	property list
(setf (symeval obj) newval)	(set obj newval)	symbol
(setf (vref obj key) newval)	(vset obj key newval)	vector

†By implication (unless otherwise specified), there is no property *setf-expand* for most of the *car, cdr* combinations as these are "created" on the fly.

14.3. Defstruct

The notion of using an access function as an update function (as manifested in the *setf* macro) can be carried even further. It is not hard to imagine, for example, that the user may want to "nest" lists within lists and data forms within data forms to create more complex data structures. In such a complex structure it might be useful to name "components"[†] to provide an easy way to access the imbedded data forms. Furthermore, access and update functions would have to be created for each element of that structure.

Consider the following example: Suppose we wished to write a FRANZ LISP program that dealt with ships. Insofar as we are concerned, the relevant data regarding each ship is its *x* and *y* position in a two-dimensional co-ordinate system; its velocity in each co-ordinate, and its total mass. We could represent the information regarding this ship as a list of five elements, an array, a vector, or any other combination of LISP data types. But how easy would it be to manipulate this data? In reading over our programs would we remember that *(cadddr ship)* was the *velocity in direction y* of *ship*?

Instead, what we would like to see is some way of representing ship so that the names of the components of the ship provided a way to access and update ship information. That facility exits with *defstruct*.

```
(defstruct ship
        ship-x-position
        ship-y-position
        ship-x-velocity
        ship-y-velocity
        ship-mass)
```

says that every *ship* has five components. For each of these five components there will be defined an *accessor function* which will return the value of that particular *component* which it names. For example, for any object (say *QE2*), which is an instance of *ship* the value of *ship-x-position* for *QE2* will be:

```
(ship-x-position QE2)
```

Furthermore, *defstruct* will define a macro, *make-ship*, which can be used to create new instances of ship. This macro is known as a *constructor macro* since it constructs new structures of type *ship*.

Finally, as we might have expected from the previous section, structures defined by *defstruct* can be updated using the *setf* macro, e.g.:

```
(setf (ship-x-position QE2) 50)
```

will set the value of the *ship-x-position* of *QE2* to be *50*.

[†]The notion of *structures* such as those we will be talking about has been described for many languages (in one variety or another), so that it becomes impossible to find "neutral" terms to describe the form of these structures. For example: the term *record* has meaning to both PASCAL and INTERLISP users, though it would not describe exactly the same thing to both. Similarly, we will refer to *structures* in FRANZ LISP although the name is familiar to PL/1 and C users, as well. In the present case, the term *components* is used to describe what might also be called *fields*, *elements*, or *slots*, principally because that term has precedent in other texts [Weinreb and Moon; Steele]. The user is cautioned to consider these structures in FRANZ LISP as being *similar to* rather than *equivalent to* structures in other languages.

(**defstruct** sl_nameargs sl_slotdescript*1* [...sl_slotdescript*i*])

RETURNS: the name of the structure created.

SIDE EFFECT: creates a structure and the corresponding accessor and constructor functions for that structure.

WHERE: the arguments to *defstruct* correspond to the forms illustrated, below.

sl_nameargs: In the simplest case, sl_nameargs can be a symbol which will be the *name* of the structure created. This was illustrated by the case of the ship, above. In a more complex fashion, sl_namelist can be a list containing the structure name and a list of keywords. In most cases, the keywords may have an argument. When a keyword-argument pair is given, it should be in the form *(keyword argument)*, unless indicated, otherwise. In FRANZ LISP, *the argument is not evaluated.* The following keywords have been defined for FRANZ LISP.

:conc-name:

This argument can have two forms, both of which provide automatic prefixing of access function names. When *:conc-name* is not given an argument, the name of the structure is concatenated to the component name with an intervening hyphen. For example:

(defstruct (person :conc-name) name age sex)

will create a structure *person*, with the **slots** *name, age* and *sex*, and the **access functions** named *person-name, person-age,* and *person-sex.* On the other hand, when given an argument, expressions like:

(defstruct (person (:conc-name human)) name age sex)

will create a structure *person* with the access functions *humanname, humanage,* and *humansex.* (Notice the absence of the hypen in this case.)

:named

Means that, where possible, the structure created will be a *named* type (*i.e.*, if the *type* would have been a *:list*, it will be a *:named-list*, instead). In addition, certain functions exist which operate on named structures.

:include

The *:include* option can be used to build new structures which are composites of other structures. The structure so defined will have the all of the slots of the *:included* structure in addition to to the newly defined slots. Access functions which are defined for the *:included* structure will operate for the structure in which it is included but the converse is not true[†].

[†]This is also the case in ZetaLISP, but not the case in Common LISP, that is, in Common LISP, the structure *astronaut* which *:included* the structure *person* would have two access functions for the slots in *astronaut* which were also in *person.* That is to say, for *astronaut* as defined in the next example, there would be two accessor functions for the *name* slot: *person-name* and *astronaut-name.* This may, someday, be the case for FRANZ LISP as well.

Consider the following‡:

```
; suppose we wish to create a structure, astronaut.
; since astronauts are people, too
=> (defstruct (astronaut :conc-name (:include person))
    helmet-size
    (favorite-drink 'Tang))
astronaut

; now, to create an instance of astronaut
=> (setq Glenn (make-astronaut
    name    'John
    age     45
    sex     'male
    helmet-size    17.5))
#<astronaut 5>

=> (person-name Glenn)
John

; since person-name was inherited from the :included file
; astronaut-name is not defined
=> (astronaut-name Glenn)
Error: eval: Undefined function astronaut-name

=> (astronaut-helmet-size Glenn)
17.5
```

Notice that *:conc-name* in the structure was only passed to the structures uniquely defined within the *defstruct*. Those components which were *:included* inherited their accessor function names from their parent structures. In addition, if default values exist for the structure which is *:included*, these values are inherited by the new structure unless an alternate form is used, such as:

```
(defstruct (astronaut (:include person sex (age 45)
                      :conc-name)
                      helmet-size
                      (favorite-drink 'Tang)
```

which says that the default value for *sex* is *nil* and for *age* is *45*. As we can see from this example, the proper way to specify defaults values is either as a *list* whose *car* is the slot and whose *cadr* is the value for that slot or as a *symbol*, where the value defaults to *nil*.

Multiple *:includes* cannot be used. For example, suppose that structure "A" has 5 slots and structure "B" has three, the result of:

```
(defstruct (C (:include A)
              (:include B))
              slot1c slot2c slot 3c)
```

will be a structure whose first three slots come from "B" and whose last three slots are from "C"; no slots from "A" will be used. This "feature" is is present because of the fact that the slots of a structure are allocated starting from the beginning of that

structure[†]. Since the accessor functions of "B" cannot be changed by their subsequent use in another structure, the reference point for accessor functions is almost always the first element of the type in which that structure was implemented. If a second structure were included, it would, by necessity, be forced to start at the first slot of the new structure since the accessor functions were previously defined. For this reason, only the last structure is *:included* if *:include* is used more than once. If this is not entirely clear, read through the next section before coming back to this point.

In FRANZ LISP *:included* structures must be of the same *:type* as the aggregate structure which *:includes* them. This is further explained in the next paragraph.

:type

The *:type* option specifies which type of LISP object will be used to implement the structure. In FRANZ LISP the default type is a *named-vector*. As mentioned, above, the *:type* of an aggregate structure cannot be different from the *:type* of the structures which are *:included* in it. By implication, this means that structures cannot be composed of other structures of more than one type. The reason for this is clear: the accessor functions are defined at the time that the structure is created. Since these accessor functions are type dependent one cannot create a structure which would alter the type of a parent structure.

Structures which are *:named* are implemented as *named types*; by default, *:named-vectors* As always, aggregate structures can only *:include* structures of the same type, therefore, in the previous example, if *astronaut* is named so must be *person*. The types available in FRANZ LISP are described in the next section.

:constructor

This option may have, zero, one or two options. If the argument is not given or if *:constructor* is not given as an option to *defstruct*, then a *keyword driven* constructor is defined whose name is the symbol "make-" concatenated to the name of the structure defined (*e.g.*, **make-person** in the case of *person*). If one argument is given and is *nil*, no constructor will be defined. If one non-*nil* argument is given then the *keyword driven* constructor macro is named with that argument. If two arguments are given, the first must be a symbol (the name of the constructor), and the second a list of the slots initialized by that macro. In this last case, any initializations which are done in the constructor will override the initializations defined in the body of the *defstruct*. In contrast to the previous cases, *:constructor* with two arguments is *by-position* rather than *keyword driven*.

The use of *:constructor* will be illustrated in the next section.

:default-pointer

In the definition of *defstruct* we outlined the procedure for assigning *default values* to slots of the structure being defined. In other applications, we may wish to create a structure which, apart from the structure definition, can serve as a model or default case of that structre. One way to do this is through the *:default-pointer* option.

Under normal conditions, accessor functions such as *person-name* take one argument, the name of the instance we wish to reference. By using *:default-pointer*, the argument to the accessor function becomes optional. If it is not specified, the value returned by the accessor function is the the value of the slot for the instance pointed to by the argument to *:default-pointer*. To illustrate:

[†]Hence, the first slot in a structure of *:type :list* will be the *car* of the list.

```
=> (defstruct (apple :conc-name
  (:default-pointer *APPLE*))
  (type 'fruit) name color)
apple
=> (setq *APPLE* (make-apple color 'red))
#<apple 3>
=> (vector-dump *APPLE*)
size = 3, prop= apple
0: fruit
1: nil
2: red
nil
=> (setq crabapple (make-apple color 'green name 'crab))
#<apple 3>

; by default, the accessor functions will point to *APPLE*.
=> (apple-color)
red
=> (apple-color crabapple)
green
```

:size-symbol

This option allows a user to specify a global variable which will be bound to the *size*[†] of the structure being created. If no option is given the variable will be called "*struct*-**size**" where *struct* is the name of the structure. Otherwise the variable will be named by the argument to *:size-symbol.*

:size-macro

This option is, in every respect, equivalent to *:size-symbol* except that rather than creating a symbol, this option creates a macro which *expands* to the value of the size of the structure.

:initial-offset

This option *requires* one argument (a fixnum), which will tell *defstruct* to skip some number of slots before assigning the slot names. No accessor functions are created for these slots, so the use pof this option presupposes some knowledge about the implementation of defstruct on the part of the user. If the structure *:includes* another structure, then the order of slot assignments is: *:include -> :initial-offset -> defstruct.*

:callable-accessors

Normally in FRANZ LISP, acessor functions are not functions but macros[‡]. If the option *:callable-accessors* appears in the *defstruct* with an argument of *t* or no argument at all, the accessor functions are defined as lambda functions, rather than as macros.

NOTE: While this option provides accessors which can be used as conveniently as functions (i.e., they can be passed as arguments to *mapcar*), the accessor functions created **cannot** be used to update slots using *setf.*

:eval-when

This option behaves just as the compiler function *eval-when* and has the same arguments only in keyword form (preceded by a colon. ":"). By default structure-associated

[†]By *size* we are referring to the the number of *elements* at the top-level of the structure, therefore, the *size* of the list *(a (b c) d (e f g))* is *4.*

[‡]In contrast to accessor functions in ZetaLISP which are, by default, *subst*s, (functions).

functions are:

(:eval-when (:eval :compile :load))

:alterant

Normally when a structure is created a function is also defined which can be used to alter, *en masse* all of the slots in a particular instance of that structure. The name of this function[‡] is made by concatenating "**alter-**" to the structure name, (e.g., *alter-person*, in the example, above). The argument to *:alterant* should be either a symbol (which will be the name of the alterant instead of the default), or *nil*, which means no alterant function will be defined. Using *:alterant* without an argument is equivalent to not using it at all; in these cases the default function will be defined.

Alterant functions are described in the next section.

:but-first

In some instances, a structure will be defined which will *always* be part of another structure, so that the accessors of that structure must reference the structure of which it is a part. The *:but-first* option is a means of expressing this relationship. Consider the following example:

```
; using person as an example, we may wish to subdivide
; name into three more slots (of type :list)

=> (defstruct (name :list (:but-first person-name))
  firstname middlename lastname)
name

; and to create an instance of person called generic-person:
; (with a name more colorful than "John Q. Public)

=> (setq generic-person (make-person name (make-name))
#<person 3>

=> (alter-name generic-person firstname 'Rufus
          middlename 'Thaddeus lastname 'Firefly)
(Firefly)

; to alter the head field of body

=> (setf (middlename generic-person 'T.)
(T. Firefly)
=> (person-name generic-person
(Rufus T. Firefly)
```

The *:but-first* option is used to create accessor functions which apply their arguments to the result of another accessor function which was the argument to *:but-first*[†]. This option always takes one argument, and always defines a structure which will exist only within the context of another structure.

Notice how an instance of *person* is created in the example, above. The call to *make-person* includes the assignment of the *name* slot to a call to *make-name*. This procedure must be followed when using *:but-first* structures: *the substructures must be*

[‡]Actually, a macro.

[†]In fact, the argument to *:but-first* should **always** be the name of an accessor function.

created (using the make- functions), at the same time that the top-level structure is instantiated. To see what would happen if this procedure is not followed, imagine the case if one had defined *test-person* without including *make-name*:

```
=> (setq test-person (make-person))
#<person 3>

; note that there will be exactly one
; slot for the value of person-name

; since alter-name expects person-name to
; be a three element list:
=> (ater-name test-person
   firstname 'John middlename 'Q lastname 'Public)
Error: Attempt to rplac[ad] nil.
<1>:
```

:print

Structures which are of a type *:named-vector* are allowed an addition option not available to other *:types*, the *:print* option. Basically, *:print* allows the user to define a format for printing the structure, using the directives defined for the *format* output package (see Chapter 5 for further details). The form of the arguments to *:print* is:

(:print control-string &rest args)

where *args* can reference any of the slots within the structure. For example, using a modified form of our *ship* structure:

```
=> (defstruct
  (ship (:print "~ &A ship at position <~ s,~ s>~ &  with velocity [~ s,~ s]~ %"
    (ship-x-position ship)(ship-y-position ship)
    (ship-x-velocity ship)(ship-y-velocity ship)))
  (ship-x-position 0)
  (ship-y-position 0)
  (ship-x-velocity 0)
  (ship-y-velocity 0)
  ship-mass)
ship

; now, using make-ship, note the form of the output
=> (setq x (make-ship ship-x-position 10))
A ship at position <10,0>
  with velocity [0,0]

; if we modify some fields:
=> (setf (ship-y-velocity x) 25)
25
=> (setf (ship-x-velocity x) 'unknown)
unknown

; this time:
=> x
A ship at position <10,0>
  with velocity [unknown,25]
```

14.3.1. Using Defstruct

14.3.1.1. :types of structures

As was mentioned, previously, by default, structures are implemented as *named-vectors*. In the case of vectors, the property of the vector becomes the *name* of the structure[†].

The following types are available in FRANZ LISP. The user writing portable code should be cautious in specifying only those types common to all LISPs of interest.

:list Implements the structure as a simple list.

:named-list
Like *:list* but the first element of the list will be the name of the structure (therefore the length of a *:named-list* implementation of a structure will be one greater than the same structure implemented as a simple *:list*.

*:list**
Like *:list*, but the last *cons* of the list created is a *dotted pair*. There is no *:named-list**.

[†]If the structure is implemented as a list, the *name* is the *car*; if the structure is an array, the *name* is the first element, and so on.

:array
Implements the structure as an array.

:named-array
Like *:list* but the first element of the array will be the name of the structure. This type may or may not be available in your FRANZ LISP implementation.

:hunk
At one time the default in FRANZ LISP whose use is, now, discouraged. Use vectors instead. Also available as a:

:named-hunk
The *zeroth* element of the hunk is the name of the structure.

:tree This structure is implemented as a binary tree of *cons cells* with the leaves serving as the slots. The advantage of using a tree as opposed to a list is that the *access time* for any slot at some level of branching is the same as the access time for any other slot at that same level. A disadvantage of using *:tree* structures is that they may not include or be included by any other structure.

:fixnum
A fixnum structure is a structure of **one** slot which is implemented as a single fixnum which, itself, may contain a number of slots, corresponding to the bits which make up the fixnum. These can be particularly useful in those LISPs where fixnums are implemented in byte sizes of 15 or 20 since many commonly used numbers can be represented in less than 10 bytes and, therefore, more than one number can be "packed" into a single fixnum space. In FRANZ LISP fixnums are only 10 bits and the utility of this type is questionable.

:vector

:named-vectors
Structures are, by default, *named-vectors*. A named vector is called thus because it prints as #<*structure-name size*>. Vectors are described in Chapter 9.

14.3.1.2. :constructor functions

In the previous section we described a way to define the constructor function for a structure. This section describes the uses of *:constructor*.

When *defstruct* is applied without the *:constructor* option, a constructor function is automatically defined and named as the concatenation of "make-" to the structure name, (e.g, *make-person* in the case of the structure *person*). The constructor defined is a *keyword driven* function, that is, arguments to the function are preceeded by *keywords* which indicate which slots to fill. For example:

(setq fred (make-person name 'Fred age 22 sex 'male))

creates an instance of *person*, with the slots initialized according to the given keywords, *name*, *age*, and *sex*. In contrast, *:constructor* functions which are explicitly defined are *by-position* constructor functions, that is, values are assigned to the individual slots on the basis of their position in the function call. This is analogous to the scheme used to apply *lambda* functions in which lambda variables are assigned on the basis of the the position of the arguments to the function.

When explicitly defined, constructor provide a means for creating instantiations of a particular structure which can be tailored to the individual application. For example, in the case of the structure *person*, we know from experience that this category can be further subdivided according to certain criteria, some of which were

used as *slots* in the definition of *person*. On the basis of age and sex, we can further subdivide *person* into male, female, adult, and minor.

Consider, as an example, the following expanded version of *person*:

```
(defstruct (person :conc-name :named :constructor
        (:constructor create-female (&aux (sex 'female)))
        (:constructor create-male (&optional name age &aux (sex 'male)))
        (:constructor create-adult (&optional sex &aux (age '>21))))
        name age (sex 'unknown))
```

This says to define the named structure *person*, and in addition, define three functions for constructing certain instantiations of *person*, *create-male*, *create-female*, and *create-adult*. The default sex for a person so created is 'unknown. In addition, specific kinds of persons can be instantiated using one of the constructors. These have been defined, arbitrarily, to indicate the various ways that an argument list might appear; the argument list corresponds to a lambda list with a few minor variations:

- **&optional** arguments are given the value of the calling arguments, if they exist, else the default value for the arguments, if given in the form *(arg default)*, else the default value for the slot, as assigned by the*defstruct* (*unknown* for the component *sex* in the example above), else *nil*.
- **&aux** arguments are given the default value, when in the form *(arg default)*; when they are alone, the value of the arguments are *unassigned* (unbound). This differs from the case of *defun* where symbolic arguments to *&aux* are initialized to *nil*.

Notice that the arguments to the *:constructor* function are NOT named arbitrarily, but according to the slots to be filled. This is a requirement; using an argument name which is not a slot name will cause an error ("unknown slot to constructor").

(**make-***name* [*'s_slot1* 'g_val*1*...])

RETURNS: a structure of the form of *name*.

WHERE: the values for each slot is either given using a keyword argument or defined by a default value with *defstruct*.

EXAMPLE: see Example 2.20.

14.3.1.3. :alterant macros In most cases, updating slots in a structure can be done, adequately, with the *setf* function, but this can be cumbersome to use if we want to update many slots instead of just one. The keyword-driven *alterant* macro provides a tool for doing such a update.

---

‡This first occurrence of *:constructor* tells *defstruct* to create a *keyword defined* constructor function for *person* (which will be called *make-person*). Normally this would automatically be done by *defstruct* but by using the *:constructor* argument to create additional constructor functions we turn this feature "off" and force defstruct to look for explicitly declared constructors. If we had neglected to include this, the constructor *make-person* would not have been defined.

(alter-*name 's_inst ['s_slot1* 'g_val*1*...])

RETURNS: the structure s_inst with the given slots altered.

WHERE: s_slot is the slot as described in the defstruct **not** the slot as referenced by the accessor function[‡]
Each g_val is evaluated **before** any slot is changed, therefore, constructs such as the following are possible:

```
=> (defstruct (ship :hunk :conc-name)
     x-position
     y-position
     x-velocity
     y-velocity
     mass)
ship
=> (setq QE2 (make-ship x-position 33 y-position 12
              x-velocity 15 y-velocity 0 mass 500000))
{33 12 15 0 500000}

; now, to exchange the values for x-position and y-position

=> (alter-ship QE2 x-position (ship-y-position QE2)
                y-position (ship-x-position QE2))
{12 33 15 0 500000}
=>
```

14.3.1.4. Other caveats

Consider, for a moment, the following:

```
=> (defstruct box heighth length width)
```

This is an error. Recall that when a structure is created, accessor functions are defined to access the slots of the structure. In the case of the example, above, an accessor function, *length* was defined. It so happens that the function *length* already exists in FRANZ LISP and is used in some fairly significant ways. It is for this reason that the *:conc-name* option exists, since this provides a means for defining accessor functions which are specific for the structure defined.

[‡]For example, in *person* the slot is *name* and the accessor function *person name*.

CHAPTER 15

The Lisp Stepper and FIXIT

Note: this debugger (*step* and *debug*) has functionally been replaced by feeatures in the current top level, tpl, which is described in chapter 13 (see the tpl commands ?debug and ?step).

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 (foo1 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.

CHAPTER 17

Packages

17.1. Introduction

LISP systems have traditionally had "flat" name-spaces. That is, all symbols resided in a common pool, and could be used by any program. Although there were some techniques that could be used to restrict conflicts, large LISP systems with modules written by many different programmers benefit from a mechanism for the avoidance of accidental name collisions. FRANZ LISP promotes modular programming and addresses this name-space problem through its *package system.* The implementation in FRANZ LISP is designed to conform to that of the COMMON LISP design as given in *COMMON LISP the Language* by Guy Steele, (Digital Press, 1984), which we refer to as CLTL in this chapter.

The programmer who is used to the interactive nature of LISP program development should be aware that the design of the package system in COMMON LISP is based on a file-centered view of program development. Attempts to resolve the meaning of the multiple-name-space scoping rules in the midst of the run-time system and interpreter have resulted in heated debate (after the publication of CLTL). Nevertheless, packages provide a useful facility, especially for debugged programs. It is *possible* to get substantially tangled up if the programmer is defining and modifying several inter-related packages with common names while interactively debugging and editing.

A *package* is a vector of data which establishes a mapping from external names to internal symbols. The single current package is used by the LISP reader and printer in converting lexical strings into symbols. The value of the global variable **package** is bound to the current package. You can refer to symbols in other packages by the use of *package qualifiers.* A package qualifier is a string (the name of the home package for the symbol) used as a prefix to to the symbol name. Usually one colon is used to separate the prefix from the symbol name. For example, you can use the name *restaurant:order* to refer to the symbolic object *order* which is defined in the *restaurant* package, if, within that package, order is defined to be an **external** symbol. In some programming languages, any symbols not declared external would simply not be accessible from outside the defining package. In LISP, it is merely made inconvenient. If *order* were not external, but **internal**, it could be reference by the syntax *restaurant::order.* Any symbol is either **external** or **internal** relative to a package.

External symbols are advertised outside a package to be available for use by programs in other packages. Their names should be chosen to be unique and associated with the useful abstractions of the package module you are building. On the other hand, you should use internal symbols for implementation of abstractions which you wish to conceal from ordinary external users. By default all symbols are created as internal symbols. You explicitly use the *export* command to identify external symbols. An important consequence of this is that different packages can reuse the same internal name without conflict.

Each symbol is implemented with a *package cell* containing a pointer to its unique *home package* in which it is said to be *interned.* That is, its representation is internalized in the Lisp interpreter's symbol table. It is possible and sometimes useful to create *uninterned* symbols which have *nil* in their package cell.

In using packages, you have the option of creating a hierarchy of naming by inheritance. By using the functions *intern, import* and *export,* can be used to make a previously unaccessible symbols accessible in a package from other packages.

The function *unintern* makes a symbol inaccessible from a package. If the package was its home package, the symbol is said to be *uninterned.*

17.1.1. Consistency Rules

These package consistency rules hold true so long as you do not change the value of **package**:

- Read-read consistency: Reading the same string always results in the same (*eq*) symbol.
- Print-read consistency: An interned symbol always prints as a sequence of characters that, when read back in, yields the same (*eq*) symbol.
- Print-print consistency: If two interned symbols are not *eq*, then their printed representations will be different sequences of characters.

The rules may not hold if you change the value of **package**, or doing so implicitly by continuing execution from an error break, or if you call one of the "dangerous" functions described subsequently. These functions are *unintern, unexport, shadow, shadowing-import,* and *unuse-package.*

17.2. Package Names

When a package is created, it is given a character string name. Nicknames or shortened versions of the "official" name can also be assigned at that time. A number of functions described below provide mappings between these concepts (*find-package, package-name, package-nicknames, find-package, rename-package*). Any of the functions that require a package-name argument from the user accept either a symbol or a string. If the user supplies a symbol, its print name will be used. A package, if printed, will look like #<package "package name">.

17.3. Translating Strings to Symbols

At any time there is a current package object in effect. It is the value of the global symbol **package**.

The current package affects the way a string, read in by the FRANZ LISP reader, is interpreted as a symbol. Since the current package may also (and usually will) inherit names from other packages, these must be searched as well as the package currently in effect. If the string is associated with no symbol at all, a new symbol must be created and "interned" in the current package. The point of all this is to make sure that if the same string is later read in the same package context, then the the same symbol will be used.

When you wish to refer to an external symbol in a package other than the current package, you can use a *qualified name,* which is a concatenation of the package name, a colon (:) , and the name of the symbol. If you need to refer to an *internal* symbol of some package other than the current one, you can use a qualified name using a double- colon (::) instead of the usual single colon as the separator for the qualified name.

There is a distinguished package named *keyword* which contains all keyword symbols used by the LISP system itself and by user-written code. Because keyword symbols are

used so frequently and must be accessible from everywhere, there is a special reader syntax for them which omits the package name, leaving just the colon and the symbol. Furthermore, all symbols in the keyword package are external constants and are also implicitly quoted. For example, the symbol :foo is the same as keyword:foo, and has the value :foo. As implemented in FRANZ LISP, each symbol contains a package cell (accessed by the function symbol-package) that points to the home package of the symbol. Uninterned symbols have nil home package pointers.

Symbols are printed in a form which would allow them to be read-in and identified with themselves, assuming the same package is current. For example, keywords are printed with a preceding colon. Normally accessible symbols are printed without colon-qualifications. Symbols which are accessible only by internal or external qualification with respect to an accessible package are printed with the appropriate colons. An uninterned symbol is printed preceded by #:. When read in, such a string causes an uninterned symbol to be created.

17.4. Exporting and Importing Symbols

While the principal advantage of using packages is to allow for the creation of separate modules and name spaces, this would not be of much use without features for controlled inter-package access to names. Other than the use of the colon syntax for reference to external or internal symbols in packages, two other mechanisms are available: *import* and *use-package*.

The function *import* will not allow you to shadow (i.e. make inaccessible) a currently accessible distinct symbol either in the current package or available by inheritance. (If the same exact symbol is already present, then *import* has no effect.) If you really want to shadow a symbol you can use the function *shadowing-import*.

Use-package makes symbols accessible by inheritance, but does not import them. It checks for name conflicts between the newly imported symbols and those already accessible in the importing package. The function *unuse-package* undoes this.

FRANZ LISP uses the standard COMMON LISP prescriptions for resolution of name conflicts.

17.5. Built-in Packages

The following packages are built into every FRANZ LISP system, Opus 42 and later:
lisp: This package contains the standard user-visible functions and global variables that are present in the FRANZ LISP system. Unless it is the intention to hide the standard Lisp functionality from the user-level program, any package should use *lisp* so that these symbols will be accessible without qualification. The nickname *fl* is available for this package.

commonlisp: This package, with nickname *cli* contains the primitives of the COMMON LISP core-compatibility package. Its external symbols include the user-visible functions and global variables that are present in the COMMON LISP system, including functions which (because of the incompatible re-use of the same name) must shadow the default FRANZ LISP system. The user can access the original FRANZ LISP functions by using the *fl:* prefix. Other packages based on COMMON LISP should use *cli* so that these symbols will be accessible without qualification.

user: This package is the default current package at the time a LISP system is started. This includes the usual top-level interpreter and debugger in FRANZ LISP. In order to provide the usual Lisp environment, this package uses the *lisp* package.

keyword: All the symbols in this package are treated as constants that evaluate to themselves, so that the user can type *:test* instead of *':test*. All the keywords used by built-in or user-defined LISP functions are automatically placed in this package as external symbols, and are always represented with an initial colon.

editor: This package contains the resident editor and its subroutines. This package uses *lisp* and has the nickname *ed*. The editor and its package are loaded by the autoloading facility when certain of its external functions are invoked. See chapter 13 for details.

system: This package contains system-dependent hooks such as "getuid" (get user identification).

system-internals: (nickname *si*) This package contains programs which are FRANZ LISP system internal primitives usually not used directly by applications programmers.

flavors: This package contains an object-oriented programming system. See chapter 19 for details.

17.6. Package System Functions and Variables

In order to promote compatibility resulting from loading compiled or "source" files, certain package operations are treated as though they were surrounded by *(eval-when (compile load eval) ...)*; eval-when is described in the compiler chapter. These operations are *make-package, in-package, shadow, shadowing-import, export, unexport, use-package, unuse-package,* and *import.* These functions should appear only at top level within a file. The recommended procedure is to write files so that all of the package setup forms precede actual programs. Normally, then, at most one package would be defined per file. In a large package, it would be usual for several files to collectively define all the required programs.

For the functions described here, all optional arguments named *package* default to the current value of **package**. Where a function takes an argument that is either a symbol or a list of symbols, an argument of *nil* is treated as an empty list of symbols. The value of the variable **package** **must** be a package (not the name of a package: see find-package below to see how to obtain a package from its name). The value of **package** is referred to as the current package. Its value is initially the *user* package. During *load*ing, **package** is lambda-bound to its current value, so that it is restored automatically at the conclusion of the loading process.

(make-package 'k_pkname [:nicknames 'l_names][:use 'l_packs])

RETURNS: a new package with the specified package name.

NOTE: K_pkname may be either a string or a symbol. L-names is a list of strings or symbols to be used as alternative names for the new package. L_packs is a list of packages or the names of those packages whose external symbols are to be inherited by the package being defined. These packages must already exist. If not supplied, The default value of :use is a list of one package, the *lisp* package.

(in-package 'k_pkname [:nicknames 'l_nicklist][:use 'p_usepack])

SIDE EFFECT: the *in-package* function is intended to be used as a declaration placed at the start of a file containing a subsystem that is to be loaded into some package other than *user*. If there is not already a package named *k_pkname*, this creates this package as though *make-package* were called, and then sets **package** to this. Although this can be changed during the loading of the file by another call to *in-package*, normally the programmer will just allow the **package** variable to reverts to its old value at the completion of the *load* operation.

NOTE: If k_pkname is any existing package, presumably this is a re-loading or augmentation of a package that was loaded previously.

(find-package 's_name)

RETURNS: the package with the name or nickname s_name, otherwise nil.

(package-name 'k_pack)

RETURNS: A string which is the name of the package k_pack.

(package-nicknames 'k_pkname)

RETURNS: A list of strings which are nicknames for the indicated package. If there are no nicknames, nil.

(rename-package 'k_pkname 's_newname ['l_newnicknames])

SIDE EFFECT: The package denoted by the name k_pkname is now denoted by the s_newname and the nicknames, if any. The old name and nicknames no longer access a package.

(package-use-list 'k_pack)

RETURNS: A list of all packages used by k_pack.

(package-used-by-list 'k_pack)

RETURNS: A list of other packages that use k_pack.

(package-shadowing-symbols 'k_pack)

RETURNS: a list of symbols that have been declared as shadowing symbols in k_pack by *shadow* or *shadowing-import*. All symbols on this list are present in the specified package.

(list-all-packages)

RETURNS: a list of all packages, regardless of usage, which exist in the current LISP system.

(intern t_string ['k_package])

WHERE: The indicated package k_package (or if missing, the default **package**), is searched for a string t_string. This search will include inherited symbols. If a symbol with the specified name is found, it is returned. If no such symbol is found, one is created and is installed in the specified package as an internal symbol (as an external symbol if the package is the *keyword* package); the specified package becomes the home package of the created symbol.

RETURNS: A multiple value (two values): The first is the symbol that was found or created. The second value is one of four possibilities:

NOTE: *intern* may also be given a symbol (instead of a string), in which case, the symbol, if not found, will be inserted into the specified package. This allows uninterned symbol to be given a home package.

nil if t_string is being established for the first time with this call;

:internal
if t_string was already in the named package as an internal symbol.

:external
if t_string was already in the named package as an external symbol.

:inherited
if t_string was inherited in the named package.

```
=> (setq r (make-package 'newpack))
#<package newpack>
=> (intern "foo" r)
multiple values returned: newpack::foo nil
newpack::foo
=> (intern "foo" r)
multiple values returned: newpack::foo :internal
newpack::foo
```

(find-symbol s_string ['k_package])

RETURNS: multiple values: nil nil if the indicated symbol is inaccessible; otherwise a result identical to *intern*.

(unintern 's_symbol ['k_package])

RETURNS: t if s_symbol is found and removed, and nil otherwise.

NOTE: *unintern* changes the state of the package system and may cause problems with consistent reading and writing.

(export 'syms ['package])

RETURNS: t

WHERE: *syms* is a symbol or a list of symbols.

SIDE EFFECT: These symbols become accessible as external symbols in the *package*.

NOTE: A call to *export* at the beginning of a file should be used to announce its public symbols. An error break will result from various questionable practices; ?ret will continue.

RETURNS: t.

(unexport 'lg_sym ['k_package])

WHERE: lg_sym is a list of symbols, or a single symbol. These symbols, which presumably were external symbols in the indicated package (or the current *package*) become internal symbols in *package*.

RETURNS: t.

(import 'lg_sym ['k_package])
(shadowing-import 'lg_sym ['k_package])
The argument should be a list of symbols, or possibly a single symbol. These symbols become internal symbols in *package* and can therefore be referred to without having to use qualified-name (colon) syntax. *import* signals a correctable error if any of the imported symbols has the same name as some distinct symbol already accessible in the package. Shadowing-import plows through without the correctable error, and furthermore notes the symbols as shadowing.

RETURNS: t.

```
=> (import 'newpack::foobar)
If continued (with ?ret), Import these symbols with Shadowing-Import.
Break: Importing these into the user package causes a name conflict:
(foobar)
c{1} ?ret
t
```

(shadow 'lg_sym [k_package])

SIDE EFFECT: For each symbol in the list lg_sym, if it is directly present in the specified package, nothing is done. Otherwise it is instantiated as an internal symbol and placed on the shadowing-symbols list of k_package.

RETURNS: t.

(use-package 'lkst_packs [k_package])

WHERE: lkst_packs is a list of packages, a single package, a list of (string or symbol) package names, or a single package name;

SIDE EFFECT: These packages are merged onto the use-list of k_package. All external symbols in the packages to use become accessible in *package* as internal symbols.

RETURNS: t.

(unuse-package 'lkst_packs [k_package])

SIDE EFFECT: reverses the effect of use-package.

RETURNS: t.

(find-all-symbols st_name)

RETURNS: a list of all symbols whose print name is the specified string. If a symbol is provided, its print-name is used for the search.

(do-all-symbols l_ilist l_body)

WHERE: The iteration list l_ilist has the form (s_var [g_result])

SIDE EFFECT: *do-all-symbols* provides iteration over all symbols in all packages. The l_body is performed once for each symbol, where, each time through l_body, the s_var is bound to a different symbol.

RETURNS: nil or the result of evaluating the optional result-form g_result.

(do-symbols l_ilist l_body)
(do-external-symbols l_ilist l_body)

WHERE: The iteration list l_ilist has the form (s_var [k_package [g_result]])

SIDE EFFECT: *do-symbols* provides iteration over the symbols of a package. The l_body is performed once for each symbol accessible in the specified package, where, each time through l_body, the s_var is bound to a different symbol.

RETURNS: nil or the result of evaluating the optional result-form g_result.

NOTE: *return* may be used to terminate the iteration prematurely. The function *do-external-symbols* iterates over the external symbols only.

```
=> (do-symbols (i (find-package 'system) 'done) (progn (print i)(terpri)))
;; all the system package symbols .... <many omitted>.. concluding with
system:link
system:time
system:int-serv
system:nice
system:unlink
system:gethostname
system:getuid
system:fpeint-serv
done
=>
```

17.7. Modules

A *module* is a COMMON LISP subsystem that is loaded from one or more files. A module is normally loaded as a single unit, regardless of how many files are involved. A module may consist of one package or several packages. The file-loading process is necessarily implementation-dependent, but COMMON LISP provides rudimentary machinery for naming modules, for keeping track of which modules have been loaded, and for loading modules as a unit.

modules is a list of names of the modules that have been loaded into the LISP system so far. This list is used by the functions *provide* and *require*. Each module has a unique name (a string). The *provide* and *require* functions accept either a string or a symbol as the *module-name* argument. If the module consists of a single package, it is customary for the package and module names to be the same.

(provide 's_name**)**
this declarative function is used in a file defining a set of functions.

SIDE EFFECT: adds the module name s_name to the list of modules **modules**, indicating that the s_name module has been previously loaded.

(require 's_name ['sl_pathname]**)**

SIDE EFFECT: The *require* function tests whether the s_name module is already present. If not, it proceeds to load the appropriate file or set of files. The pathname argument, if present, is a single pathname or a list of pathnames whose files are to be loaded in order, left to right. Conventional path search-names will be used if necesary to find the files.

A convenient way to customize a system is to require certain modules to be loaded into *user* package whenever FRANZ LISP is started afresh. Typically an installation or a group of programmers will set up an initialization file in the *.lisprc* file when FRANZ LISP for this purpose. Usually such an initialization file simply causes other facilities to be loaded.

17.8. Creating Packages in Files

When each of two files uses some symbols from the other, one must be careful to arrange the contents of the file in the proper order. Typically each file contains a single package that is a complete module. The contents of such a file should include the following items, in order:

A call to *provide* that announces the module name.

A call to *in-package* that establishes the package.

A call to *shadow* that establishes any local symbols that will shadow symbols that would otherwise be inherited from packages that this package will use.

A call to *export* that establishes all of this package's external symbols.

Any number of calls to *require* to load other modules that the contents of this file might want to use or refer to. (Because the calls to *require* follow the calls to *in-package*, *shadow*, and *export*, it is possible for the packages that may be loaded to refer to external symbols in this package.)

Any number of calls to *use-package*, to make external symbols from other packages

accessible in this package.

Any number of calls to *import*, to make symbols from other packages present in this package.

Finally, the definitions making up the contents of this package/module.

For very large modules whose contents are spread over several files it is recommended that the user create the package and declare all of the shadows and external symbols in a separate file, so that this can be loaded before anything that might use symbols from this package.

CHAPTER 18

Interfacing Foreign Functions to Franz

Contents:

[Adapted in part from "Parlez-vous Franz" by J. Larus, University of California, Berkeley.]

FRANZ LISP is unusual in its capability to load object modules into a running system and to call "foreign" functions, i.e., functions and subroutines written in other programming languages. This document describes how FRANZ LISP can interface with functions written in C and languages following similar conventions. Pascal and Fortran generally fall in this category under most versions of the Unix operating system. The FRANZ LISP process can also interface with other processes by means of Unix pipes.

18.1. A simple example

This section gives an example of a C function which can be compiled and loaded into FRANZ LISP. It can then be called as if it were a LISP function. The example is a function of two arguments, an integer and a real, and returns an integer value.

The basic integer and real types in FRANZ LISP are called fixnums and flonums, respectively. Fixnums are 32-bit two's complement integers and flonums are 64-bit floating-point numbers. The corresponding types in C are ints and doubles. (Note: The size of an int depends on the implementation of C. If int is not 32 bits but long is, then the C program must specify long instead of int.)

The following C function takes an integer and a real argument, and returns 1 if the integer is larger, and 0 otherwise.

```
int
great(x,y)
int x;
double y;
{
    return x > y ? 1 : 0 ;
}
```

In the Unix operating system, this function can be put into a file called *file.c* and compiled with

```
cc -c file.c
```

The resulting object file is *file.o*. Within Franz LISP, the C object file can be loaded with:

(cfasl 'file.o '_great 'greater 'c-function)

The function cfasl loads the file *file.o*, looks for an entry-point called _great, and makes it into a LISP function with the name greater. (It is a peculiarity of most implementations that the C compiler prepends an underscore to function names.) The symbol *c-function* indicates that the function arguments are to be treated as "call by value" as in C, rather than as in LISP functions, which pass and return pointers to LISP objects. The function can then be called in LISP just like any other LISP function:

```
=> (greater 7 3.1)
1
=> (greater 5 6.2)
0
```

The C function can use any of the features normally available to C functions. It can read and write to the standard I/O, open and close files, and call C library functions. More complicated interfaces with LISP are treated in the following sections. While most foreign functions merely compute a value and return it as indicated above, it is possible to use foreign functions for system call interfaces, graphics, multi-processing, etc.

18.2. Rules for loading C functions

As illustrated above, a function can be loaded into LISP and then invoked. A function not written in LISP must still maintain certain environmental consistencies with the LISP environment. Usually, such foreign functions are written so they can be treated as "black boxes" that take arguments and produce answers, but produce no side effects on the environment.

Before being loaded into LISP, a function must be incorporated into an object module along with whatever functions it calls. The C compiler's output (e.g., *work.o*) is an object module, but the loader's usual output (e.g., *a.out*) is an executable file, not an object module. If all the functions of interest are in one file, and it is compiled with the *-c* flag, an object module will be produced.

If the functions are in more than one file and refer to each other, then the compiled object modules should be combined into a single module that can be loaded into LISP while resolving external references. The modules are combined by the loader, with its *-r* switch. The resulting file, either *a.out* or whatever follows the *-o* switch, is in this case, an object, not an executable, module.

For example, suppose we want to combine the files *in.c*, *out.c*, and *work.c* into the object module *code.o*:

```
cc -c in.c out.c work.c
ld -r in.o out.o work.o -o code.o
```

Once in LISP, you load an object module with the *cfasl* function:

(**cfasl** 'f_name 's_cname 's_lispname ['s_discipline [s_libraries]])
(**ffasl** 'f_name 's_cname 's_lispname ['s_discipline [s_libraries]])

WHERE: f_name is the module's file name, s_cname is the entry point of the C function name; s_lispname is the LISP symbol name bound to the code, and can be different from the entry-point name; s_discipline is function's "calling discipline" (see below); s_libraries is also optional and lists the libraries to be searched to resolve references in the object module.

NOTE: some C compilers prepend an underscore (_) to functions' names; if yours does, you should do the same. *ffasl* is just like *cfasl* except that it is intended for loading Fortran modules, and the proper Fortran libraries are loaded. The loading of Fortran programs is not supported in all versions of FRANZ LISP.

SIDE EFFECT: *Cfasl* normally prints out the command line passed to the loader. This message can be suppressed by binding the variable *$ldprint* to *nil*. Normally, the file is loaded and the function s_lispname is consequently available from LISP.

In our example above, we loaded the function *greater* from the module *file.o* and made the function available in LISP under the name *greater*:

(cfasl 'code '_greater 'greater 'c-function)

If, in addition to the function loaded by *cfasl*, you wish to use other functions from a module, notify LISP of their names and disciplines with the *getaddress* function (do not call *cfasl* again). For example, suppose that the file *file.o* also containes the function *less* defined below.

```
int
less(x,y)
int x;
double y;
{
    return x < y ? 1 : 0 ;
}
```

To make this function available in LISP, use the following command:

(getaddress '_less 'less 'c-function)

To make several functions available, it is more efficient to call getaddress once with a sequence of triples as arguments than to call getaddress several times.

The function *removeaddress*, removes a name from the foreign function symbol table in LISP, thereby making it possible to *cfasl* another file that contains that name. Before *cfasl*ing a file for the second time, you must remove from LISP *all* names in that file. Thus, to remove the functions *greater* and *less* from LISP, use the following command:

(removeaddress '_greater '_less)

*Cfasl*ing a file is somewhat slower than loading a equivalent-sized file of LISP functions with *fasl*. If a collection of C functions is commonly used (and sufficient disk space is available), the process containing the *cfasl*ed functions can be saved by the *dumplisp* function.

When a file is *cfasl*ed into LISP, there is always the risk that a name conflict will occur. If you use an external name which happens to be the same as an external name that is used by the LISP system, then the loader will report an error that the name is multiply defined. There is no way for you to change a name inside the LISP kernel, so you must change the entry-point name in the file that you are *cfasl*ing in.

18.3. Function disciplines

Every foreign function has a *discipline* that is declared when the function is incorporated into LISP. The *discipline* tells LISP how to pass arguments to the function, and what type of results to expect when the function is called, so that LISP can store it as a conventional LISP value.

Franz LISP recognizes eight foreign function disciplines. The first four are more suited for "call by address" languages, like Fortran, while the last four are designed for "call by value" languages like C. The function disciplines are:

18.3.1. Call-by-address disciplines

function

This is a function which takes LISP values as arguments, and returns a LISP value. This is as close to a LISP function as a foreign function can be. Recall that in LISP, function arguments are normally evaluated, but a LISP value is a pointer to the object. Thus the function must return a pointer to a valid LISP object, or a LISP error may result. Normally this is accomplished by passing the function a LISP object which is modified and then returned as the value of the function. Since the resulting value originated in LISP, LISP treats it like any other LISP object.

subroutine

This is the default discipline. As with the previous (function) discipline, the arguments passed to the function are pointers to LISP objects. However, with a subroutine, the return value, if any, is ignored. Franz always returns t from a subroutine.

integer-function

This returns an integer which Franz stores as a fixnum. Again the arguments passed to the function are pointers to LISP objects.

real-function

This is just like an integer-function except that the value returned is a double precision real number which Franz stores as a flonum.

18.3.2. Call-by-value disciplines

c-function

This is a C function which returns an integer, which Franz stores as a fixnum. This differs from an integer-function in that fixnum and flonum arguments are passed by value. Other arguments (except possibly for structures) are passed unchanged.

void-c-function

This is a C function whose return value is ignored. It is just like c-function except that in LISP, t is always returned. (In C, the function can be declared to be of type *void*, or of any other type.)

double-c-function

This is a C function which returns a double, which Franz stores as a flonum. Arguments are treated the same as for a c-function.

vector-c-function

This is a C function which returns a structure. This type of function is used less frequently and is explained in a later section.

18.4. Rules for calling C functions

The major requirement for successfully using foreign functions is an understanding of how values are passed to the functions and how the results are returned. LISP has a definite conception of where particular types of objects belong and of their structure. C, on the other hand, respects few, if any, conventions and C functions have difficulty in using the intrinsic LISP functions for creating, modifying, or storing LISP objects. Foreign functions can find the fields of LISP structures and use the values in them, but these functions cannot easily deduce the types of LISP objects. Hence arguments to foreign functions and values returned are assumed to be of known correct types. Values created and returned by foreign functions are either treated as LISP values, without any checking, or as tightly closed "black boxes".

LISP values passed to foreign functions are treated specially by LISP, with the exact treatment depending on the value's type:

18.4.1. integers and real numbers

Integers and real numbers are copied and a pointer to the new value is passed. The values are copied so that foreign functions cannot change a shared number (small integers are stored uniquely). Arguments of this sort cannot be used for returning results to LISP.

18.4.2. strings and symbols

Strings are passed as pointers to null-terminated sequences of characters. Changes made by the C function to a string affect the original LISP string.

Symbols are passed as pointers to the symbol data structure (see Chapter 1). The symbol's data structure has many useful fields, but the whole structure generally need not be passed to a foreign function. If the foreign function needs only the symbol's literal representation, use its print name (see *get_pname*), which is a string easily handled by C functions.

18.4.3. vectors and arrays

Vectors, immediate-vectors, and arrays may be created in LISP and passed to C functions as arguments.

LISP immediate-vectors are similar to one-dimensional arrays in C. A vectori-byte corresponds to a C char array, a vectori-word corresponds to a C short int array,

and a vectori-long corresponds to a C array of 32 bit integers. (Depending on the C compiler, these could be ints or longs or both.) There are also objects which correspond to C float and double arrays.

The following C function fills a given array with zeros:

```
int zero(n,a)
int n, a[];
{   int i;
    for ( i=0; i<n; ++i)
      a[i] = 0;
 return n;
}
```

If this is in a file called *zero.c* and compiled into an object file called *zero.o*, then it can be loaded into LISP and called by LISP as follows:

```
=> (cfasl 'zero.o '_zero 'zero 'c-function)
t
=> (setq x (new-vectori-long 10))
#vector[40]
=> (vseti-long x 0 27)          ;; set x[0] := 27
27
=> (vrefi-long x 0)             ;; print it out
27
=> (zero 10 x)
10
=> (vrefi-long x 0)             ;; now it's gone
0
=>
```

A LISP vector is similar to an array of pointers in C. The following C function rotates the pointers in a given array:

```
int rotate(n,a)
int n, *a[];
{   int i, *p;
    p = a[0];
    for ( i=0; i<n-1; ++i)
      a[i] = a[i+1];
 a[n-1] = p;
    return n;
}
```

If this is in a file called *rotate.c* and compiled into an object file called *rotate.o*, then it can be loaded into LISP and called by LISP as follows:

```
=> (cfasl 'rotate.o '_rotate 'rotate 'c-function)
t
=> (setq x (vector 0 1 2 3))
#vector[4]
=> (rotate 4 x)
4
=> (vref x 0)
1
=> (vref x 1)
2
=> (vref x 2)
3
```

```
=> (vref x 3)
0
=>
```

To pass floating point arrays, the following functions are available:

```
vectori-float
vectori-double
new-vectori-float
new-vectori-double
vrefi-float
vrefi-double
vseti-float
vseti-double
```

These functions work just like the corresponding vectori functions for bytes, words, and longs. For example, the Lisp code to create a *double* vectori with size appropriate for ten doubles, initialize it with zeros, and set the seventh entry equal to 12.3.

```
=> (setq x (new-vectori-double 10))
vectori[80]
=> (vseti-double x 7 12.3)
12.3
```

Since vectori-floats and vectori-doubles were not available in versions of FRANZ LISP prior to Opus 42, the appendix to this chapter gives substitutes which work on Opus 41.

Arrays can be passed to foreign functions, but they are somewhat more complicated than vectoris and in most cases it is preferable to use vectoris. For the "call by address" disciplines, arrays are passed as pointers to an array's *data* field. The values in the field are generally not stored sequentially in memory, but are pointed to by sequentially-stored pointers. In order to pass an array using the C-style "call by value" discipline, it is necessary to know what the array's data field looks like. The appropriate structure is defined in the header file lstructs.h which is part of the FRANZ LISP distribution and is usually kept in the directory /usr/franz/franz/h.

Fixnum-block and *flonum-block* arrays are alternatives to immediate vectors for passing numbers to Fortran routines other foreign routines using "call by address" disciplines. *Fixnum-block* arrays are integer arrays and *flonum-block* arrays are double precision arrays. If you want to pass a contiguous block of objects in an array, then the type of the array must be declared to be *fixnum-block* or *flonum-block*, depending on the type of LISP values that are stored in the array. (*Fixnum-block* and *flonum-block* arrays are alternatives to immediate vectors for passing numbers to Fortran routines. *Fixnum-block* arrays are integer arrays and *flonum-block* arrays are double-precision arrays.) Changing a value in an array propagates the new value back to LISP.

18.4.4. lists

It is possible to pass lists and other LISP objects to a C function, and have the C function perform LISP-like operations on them. Creating new LISP data is somewhat tricky, and recommended only for sophisticated Franz users.

As an example of some of the pitfalls, consider a C function intending to return a LISP list containing two integers. The C function might require that it be be passed a workspace of a LISP list of length two as an argument and then proceed to modify it. However, if the original list contains an integer between -1024 and 1023, then the C

function cannot change ("clobber") the value of that integer since it is stored in read-only memory to preserve its uniqueness. If the numbers are larger, then they are not stored uniquely and so can be changed.

The Franz files lstructs.h and global.h define most of the relevant structures. It is usually kept in the directory /usr/franz/franz/h. In particular, lists are defined there to be pointers to a data structure made of a pair of pointers (*cdr* followed by *car*, both full 32-bit words).

The following C function is a simple example which manipulates lists. It takes two arguments, both lists, and destructively appends the second one to the first.

```
#include "global.h"

lispval
nconc(a,b)
lispval a, b;
{
   lispval x, y;

   if (TYPE(a) != DTPR || TYPE(b) != DTPR)
   {
        printf("Error: arguments must be lists.\n");
        return nil;
   }

   if (a == b) return b;

   for ( x = a; x != nil; x = x->d.cdr)
        y = x;

   y->d.cdr = b;
   return a;
}
```

Notes:

1. The header file *global.h* must be included, as essential types and macros are defined there. This header file includes other header files in the same directory as *global.h*. In case these files are in the directory */usr/franz/franz/h* and the above C function is in a file called *nconc.c*, the Unix operating system command to compile the file is

```
cc -c -O -I/usr/franz/franz/h nconc.c
```

2. A lispval (LISP value) is the type by means of which s-expressions are referenced in C. It is a pointer to a lispobj (LISP object) which is a cell which might hold a number of different things. If the LISP object is a dotted pair, it holds two lispvals, its cdr followed by its car. In C, these declarations are:

```
struct atom { lispval clb; ... };
typedef union lispobj *lispval;
struct dtpr { lispval cdr, car;};
union lispobj {
  struct atom a;
  FILE *p;
  struct dtpr d;
  double r;
```

... };

These structures are occasionally modified with new release of FRANZ LISP so "including" the header file is preferable to copying the type definition.

3. *TYPE* is a macro for testing types. Lists have the type *DTPR* (dotted pair). Other type macros are in the file *ltypes.h*.

4. The macro *nil* is defined in *global.h*. It is a pointer to the atom called *nilatom* in C. The macro *nil* is defined to be zero on some machines, but not all.

5. The above function is destructive in the sense that it usually modifies its first argument, as does the FRANZ LISP nconc. No storage is allocated or released.

18.4.5. passing C structures by value

The C language has the feature that structures and unions can be passed by value, either as an argument to a function or as the value returned by a function. Franz can interface with a C function which passes as an argument a structure if:
1. The function's discipline is *c-function*, *void-c-function*, *double-c-function*, or *vector-c-function*.
2. The structure is represented as a vectori in LISP.
3. When the structure is passed by value, its property field is set to *value-structure-argument*.

Franz can interface with a C function which returns a structure by value if:
1. The function's discipline is vector-c-function.
2. When the C function is called from LISP, the first argument in LISP is an immediate vector (vectori) into which the result is to be stored. The second LISP argument is then the first argument to the C function, and so on.

An example where structures are passed by value both to and from the C function is the following function:

```
struct foo { int a, b, c;};
struct foo bar(x)
struct foo x;
{
    int t;
    t = x.a;
    x.a = x.b;
    x.b = t;
    x.c += 13;
    return x;
}
```

If this function is in a file called barfile.c and compiled into barfile.o, then it can be loaded into LISP with:

```
=> (cfasl 'barfile.o '_bar 'bar 'vector-c-function)
t
=> (setq foo-in (vectori-long 12 23 34))
vectori[12]
=> (setq foo-out (new-vectori-long 3))
vectori[12]
=> (vsetprop foo-in 'value-structure-argument)
value-structure-argument
=> (bar foo-out foo-in)
```

```
vectori[12]
=> (vrefi-long foo-out 0)
23
=> (vrefi-long foo-out 1)
12
=> (vrefi-long foo-out 2)
47
=>
```

18.4.6. accessing C structures from LISP

The file cstruct.l contains LISP functions which are useful for accessing C structures passed back to Franz. It is a library file and it is located (along with its compilation) with the other LISP library files (usually /usr/lib/lisp). If you invoke the LISP function *c-declare* it will be loaded into memory by the auto-load feature.

C-like structures are implemented in LISP with immediate vectors as the underlying data type. You can avoid structures in LISP just as you can avoid it in C by counting byte offsets, but in either case the process is tedious and error-prone.

18.4.6.1. An example

Here are some structures taken from a popular window system:

```
struct pr_pos { int x,y; };
struct pr_subregion {
        struct   pixrect *pr;
        struct   pr_pos  pos;
        struct   pr_size  { int x,y; }       size;
};
```

Suppose we want to call a function in C which takes, as an argument, a pointer to an object of type pr_subregion, and assigns values to the fields of p. We could allocate an immediate vector of size 20, and bind it to something, say,

```
(setq p (new-vectori 20)).
```

We could then call the C function with p as a parameter, and the C function would have no trouble assigning values to the fields of p. However, it would be difficult to decompose the structure in LISP and access the various fields of p. Futhermore, the garbage collector would not handle the pointers correctly.

The above structures can be declared in FRANZ LISP in order to circumvent these difficulties. The declarations are:

```
(c-declare
 (struct pr_pos
          (x int)
          (y int))
 )
(c-declare
 (struct pr_subregion
          (pr * (struct pixrect))
          (pos (struct pr_pos))
          (size (struct pr_size
                        (x int)
                        (y int)))))
```

These declarations create a number of LISP functions. They create *make-pr_subregion* which allocates the necessary memory for the structure pr_subregion. They also create functions which access the fields of the structure. The LISP statement

```
(setq p (make-pr_subregion))
```

allocates memory and makes p a pointer to it. In LISP, p has the type *vectori*. If p is passed to C, it should have the C declaration

```
struct pr_subregion *p;
```

The field which would be accessed in C by

```
p->pos.x
```

is accessed in LISP by

```
(pr_subregion->pos.x p)
```

and assigned to the value of b by

```
(setf (pr_subregion->pos.x p) b)
```

18.4.6.2. Syntax

(c-declare l_struct1 [l_struct2 ...])

Each structure declaration is a list. The first element is either of the symbols **struct** or **union**. The second element is a symbol, taken to be the name of the structure. Each subsequent element is another list taken to a declaration of a member of the structure. The first element of the member declaration is its formal name and the remaining elements in the list are its type description.

The type description may be a symbol representing a scalar type. Scalar types from C that are currently implemented are char, short, long, int, float, double, unsigned-char, unsigned-short, unsigned-int, and unsigned-long. The types int and long are synonymous.

Another entry may be the symbol "*", indicating the type is a pointer to an object whose type is given by the remainder of the list. Pointers can be to any type, including unions and structures.

An entry may be the symbol "function", indicating the type is a C function returning an object described by the remainder of the list. It will always be preceded by * since arrays or structures cannot contain functions directly. There is no facility in LISP for doing anything with a pointer to a C function, except to pass it to another C function which could make use of it.

Entries may be lists. To describe an array of objects, the entry is a list beginning with the symbol **array** and the remaining entries are the dimensions of the array.

You may indicate structures or unions either by having a list whose first element is either **union** or **struct** and whose second element identifies the structure. As in the language C, you can formally declare the structure here.

The type **lispval** is provided for storing lisp data in a structure. In C, such an object is always a pointer, whereas in Lisp it may be any Lisp object.

Here is a more complex example:

```
typedef struct { int *cdr, *car;} *lispval;
struct a { int m;
           short n;
           struct c *p;
           struct box { int x, *y;} b;
           int (*f[3][4])();
           char c, c1, c2;
           lispval l;
           float r[3][4];
};
struct c { int x, y;
           lispval ls;
};


(c-declare (struct a
                   (m int)
                   (n short)
                   (p * (struct c))
                   (b (struct box (x int) (y * int)))
                   (f (array 3 4) * function int)
                   (c char)
                   (c1 char)
                   (c2 char)
                   (l lispval)
                   (r (array 3 4) float)
                   )
           (struct c (x int) (y int) (ls lispval))
                   )

A structure s which is declared in C as
        struct a *s;
has a float entry which is accessed in C as
        s->r[1][2];
and in Lisp as
        (a->r s 1 2)
```

18.4.6.3. Semantics

By writing one of these declarations, many functions may be generated. For each structure, you will get a creation function whose name is the concatenation of **make–** and the structure name. It is actually a macro and allows you to initialize individual fields by member name. Initializers can be inserted as pairs of an unevaluated member name and an expression to compute its value. Initialization for arrays is not currently implemented. The LISP statement

```
(setq p (make-pr_pos x 50 y (+ 20 (* 5 40))))
```

is thus equivalent to the C statement

```
struct pr_pos p = { 50, 20 + 5*40 };
```

There are two classes of accessors created. The first assumes that the structure is in an immediate vector allocated by FRANZ LISP. These functions are made by concatenating the structure name, the symbol ->, and the member name. If the member is a structure in its own right, additional accessor functions are generated for each member of the substructure. Thus the LISP expression

```
(pr_subregion->pos.y p).
```

is analogous to the C expression p->pos.y. The other class of accessors assume that the first argument is an integer, whose value is really a pointer to a structure. Their names have an asterisk prepended. This is particularly useful for structures which contain pointers to other structures. For example, suppose one wished to create a structure in Lisp to pass to C using the C declaration:

```
struct dog {    int mice;
                struct cat { char head,paw;} *teeth;
                } *rover;
```

Then one would say in LISP,

```
(c-declare (struct dog
                (mice int)
                (teeth * (struct cat (head char) (paw char)))))
(setq rover (make-dog teeth (make-cat)))
```

The C expression rover->teeth->head has as its Lisp equivalent

```
(*cat->head (dog->teeth rover))
```

18.4.6.4. array accessors If a member is an array, the corresponding accessor is a function of several arguments: the base structure and as many indices as were declared. It returns an element of the array.

18.4.6.5. pointers to structures One potentially confusing aspect of interfacing LISP to C is that LISP generally uses one more level of indirection than C does. In LISP, an integer is really a pointer to an integer. Likewise, the structures and pointers to structures in C become pointers to structures and pointers to pointers to structures in LISP. Thus, when a structure contains a pointer to another structure, and that pointer is accessed in LISP, it is really a pointer to a pointer to a structure. The function *maknum* is useful in this context, since it effectively creates a pointer to its argument. The following example illustrates using pointers to structures.

```
(c-declare (struct outer
                    (x int)
                    (pt * (struct inner (a short)
                                        (b short)))))

(setq s (make-outer x 7))
(setf (outer->pt s) (maknum (make-inner a 12 b 23)))
(msg (*inner->b (outer->pt s)) " should be 23" N)
```

18.4.6.6. LISP data Putting LISP data into a C structure presents a garbage collection problem. The garbage collector assumes that the contents of a vectori is raw data, and does not pay any attention to what pointers within the vectori point to. For this reason, there are some special procedures for protecting LISP data in a C structure from garbage collection.

When a C structure has a field of type lispval in it, and *setf* is used with the cstructs accessor functions to assign a LISP object to that field, then a copy is put on the property list of the vectori. Likewise, if a lispval field of a new structure is being initialized, a copy is also put on the property list.

If LISP data is put into a C structure by a C function, or by a LISP function other than setf, a copy of that LISP data must be kept elsewhere in order to protect it from garbage collection. Each c-declare of a structure which contains lispvals creates a function with the name *protect-* concatenated with the structure name. This function takes one argument, a vectori which has been created by cstructs, and copies all of the lispval fields within the structure into the property list of the vectori. For example, (protect-slop) will save the lispval fields in a vectori created by (make-slop).

18.4.6.7. debugging tools

(describe-cs v_item**)**

WHERE: v-item is a vectori which has been created by a cstructs "make-" function.

SIDE EFFECT: Information about the structure is printed out, including the structure tag, the field names, the C type of each field, and the current value of each field. The list of accessor macros is also printed out.

(double-to-float 'f_flo)

RETURNS: a fixnum, which may be passed to a C routine expecting an argument of type *float*.

NOTE: In FRANZ LISP, flonums are double precision floating point numbers which correspond to the C type *double*. This function and *float-to-double* provide a way to convert between a lisp flonum (a C double) and a C float (which is the same size as a lisp fixnum).

(float-to-double 'x_fix)

RETURNS: a flonum, after converting the fixnum x_fix from a C *float* to a lisp flonum.

NOTE: See the discussion at *double-to-float*.

18.5. The C program

18.5.1. external references

Foreign functions can do much of what they would do if executed autonomously. They can use and change *extern* and *static* variables and invoke other functions that are in *cfasl*ed files or are part of FRANZ LISP. Even input and output from foreign functions is fine.

18.5.2. memory allocation

C functions can use the C library function *malloc()* to dynamically allocate memory. Memory allocated in this way will not be garbage-collected by LISP. The LISP system knows that this value is foreign and does not allow it to be used in place of a LISP value. The only use of these foreign values is as arguments to foreign functions. However, LISP objects cannot be allocated this simply, because the LISP system must know the data type at memory allocation time. Allocating LISP objects from your own foreign function is complicated, and not recommended. It is usually much better to allocate LISP objects in LISP, and pass pointers to the C function.

18.5.3. calling Franz from C

LISP functions cannot get the value of a C variable unless it is returned by a C function. However, C functions can call the built-in function *matom*, which takes a symbol's name (as a string) as its argument and returns a pointer to the symbol's data structure. If the symbol does not exist, it is created. The C code can look at the symbol's value field to find a variable's binding. Usually, passing this value as an argument is safer and easier than finding it from the C code.

C functions can invoke LISP functions. The built-in function *ftolsp*'s first argument is an integer indicating how many arguments follow. The next argument is the symbol whose function binding is to be invoked. The rest of the arguments to *ftolsp* are LISP values that are passed to the LISP function. *Ftolsp* returns the result returned by the LISP function. *Ftolsp* has the same arguments and effect as the *funcall*

function does in LISP.

For example, suppose a C function is set up to find the time consumed by LISP. The following C code invokes the LISP function *ptime*:

```
x = ftolsp(1,matom("ptime"));
```

Ftolsp returns a value of type *lispval*, which in this case is a pointer to a dtpr cell.

Passing arguments to LISP from C requires some knowledge of FRANZ LISP internals. For example, the following function returns the product of its arguments by calling the function *times*.

```
prod(x,y)
int x, y;
{
 /* As long as there is no overflow, the next line produces
    the same result as  return x * y;  */
    return * (int *) ftolsp(3,matom("times"),inewint(x),inewint(y));
}
```

In LISP, integers are "boxed" in the sense that a LISP value of type fixnum is really a pointer to a memory location where the integer is stored. The built-in function *inewint* takes an integer argument and returns a corresponding LISP value, i.e., a valid LISP pointer to a memory location where the integer is stored. *Ftolsp* returns a LISP value which in this case is a fixnum, and hence a pointer to what C would understand as an integer.

On some systems, notably VAXes and Suns, the function *ftolsp_* is available. It allows the number of arguments to be omitted, so the above example is equivalent to:

```
x = ftolsp_(matom("ptime"));
```

*Ftolsp_* thus has the same arguments and effect as the *funcall* function does in LISP.

18.6. Fortran and Pascal

{This languages may not be available on your system.}

The cfasl function loads into LISP an object file which has the format specified by the C compiler. On those systems where Fortran and Pascal use the same object-module format as C, routines written in these languages may be loaded into LISP much the same way C functions can be. The differences are:

a. Calling conventions. C is call by value, Fortran is call by address, and Pascal allows either.
b. Fortran does not have pointer types.
c. Pascal allows passing arrays and structures by value, but interfacing such a procedure with Franz is different.
d. Libraries must be explicitly invoked for Fortran and Pascal.

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 how how LISP can interact not only with C, as previously discussed, but with two additional languages: 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 function prints its arguments (or parts of them). The 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 it will not work for fixnums whereas the LISP *memq* will work for small fixnums). In the script, typed input is in **bold**, computer output is in roman and comments are in *italics*.

These are the C coded functions
% **cat ch8auxc.c**

```
/* demonstration of c coded foreign integer-function */

/* the following will be 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
% **cat 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 we declare " a : integer " instead of " var a : integer " since
  we 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 -c ch8auxc.c
1.0u 1.2s 0:15 14% 30+39k 33+20io 147pf+0w
% pc -c ch8auxp.p
3.0u 1.7s 0:37 12% 27+32k 53+32io 143pf+0w
```

```
% lisp
Franz Lisp, Opus 42.04
```

First the files are loaded and we set up one foreign function binary. We have two functions in each file so we must choose one to tell cfasl about. The choice is arbitrary.

```
=> (cfasl 'ch8auxc.o '_cfoo 'cfoo "integer-function")
/usr/lib/lisp/nld -N -A /usr/local/lisp -T 63000 ch8auxc.o -e _cfoo -o /tmp/Li7055.0  -lc
t
=> (cfasl 'ch8auxp.o '_pfoo 'pfoo "integer-function" "-lpc")
/usr/lib/lisp/nld -N -A /tmp/Li7055.0 -T 63200 ch8auxp.o -e _pfoo -o /tmp/Li7055.1 -lpc -lc
t
```

Here we set up the other foreign function binary objects

```
=> (getaddress '_cmemq 'cmemq "function" '_pmemq 'pmemq "function")
#6306c-"function"
```

We want to create and initialize an array to pass to the cfoo function. In this case we create an unnamed array and store it in the value cell of testarr. When we create an array to pass to the Pascal program we will 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 we 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. Note that the entry point corresponding to the Fortran function ffoo is _ffoo_ as opposed to the C and Pascal convention of preceding the name with an underscore.

```
% cat 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
% f77 -c ch8auxf.f
ch8auxf.f:
  ffoo:
0.9u 1.8s 0:12 22% 20+22k 54+48io 158pf+0w
% lisp
Franz Lisp, Opus 42.04
=> (cfasl 'ch8auxf.o '_ffoo_ 'ffoo 'real-function "-lI77 -lF77")
/usr/lib/lisp/nld -N -A /usr/local/lisp -T 63000 ch8auxf.o -e _ffoo_
-o /tmp/Li11066.0 -lF77 -lI77 -lc
t

=> (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)
```

18.7. Pipes

{If you are using a non-Unix operating system, pipes may be unavailable on your system.}

Pipes are the principal method of program composition in the Unix operating system. They offer the considerable virtues of being easy to understand, quick to program, and standard throughout Unix. However pipes also consume substantial overhead in comparison to function calls described earlier in this chapter, and are ill-suited to passing structured data. Nevertheless, pipes are important, particularly since they are the only method used for many existing programs.

18.7.1. LISP in a Pipeline

Frequently, a program – e.g. the shell – creates a pipeline, one of whose component programs is written in LISP. The LISP program reads its data from the standard input and writes its results to the standard output (i.e. neither *read* nor *print* requires a port be specified as an argument).

At this point, a quick reader may wonder how a LISP program, which is usually run by an interpreter, can be executed by the shell. If the LISP program is interpreted, the commands that invoke *lisp*, load the requisite files, and start the program running can be put in a shell command file, which can be invoked in the same way as any other Unix command. If the LISP program is compiled, it can be compiled in a way that allows it to execute without being explicitly loaded into the interpreter (see the *liszt* compiler's *-r* switch). The executable file produced by *liszt* with the -r flag is invoked as is any other compiled program; but before the LISP code is run, the LISP interpreter is invoked and the compiled code loaded.

18.7.2. Using Sub-Processes

The complementary situation is not as simple – a LISP program invokes another program, perhaps supplying it with data and retrieving its results. In addition to starting this program running, someone must connect the pipes that service the program. LISP provides functions to handle this.

Releases of FRANZ LISP after Opus 38.30 have a very flexible set of functions – **process, *process-send,* and **process-receive.*

**process* takes between one and three arguments. The first is the subprocess's file name. The other two are optional boolean flags that, if present and non-nil, cause **process* to open ports to and from the subprocess, respectively. **process* returns a list of up to three items. If ports are opened, they are the first items. If a port is not opened, nothing is put into its place in the list (i.e. if only one port is opened, the list contains two items). The last item in the list is always the subprocess's id number. The values in the list are easily assigned to individual variables by the *desetq* function.

The example above reduces to:

```
(let* ((ResultPort (car (*process 'pwd nil t)))          ; NB args evaluated
       (Result (read ResultPort)))
      (close ResultPort)
      Result)
```

Since communication with a process is frequently in one direction and the order of arguments to **process* is difficult to remember, the other two function – **process-send* and **process-receive* – take only the subprocess's name and return a list of an open port and a process id.

Hence the example is best written:

```
(let* ((ResultPort (*process-receive 'pwd))
       (Result (read ResultPort)))
      (close ResultPort)
      Result)
```

18.7.3. The Perils of Pipes

In addition to the usual care which must be exercised by users of pipes, LISP programmers must observe additional cautions unique to FRANZ LISP. Pipes belong to the LISP process, not the executing LISP program.

Most important, each pipe uses a port from LISP's limited repertoire (20, but two are dedicated to the standard input and output). If a LISP function does not close its pipes for any reason (e.g. the function dies), the ports continue to be "used" and the associated subprocesses continue to exist. If all the ports are used up while you are debugging a program or running an erroneous program that fails to close ports, you will get an error from the process-creation function that it cannot open a port.

Execution of LISP's *resetio* function which closes the open ports (with the exception of standard input and output) will cure this problem; however, this indiscriminately severs the connections to all subprocesses and kills them. Unless you run out of ports by erroneously leaving them in use, you should probably avoid the *resetio* function. (if you must call *resetio* to close all pipes, remember to call *wait* repeatedly; see below).

You should call *wait* every time a process dies. Calling *wait* ensures that a dying process does not linger as a zombie, which can cause you to exceed your process limit. (If a process creation function fails for no apparent reason, check for "zombies" by CTRL-Zing and running *ps.*) *Wait* returns a dotted-pair consisting of the terminated process's id and status. Contrary to the implication in some previous editions of the LISP manual, *wait* always returns immediately. If no processes have died since the last call on *wait*, then the returned process-id is -1.

18.8. Appendix

To make it easier for users of FRANZ LISP to pass floating-point arrays to C functions, vectori-floats and vectori-doubles have been added to Opus 42. Users of Opus 41 systems can use the following functions as substitutes for the Opus 42 features.

```
; This file has a few functions which make it easier
; to pass floating point arrays to a C function.
; It is valid for Opus 41 only.
; These functions should NOT be used with Opus 42
; as they will be built into Opus 42 and higher.

; create new vectori to hold given number of floats
(defun new-vectori-float (x)
  (do ((v (new-vectori-long x))
       (n 0 (1+ n)))
      ((eq n x) v)
      (vseti-float v n (float 0))))

; create new vectori to hold given number of doubles
(defun new-vectori-double (x)
  (do ((v (new-vectori-long (* 2 x)))
       (n 0 (1+ n)))
      ((eq n x) v)
      (vseti-double v n (float 0))))

; access a float at offset ind of vectori vect
(defun vrefi-float (vect ind)
   (int:vref (maknum vect) (* 4 ind) 7))

; access a double at offset ind of vectori vect
(defun vrefi-double (vect ind)
   (int:vref (maknum vect) (* 8 ind) 8))

; at offset ind of vectori vect, assign val
(defun vseti-float (vect ind val)
   (int:vset (maknum vect) (* 4 ind) (float val) 7))

; at offset ind of vectori vect, assign val
(defun vseti-double (vect ind val)
   (int:vset (maknum vect) (* 8 ind) (float val) 8))

; use vsize to compute the number of floats in a vectori

; compute number of doubles in vectori vect
(defun vsize-double (vect)
   (int:vsize vect 3))
```

CHAPTER 19

Objects, Message Passing, and Flavors

19. Introduction

The object-oriented programming style used in the Smalltalk and Actor families of languages is available in FRANZ LISP. Its purpose is to perform *generic operations* on objects. Part of its implementation is simply a convention in procedure-calling style; part is a powerful language feature, called Flavors, for defining abstract objects. This chapter explains the principles of object-oriented programming and message passing, and the use of Flavors in implementing these in FRANZ LISP. It assumes no prior knowledge of any other languages.

This chapter is a heavily-edited version of Chapter 20 from the MIT LISP Machine Manual, as made available through MIT's Project Athena. It has been subsequently edited by the staff of Franz Inc. for inclusion in the FRANZ LISP manual. An entirely new and much more efficient implementation of Flavors is forthcoming from Franz Inc; this chapter documents the existing public-domain version.

In general, the Franz implementation of Flavors is quite similar to that in Zetalisp, although details and extensions may differ. Most code should port easily between versions.

19.1. Objects

When writing a program, it is often convenient to model what the program does in terms of *objects*, conceptual entities that can be likened to real-world things. Choosing what objects to provide in a program is very important to the proper organization of the program. In an object-oriented design, specifying what objects exist is the first task in designing the system. In a text editor, the objects might be "pieces of text", "pointers into text", and "display windows". In an electrical design system, the objects might be "resistors", "capacitors", "transistors", "wires", and "display windows". After specifying what objects there are, the next task of the design is to figure out what operations can be performed on each object. In the text editor example, operations on "pieces of text" might include inserting text and deleting text; operations on "pointers into text" might include moving forward and backward; and operations on "display windows" might include redisplaying the window and changing which "piece of text" the window is associated with.

In this model, we think of the program as being built around a set of objects, each of which has a set of operations that can be performed on it. More rigorously, the program defines several *types* of object (the editor above has three types), and it can create many *instances* of each type (that is, there can be many pieces of text, many pointers into text, and many windows). The program defines a set of types of object and, for each type, a set of operations that can be performed on any object of the type.

The new type abstractions may exist only in the programmer's mind. The mapping into a concrete representation may be done without the aid of any programming features. For example, it is possible to think of an atom's property list as an implementation of an abstract data type on which certain operations are defined, implemented in terms of the

LISP functions **get** and **putprop**. As another example, it is also possible to use "disembodied property lists". These are property lists (association lists of pairs) which are however not stored in the global structure of an atom. A disembodied property list has an odd number of items, the first being **nil**. **Get** and **putprop** still work. This type can be instantiated with **(cons nil nil)** (that is, by evaluating this form you can create a new disembodied property list); the operations are invoked through functions defined just for that purpose. The fact that disembodied property lists are really implemented as lists indistinguishable from any other lists, does not invalidate this point of view. However, such conceptual data types cannot be distinguished automatically by the system; one cannot ask "is this object a disembodied property list, as opposed to an ordinary list".

The **defstruct** for **ship** used as an example of the LISP deftype type-structure definitional capability in chapter 19 is another application. This is reviewed in section 19.2 following. **defstruct** automatically defines some operations on the objects: the operations to access its elements. We could define other functions that did useful computation with **ship**s, such as computing their speed, angle of travel, momentum, or velocity, stopping them, moving them elsewhere, and so on.

In both cases, we represent our conceptual object by one LISP object. The LISP object we use for the representation has *structure* and refers to other LISP objects. In the disembodied property list case, the LISP object is a list of pairs; in the **ship** case, the LISP object is an array whose details are taken care of by **defstruct**. In both cases, we can say that the object keeps track of an *internal state*, which can be *examined* and *altered* by the operations available for that type of object. **get** examines the state of a property list, and **putprop** alters it; **ship-x-position** examines the state of a ship, and **(setf (ship-x-position** *ship*) 5.0) alters it.

This is the essence of object-oriented programming. A conceptual object is modeled by a single LISP object, which bundles up some state information. For every type of object, there is a set of operations that can be performed to examine or alter the state of the object.

19.2. Modularity

An important benefit of the object-oriented style is that it lends itself to a particularly simple and lucid kind of modularity. If you have modular programming constructs and techniques available, they help and encourage you to write programs that are easy to read and understand, and so are more reliable and maintainable. Object-oriented programming lets a programmer implement a useful facility that presents the caller with a set of external interfaces, without requiring the caller to understand how the internal details of the implementation work. In other words, a program that calls this facility can treat the facility as a black box; the calling program has an implicit contract with the facility guaranteeing the external interfaces, and that is all it knows.

For example, a program that uses disembodied property lists never needs to know that the property list is being maintained as a list of alternating indicators and values; the program simply performs the operations, passing them inputs and getting back outputs. The program only depends on the external definition of these operations: it knows that if it **putprop**'s a property, and doesn't **remprop** it (or **putprop** over it), then it can do **get** and be sure of getting back the same thing it put in. This hiding of the details of the implementation means that someone reading a program that uses disembodied property lists need not concern himself with how they are implemented; he need only understand how what abstract operations are represented. This lets the programmer concentrate his energies on building a higher-level program rather than understanding the implementation of the support programs. This hiding of implementation means that the representation of property lists could be changed and the higher-level program would continue to work. For

example, instead of a list of alternating elements, the property list could be implemented as an association list or a hash table. Nothing in the calling program would change at all.

The same is true of the **ship** example. The caller is presented with a collection of operations, such as **ship-x-position**, **ship-y-position**, **ship-speed**, and **ship-direction**; it simply calls these and looks at their answers, without caring how they did what they did. In our example above, **ship-x-position** and **ship-y-position** would be accessor functions, defined automatically by **defstruct**, while **ship-speed** and **ship-direction** would be functions defined by the implementor of the **ship** type. The code might look like this:

```
(defstruct (ship :conc-name)
  x-position
  y-position
  x-velocity
  y-velocity
  mass)

(defun ship-speed (ship)
  (sqrt (+ (^ (ship-x-velocity ship) 2)
           (^ (ship-y-velocity ship) 2))))

(defun ship-direction (ship)
  (atan2 (ship-y-velocity ship)
         (ship-x-velocity ship)))
```

The caller need not know that the first two functions were structure accessors and that the second two were written by hand and perform arithmetic. Those facts would not be considered part of the black-box characteristics of the implementation of the **ship** type. The **ship** type does not guarantee which functions will be implemented in which ways; such aspects are not part of the contract between **ship** and its callers. In fact, **ship** could have been written this way instead:

```
(defstruct (ship :conc-name)
  x-position
  y-position
  speed
  direction
  mass)

(defun ship-x-velocity (ship)
  (* (ship-speed ship) (cos (ship-direction ship))))

(defun ship-y-velocity (ship)
  (* (ship-speed ship) (sin (ship-direction ship))))
```

In this second implementation of the **ship** type, we have decided to store the velocity in polar coordinates instead of rectangular coordinates. This is purely an implementation decision. The caller has no idea which of the two ways the implementation uses; he just performs the operations on the object by calling the appropriate functions.

We have now created our own types of objects, whose implementations are hidden from the programs that use them. Such types are usually referred to as *abstract types*. The object-oriented style of programming can be used to create abstract types by hiding the implementation of the operations and simply documenting what the operations are defined to do.

Some more terminology: the quantities being held by the elements of the **ship** structure are referred to as *instance variables*. Each instance of a type has the same operations defined on it; what distinguishes one instance from another (besides **eq**-ness) is the values that reside in its instance variables. The example above illustrates that a caller of operations does not know what the instance variables are; our two ways of writing the **ship** operations have different instance variables, but from the outside they have exactly the same operations.

One might ask: "But what if the caller evaluates (arrayref ship 2) and notices that he gets back the x-velocity rather than the speed? Then he can tell which of the two implementations were used." This is true; if the caller were to do that, he could tell. However, when a facility is implemented in the object-oriented style, only certain functions are documented and advertised, the functions that are considered to be operations on the type of object. The contract from **ship** to its callers only speaks about what happens if the caller calls these functions. The contract makes no guarantees at all about what would happen if the caller were to start poking around on his own using **arrayref**. A caller who does so *is in error*; he is depending on the concrete implementation of the abstraction: something that is not specified in the contract. No guarantees were ever made about the results of such action, and so anything may happen; indeed, if **ship** were reimplemented, the code that does the **arrayref** might have a different effect entirely and probably stop working. This example shows why the concept of a contract between a callee and a caller is important: the contract specifies the interface between the two modules.

Unlike some other languages that provide abstract types, FRANZ LISP makes no attempt to have the language automatically forbid constructs that circumvent the contract. This is intentional. One reason for this is that LISP is an interactive system, and so it is important to be able to examine and alter internal state interactively (usually from a debugger). Furthermore, there is no strong distinction between the "system" and the "user" portions of the FRANZ LISP system; users are allowed to get into nearly any part of the language system and change what they want to change.

In summary: by defining a set of operations and making only a specific set of external entry-points available to the caller, the programmer can create his own abstract types. These types can be useful facilities for other programs and programmers. Since the implementation of the type is hidden from the callers, modularity is maintained and the implementation can be changed easily.

We have hidden the implementation of an abstract type by making its operations into functions which the user may call. The important of the concept is not that they are functions--in LISP everything is done with functions. The important point is that we have defined a new conceptual operation and given it a name, rather than requiring each user who wants to do the operation to write it out step-by-step. Thus we say **(ship-x-velocity s)** rather than **(arrayref s 2)**.

Often a few abstract operation functions are simple enough that it is desirable to compile special code for them rather than really calling the function. (Compiling special code like this is often called *open-coding*.) The compiler is directed to do this through use of macros for example. **defstruct** arranges for this kind of special compilation for the functions that get the instance variables of a structure.

When we use this optimization, the implementation of the abstract type is only hidden in a certain sense. It does not appear in the LISP code written by the user, but does appear in the compiled code. The reason is that there may be some compiled functions that use the macros (or other concrete manifestation of the implementation). Even if you

change the definition of the macro, the existing compiled code will continue to use the old definition. Thus, if the implementation of a module is changed, programs that use it may need to be recompiled. This sacrifice of compatibility between interpreted and compiled code is usually quite acceptable for the sake of efficiency in debugged code.

In the FRANZ LISP implementation of Flavors which is discussed below, there is no such compiler incorporation of nonmodular knowledge into a program, except when the **:ordered-instance-variables** feature is used; this is explained in the section on "ordered-instance-variables-option". If you don't use the **:ordered-instance-variables** feature, you don't have to worry about this.

19.3. Generic Operations

Consider the rest of the program that uses the **ship** abstraction. It may want to deal with other objects that are like **ship**s in that they are movable objects with mass, but unlike **ship**s in other ways. A more advanced model of a ship might include the concept of the ship's engine power, the number of passengers on board, and its name. An object representing a meteor probably would not have any of these, but might have another attribute such as how much iron is in it.

However, all kinds of movable objects have positions, velocities, and masses, and the system will contain some programs that deal with these quantities in a uniform way, regardless of what kind of object is being modeled. For example, a piece of the system that calculates every object's orbit in space need not worry about the other, more peripheral attributes of various types of objects; it works the same way for all objects. Unfortunately, a program that tries to calculate the orbit of a ship needs to know the ship's attributes, and must therefore call **ship-x-position** and **ship-y-velocity** and so on. The problem is that these functions won't work for meteors. There would have to be a second program to calculate orbits for meteors that would be exactly the same, except that where the first one calls **ship-x-position**, the second one would call **meteor-x-position**, and so on. This would be very bad; a great deal of code would have to exist in multiple copies, all of it would have to be maintained in parallel, and it would take up space for no good reason.

What is needed is an operation that can be performed on objects of several different types. For each type, it should do the thing appropriate for that type. Such operations are called *generic* operations. The classic example of generic operations is the arithmetic functions in many programming languages, including FRANZ LISP. The **plus** function accepts integers, floats or bignums; (and in Common Lisp compatibility mode, + accepts ratios and complex numbers), and perform an appropriate kind of addition, based on the data types of the objects being manipulated. In Macsyma, an algebraic manipulation system written in FRANZ LISP, the + operation works for matrices, polynomials, rational functions, and arbitrary algebraic expression trees. In our example, we need a generic **x-position** operation that can be performed on either **ship**'s, **meteor**'s, or any other kind of mobile object represented in the system. This way, we can write a single program to calculate orbits. When it wants to know the x position of the object it is dealing with, it simply invokes the generic **x-position** operation on the object, and whatever type of object it has, the correct operation is performed, and the x position is returned.

In the following discussion we use another idiom adopted from the Smalltalk language: performing a generic operation is called *sending a message*. The message consists of an operation name (a symbol) and arguments. The objects in the program are thought of as little people, who get sent messages and respond with answers (returned values). In the example above, the objects are sent **x-position** messages, to which they respond with their x position.

Sending a message is a way of invoking a function without specifying which function is to be called. Instead, the data determines the function to use. The caller specifies an operation name and an object; that is, it said what operation to perform, and what object to perform it on. The function to invoke is found from this information.

The two data used to figure out which function to call are the *type* of the object, and the *name* of the operation. The same set of functions are used for all instances of a given type, so the type is the only attribute of the object used to figure out which function to call. The rest of the message besides the operation is data which are passed as arguments to the function, so the operation is the only part of the message used to find the function. Such a function is called a *method.* For example, if we send an **x-position** message to an object of type **ship**, then the function we find is "the **ship** type's **x-position** method". A method is a function that handles a specific operation on a specific kind of object; this method handles messages named **x-position** to objects of type **ship**.

In our new terminology: the orbit-calculating program finds the x position of the object it is working on by sending that object a message consisting of the operation **x-position** and no arguments. The returned value of the message is the x position of the object. If the object was of type **ship**, then the **ship** type's **x-position** method was invoked; if it was of type **meteor**, then the **meteor** type's **x-position** method was invoked. The orbit-calculating program just sends the message, and the right function is invoked based on the type of the object. We now have true generic functions, in the form of message passing: the same operation can mean different things depending on the type of the object.

19.4. Generic Operations in LISP

How do we implement message passing in LISP? Our convention is that objects that receive messages are always *functional* objects (that is, you can apply them to arguments). A message is sent to an object by calling that object as a function, passing the operation name as the first argument and the arguments of the message as the rest of the arguments. Operation names are represented by symbols; normally these symbols are installed in the keyword package, if your system includes the Common LISP- compatibility "package" system. The principal operational difference is that with the package system, you may omit certain quotes. See the example below. So if we have a variable **my-ship** whose value is an object of type **ship**, and we want to know its x position, we send it a message as follows:

```
(send my-ship ':x-position)    ;; no packages
```

If you have an installed package system as in Franz Opus 42.09 and later (as we assume for future examples) you may use the unquoted symbol as in:

```
(send my-ship :x-position)    ;; package system installed
```

To set the ship's x position to **3.0**, we send it a message like this:

```
(send my-ship :set-x-position 3.0)
```

A variation supported in some Flavor systems would allow

```
(send my-ship :set :x-position 3.0) ;; not supported
```

but this is not provided in FRANZ LISP.

It should be stressed that no new features are added to LISP for message sending; we simply define a convention on the way objects take arguments. The convention says that an object accepts messages by always interpreting its first argument as an operation name. The object must consider this operation name, find the function which is the method for that operation, and invoke that function.

To emphasize the relationship between well-known features and the new object-oriented version, we define the two basic functions for message passing as follows:

(send 's_object 's_operation [arguments]**)**

NOTE: Same as
funcall.

Conceptually, this sends *s_object* a message with operation and arguments as specified. The function **send** is preferable to **funcall**, when a message is being sent, since it reminds the programmer of the usage.

In some implementations of Flavors, the semantics of *send* may differ from *funcall* in those cases where *s_object* is a symbol, list, number, or other object that does not normally handle messages.

(lexpr-send)

NOTE: Same as
apply.

How does this all work? The object must somehow find the right method for the message it is sent. Furthermore, the object now has to be callable as a function. However,an ordinary function will not do: we need a data structure that can store the instance variables (the internal state) of the object. Of the FRANZ LISP features available,
the most appropriate is the closure. A message-receiving object could be implemented as a closure over a set of instance variables. The function inside the closure would have a big **selectq** form to dispatch on its first argument.

While using closures as given does work, it has several problems. The main problem is that in order to add a new operation to a system, it is necessary to modify code in more than one place: you have to find all the types that "understand" that operation, and add a new clause to the **selectq**. The problem with this is that you cannot textually separate the implementation of your new operation from the rest of the system: the methods must be interleaved with the other operations for the type. Adding a new operation should only

require *adding* LISP code; it should not require *modifying* LISP code.

For example, the conventional way of making generic operations for arithmetic on various new mathematical objects is to have a procedure for each operation (+, *, etc), which has a big **selectq** for all the types; this means you have to modify code in (generic-plus, generic-times, ...) to add a type. This is inconvenient and error-prone.

The *flavor* mechanism is a streamlined, more convenient, and time-tested system for creating message-receiving objects. With flavors, you can add a new method simply by adding code, without modifying existing code. Furthermore, many common and useful things are very easy to do with flavors. The rest of this chapter describes flavors.

19.5. Simple Use of Flavors

A *flavor*, in its simplest form, is a definition of an abstract type. New flavors are created with the **defflavor** special form, and methods of the flavor are created with the **defmethod** special form. New instances of a flavor are created with the **make-instance** function. This section explains simple uses of these forms.

For an example of a simple use of flavors, here is how the **ship** example above would be implemented.

```
(defflavor ship (x-position y-position
                 x-velocity y-velocity mass)
                ()
 :gettable-instance-variables)

(defmethod (ship :speed) ()
  (sqrt (+ (^ x-velocity 2)
           (^ y-velocity 2))))

(defmethod (ship :direction) ()
  (atan2 y-velocity x-velocity))
```

The code above creates a new flavor. The first subform of the **defflavor** is **ship**, which is the name of the new flavor. Next is the list of instance variables; they are the five that should be familiar by now. The next subform is something we will get to later. The rest of the subforms are the body of the **defflavor**, and each one specifies an option about this flavor. In our example, there is only one option, namely **:gettable-instance-variables**. This means that for each instance variable, a method should automatically be generated to return the value of that instance variable. The name of the operation is a symbol with the same name as the instance variable, but interned on the keyword package. Thus, methods are created to handle the operations **:x-position**, **:y-position**, and so on.

Each of the two **defmethod** forms adds a method to the flavor. The first one adds a handler to the flavor **ship** for the operation **:speed**. The second subform is the lambda-list, and the rest is the body of the function that handles the **:speed** operation. The body can refer to or set any instance variables of the flavor, just like variables bound by a containing **let**. When any instance of the **ship** flavor is invoked with a first argument of **:direction**, the body of the second **defmethod** is evaluated in an environment in which the instance variables of **ship** refer to the instance variables of this instance (the one to which the message was sent). So the arguments passed to **atan** are the the velocity

components of this particular ship. The result of **atan** becomes the value returned by the **:direction** operation.

Now we have seen how to create a new abstract type: a new flavor. Every instance of this flavor has the five instance variables named in the **defflavor** form, and the seven methods we have seen (five that were automatically generated because of the **:gettable-instance-variables** option, and two that we wrote ourselves). The way to create an instance of our new flavor is with the **make-instance** function. Here is how it could be used:

```
(setq my-ship (make-instance 'ship))
```

This returns an object whose printed representation is something like #<SHIP 13731210>. (the details of the print form will vary; it is an object which cannot be read back in from the short-hand representation). The argument to **make-instance** is the name of the flavor to be instantiated. Additional arguments, not used here, are *init options*, that is, commands to the flavor of which we are making an instance, selecting optional features. This will be discussed more in a moment.

Examination of the flavor we have defined shows that it is quite useless as it stands, since there is no way to set any of the parameters. We can fix this up easily by putting the **:settable-instance-variables** option into the **defflavor** form. This option tells **defflavor** to generate methods for operation **:set** for first argument **:x-position**, **:y-position**, and so on. Each such method takes one additional argument and sets the corresponding instance variable to that value. It also generates methods for the operations **:set-x-position**, **:set-y-position** and so on. Each of these takes one argument and sets the corresponding variable.

Another option we can add to the **defflavor** is **:inittable-instance-variables**, (alternative spelling for compatibility is **:initable-instance-variables**) which allows us to initialize the values of the instance variables when an instance is first created. **:inittable-instance-variables** does not create any methods; instead, it makes *initialization keywords* named **:x-position**, **:y-position**, etc., that can be used as init-option arguments to **make-instance** to initialize the corresponding instance variables. The list of init options is sometimes called the *init-plist* because it is like a property list.

Here is the improved **defflavor**:

```
(defflavor ship (x-position y-position
                 x-velocity y-velocity mass)
                ()
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables)
```

All we have to do is evaluate this new **defflavor**, and the existing flavor definition is updated and now includes the new methods and initialization options. In fact, the instance we generated a while ago now accepts the new operations! We can set the mass of the ship we created by evaluating

```
(send my-ship :set-mass 3.0)
```

and the **mass** instance variable of **my-ship** is properly set to **3.0**. Whether you use **:set-mass** or the general operation **:set** is a matter of style; **:set** is used by the expansion of **(setf (send my-ship :mass) 3.0)**.

[IMPORTANT REMINDER: IF YOU ARE USING EARLIER VERSIONS OF THIS SYSTEM IT WILL BE NECESSARY FOR YOU TO 'QUOTE' THE ATOMS IN THE MESSAGES: e.g. (send my-ship ':set-mass 3.0)]

If you want to play around with flavors, it is useful to know that **describe** of an instance tells you the flavor of the instance and the values of its instance variables. If we were to evaluate **(describe my-ship)** at this point, the following would be printed:

```
#<ship 3214320>, an object of flavor ship,
 has instance variable values:
        x-position:     nil
        y-position:     nil
        x-velocity:     nil
        y-velocity:     nil
        mass:           3.0
```

Now that the instance variables are *inittable*, we can create another ship and initialize some of the instance variables using the init-plist. Let's do that and **describe** the result:

```
=> (setq her-ship (make-instance 'ship :x-position 0.0 :y-position 2.0 :mass 3.5))

#<SHIP 3242340>

=> (describe her-ship)
#<SHIP 3242340>, an object of flavor ship,
 has instance variable values:
        x-position:     0.0
        y-position:     2.0
        x-velocity:     nil
        y-velocity:     nil
        mass:           3.5
```

A flavor can also establish default initial values for instance variables. These default values are used when a new instance is created if the values are not initialized any other way. The syntax for specifying a default initial value is to replace the name of the instance variable by a list, whose first element is the name and whose second is a form to evaluate to produce the default initial value. For example when read in the definitions:

```
(defvar *default-x-velocity* 2.0)
(defvar *default-y-velocity* 3.0)

(defflavor ship ((x-position 0.0)
                 (y-position 0.0)
                 (x-velocity *default-x-velocity*)
                 (y-velocity *default-y-velocity*)
                 mass)
                ()
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables)
```

Then the system works as follows:

```
=> (setq another-ship (make-instance 'ship :x-position 3.4))
#<SHIP 2342340>
=> (describe another-ship)
#<SHIP 2342340>
an object of flavor ship,
 has instance variable values:
        x-position:     3.4
        y-position:     0.0
        x-velocity:     2.0
        y-velocity:     3.0
        mass:           nil
```

The value of **x-position** was initialized explicitly, so the default was ignored. The value of **y-position** was initialized from the default value, which was **0.0**. The two velocity instance variables were initialized from their default values, which came from two global variables. The value of **mass** was not explicitly initialized and did not have a default initialization, so it was left as nil. Some flavor implementations use a special value "void" rather than nil.

There are many other options that can be used in **defflavor**, and the init options can be used more flexibly than just to initialize instance variables; full details are given later in this chapter. But even with the small set of features we have seen so far, it is easy to write object-oriented programs.

19.6. Mixing Flavors

Now we have a system for defining message-receiving objects so that we can have generic operations. If we want to create a new type called **meteor** that would accept the same generic operations as **ship**, we could simply write another **defflavor** and two more **defmethod**'s that looked just like those of **ship**, and then meteors and ships would both accept the same operations. Objects of type **ship** would have some more instance variables for holding attributes specific to ships and some more methods for operations that are not generic, but are only defined for ships; the same would be true of **meteor**.

However, this would be a a wasteful thing to do. The same code has to be repeated in several places, and several instance variables have to be repeated. The code now needs to be maintained in many places, which is always undesirable. The power of flavors (and the name "flavors") comes from the ability to mix several flavors and get a new flavor. Since the functionality of **ship** and **meteor** partially overlap, we can take the common functionality and move it into its own flavor, which might be called **moving-object**. We would define **moving-object** the same way as we defined **ship** in the previous section. Then, **ship** and **meteor** could be defined like this:

```
(defflavor ship (engine-power number-of-passengers name)
          (moving-object)
  :gettable-instance-variables)

(defflavor meteor (percent-iron)
                  (moving-object)
  :inittable-instance-variables)
```

These **defflavor** forms use the second subform, for which we previously used (). The second subform is a list of flavors to be combined to form the new flavor; such flavors are called *components*. Concentrating on **ship** for a moment (analogous statements are true of **meteor**), we see that it has exactly one component flavor: **moving-object**. It also has a list of instance variables, which includes only the ship-specific instance variables and not the ones that it shares with **meteor**. By incorporating **moving-object**, the **ship** flavor acquires all of its instance variables, and so need not name them again. It also acquires all of **moving-object**'s methods, too. So with the new definition, **ship** instances still implement the **:x-velocity** and **:speed** operations, with the same meaning as before. However, the **:engine-power** operation is also understood (and returns the value of the **engine-power** instance variable).

What we have done here is to take an abstract type, **moving-object**, and build two more specialized and powerful abstract types on top of it. Any ship or meteor can do anything a moving object can do, and each also has its own specific abilities. This kind of building can continue; we could define a flavor called **ship-with-passenger** that was built on top of **ship**, and it would inherit all of **moving-object**'s instance variables and methods as well as **ship**'s instance variables and methods. Furthermore, the second subform of **defflavor** can be a list of several components, meaning that the new flavor should combine all the instance variables and methods of all the flavors in the list, as well as the ones *those* flavors are built on, and so on. All the components taken together form a big tree of flavors. A flavor is built from its components, its components' components, and so on. We sometimes use the term "components" to mean the immediate components (the ones listed in the **defflavor**), and sometimes to mean all the components (including the components of the immediate components and so on). (Actually, it is not strictly a tree, since some flavors might be components through more than one path. It is really a directed graph; it can even be cyclic.)

The order in which the components are combined to form a flavor is important. The tree of flavors is turned into an ordered list by performing a *top-down, depth-first* walk of the tree, including non-terminal nodes *before* the subtrees they head, ignoring any flavor that has been encountered previously somewhere else in the tree. For example, if **flavor-1**'s immediate components are **flavor-2** and **flavor-3**, and **flavor-2**'s components are **flavor-4** and **flavor-5**, and **flavor-3**'s component was **flavor-4**, then the complete list of components of **flavor-1** would be: (flavor-1, flavor-2, flavor-4, flavor-5, flavor-3) The flavors earlier in this list are the more specific, less basic ones; in our example, **ship-with-**

passengers would be first in the list, followed by **ship**, followed by **moving-object**. A flavor is always the first in the list of its own components. Notice that **flavor-4** does not appear twice in this list. Only the first occurrence of a flavor appears; duplicates are removed. (The elimination of duplicates is done during the walk; if there is a cycle in the directed graph, it does not cause a non-terminating computation.)

The set of instance variables for the new flavor is the union of all the sets of instance variables in all the component flavors. If both **flavor-2** and **flavor-3** have instance variables named **foo**, then **flavor-1** has an instance variable named **foo**, and any methods that refer to **foo** refer to this same instance variable. Thus different components of a flavor can communicate with one another using shared instance variables. (Typically, only one component ever sets the variable; the others only look at it.) The default initial value for an instance variable comes from the first component flavor to specify one.

The way the methods of the components are combined is the heart of the flavor system. When a flavor is defined, a single function, called a *combined method*, is constructed for each operation supported by the flavor. This function is constructed out of all the methods for that operation from all the components of the flavor. There are many different ways that methods can be combined; these can be selected by the user when a flavor is defined. The user can also create new forms of combination.

There are several kinds of methods, but so far, the only kinds of methods we have seen are *primary* methods. The default way primary methods are combined is that all but the earliest one provided are ignored. In other words, the combined method is simply the primary method of the first flavor to provide a primary method. What this means is that if you are starting with a flavor **foo** and building a flavor **bar** on top of it, then you can override **foo**'s method for an operation by providing your own method. Your method will be called, and **foo**'s will never be called.

Simple overriding is often useful; for example, if you want to make a new flavor **bar** that is just like **foo** except that it reacts completely differently to a few operations. However, often you don't want to completely override the base flavor's (**foo**'s) method; sometimes you want to add some extra things to be done. This is where combination of methods is used.

The usual way methods are combined is that one flavor provides a primary method, and other flavors provide *daemon methods*. The idea is that the primary method is "in charge" of the main business of handling the operation, but other flavors just want to keep informed that the message was sent, or just want to do the part of the operation associated with their own area of responsibility.

daemon methods come in two kinds, *before* and *after*. There is a special syntax in **defmethod** for defining such methods. Here is an example of the syntax. To give the **ship** flavor an after-daemon method for the **:speed** operation, the following syntax would be used:

(defmethod (ship :after :speed) () *body*)

Now, when a message is sent, it is handled by a new function called the *combined* method. The combined method first calls all of the before daemons, then the primary method, then all the after daemons. Each method is passed the same arguments that the combined method was given. The returned values from the combined method are the values returned by the primary method; any values returned from the daemons are ignored. Before-daemons are called in the order that flavors are combined, while after-

daemons are called in the reverse order. In other words, if you build **bar** on top of **foo**, then **bar**'s before-daemons run before any of those in **foo**, and **bar**'s after-daemons run after any of those in **foo**.

The reason for this order is to keep the modularity order correct. If we create **flavor-1** built on **flavor-2**, then the components of **flavor-2** should not matter. Our new before-daemons go before all methods of **flavor-2**, and our new after-daemons go after all methods of **flavor-2**. Note that if you have no daemons, this reduces to the form of combination described above. The most recently added component flavor is the highest level of abstraction; you build a higher-level object on top of a lower-level object by adding new components to the front. The syntax for defining daemon methods can be found in the description of **defmethod** below.

To make this a bit more clear, let's consider a simple example that is easy to play with: the **:print-self** method. The LISP printer (i.e. the **print** function) prints instances of flavors by sending them **:print-self** messages. The first argument to the **:print-self** operation is a port (we can ignore the others for now), and the receiver of the message is supposed to print its printed representation on the port. In the **ship** example above, the reason that instances of the **ship** flavor printed the way they did is because the **ship** flavor was actually built on top of a very basic flavor called **vanilla-flavor**; this component is provided automatically by **defflavor**. It was **vanilla-flavor**'s **:print-self** method that was doing the printing. Now, if we give **ship** its own primary method for the **:print-self** operation, then that method completely takes over the job of printing: **vanilla-flavor**'s method will not be called at all. However, if we give **ship** a before-daemon method for the **:print-self** operation, then it will get invoked before the **vanilla-flavor** method, and so whatever it prints will appear before what **vanilla-flavor** prints. So we can use before-daemons to add prefixes to a printed representation; similarly, after-daemons can add suffixes.

There are other ways to combine methods besides daemons, but this way is the most common. The more advanced ways of combining methods are explained in a later section. The details of **vanilla-flavor** and what it does for you are also explained later.

19.7. Flavor Functions

We've been using this special form informally:

(**defflavor** flavor-name ([var]..) ([flav]...) [options] ...))

WHERE: *flavor-name* is a symbol which serves to name this flavor.

var's are the names of the instance-variables containing the local state for this flavor. A list of two elements: the name of an instance-variable and a default initialization form is also acceptable; the initialization form is evaluated when an instance of the flavor is created if no other initial value for the variable is obtained. If no initialization is specified, the variable has value nil.

flav's are the names of the component flavors out of which this flavor is built. The features of those flavors are inherited as described previously.

opt's are options; each option may be either a keyword symbol or a list of a keyword symbol and arguments. The options to **defflavor** are described under defflavor-options.

SIDE EFFECT: The symbol flavor-name is given a flavor property which is the internal data-structure containing the details of the flavor.

NOTE: Objects which are instances of flavors are FRANZ LISP vectors. More detail can be obtained with the function :typep, below.

(:typep 's_item 's_flavor**)**

RETURNS: t if the sybol s_item is an instance of s_flavor, nil otherwise.

NOTE: The name of this function is of historical origin.

```
(:typep my-ship 'ship)
=> t
(type my-ship)
=> vector
```

An important and useful global variable is available: A list of the names of all the flavors that have ever been **defflavor**'ed.

(defmethod (flavor-name method-type operation) lambda-list [form]...**))**

WHERE: *flavor-name* is a symbol which is the name of the flavor which is to receive the method. *operation* is a keyword symbol which names the operation to be handled. *method-type* is a keyword symbol for the type of method; it is omitted when you are defining a primary method. For some method-types, additional information is expected. It comes after *operation*.

SIDE EFFECT: Defmethod defines a method, that is, a function to handle a particular operation for instances of a particular flavor. The meaning of *method-type* depends on what style of method combination is declared for this operation. For instance, if **:daemon** combination (the default style) is in use, method types **:before** and **:after** are allowed. See section 19.11 for a complete description of the way methods are combined.

lambda-list describes the arguments and aux variables of the function. The first argument to the method, which is the operation name itself, is automatically handled and so is not included in the lambda-list. Note that methods may not have unevaluated (**&quote**) arguments; that is, they must be functions, not special forms. *forms*, are the function body; the value of the last form is returned when the method is applied.

The variant form

(defmethod (*flavor-name operation*) *function*)

---

where *function* is a symbol, says that *flavor-name*'s method for *operation* is *function*, a symbol which names a function. That function must take appropriate arguments; the first argument is the operation. When the function is called, **self** will be bound.

If you redefine a method that is already defined, the old definition is replaced by the new one. Given a flavor, an operation name, and a method type, there can only be one function (with the exception of **:case** methods) so if you define a **:before** daemon method

for the **foo** flavor to handle the **:bar** operation, then you replace the previous before-daemon; however, you do not affect the primary method or methods of any other type, operation or flavor.

This is useful to know if you want to trace a method, or if you want to poke around at the method function itself.

(make-instance flavor-name [init-option value]... **)**

RETURNS: an instance of the specified flavor which has just been created.

NOTE: Arguments after the first are alternating init-option keywords and arguments to those keywords. These options are used to initialize instance variables and to select arbitrary options, as described above. An **:init** message is sent to the newly-created object with one argument, the init-plist. This is a disembodied property-list containing the init-options specified and those defaulted from the flavor's **:default-init-plist** (however, init keywords that simply initialize instance variables, and the corresponding values, may be absent when the **:init** methods are called). **make-instance** is an easy-to-call interface to **instantiate-flavor**, below.

If **:allow-other-keys** is used as an init keyword with a non-**nil** value, this error check is suppressed. Then unrecognized keywords are simply ignored. Example:

```
(make-instance 'foo :lose 5 :allow-other-keys t)
```

specifies the init keyword **:lose**, but prevents an error should the keyword not be handled.

(instantiate-flavor flavor-name init-plist [send-init-message-p return-unhandled-keywords area]**)**

RETURNS: an instance.

NOTE: This is an extended version of **make-instance**, giving you more features. Note that it takes the *init-plist* as an individual argument, rather than taking a rest argument of init options and values.

The *init-plist* argument must be a disembodied property list, Beware! This property list can be modified; the properties from the default init plist are **putprop**'ed on if not already present, and some **:init** methods do explicit **putprop**'s onto the *init-plist*.

In the event that **:init** methods **remprop** properties already on the *init-plist* (as opposed to simply doing **get** and **putprop**), then the *init-plist* is **rplacd**'ed. This means that the actual supplied list of options is modified. It also means that the caller of **instantiate-flavor** should copy its init-plist argument (e.g. with *append*).

Do not use **nil** as the *init-plist* argument. This would mean to use the properties of the symbol **nil** as the init options. If your goal is to have no init options, you must provide a property list containing no properties, such as the list **(nil)**.

Here is the sequence of actions by which **instantiate-flavor** creates a new instance:

First, the specified flavor's instantiation flavor function if it exists, is called to determine which flavor should actually be instantiated. If there is no instantiation flavor function, the specified flavor is instantiated.

If the flavor's method hash-table and other internal information have not been computed or are not up to date, they are computed. This may take a substantial amount of time or even invoke the compiler, but it happens only once for each time you define or redefine a

particular flavor.

Next, the instance itself is created. The *area* argument is irrelevant to FRANZ LISP but refers to consing in specified areas, a feature used in some Lisp machines.

Then the initial values of the instance variables are computed. If an instance variable is declared inittable, and a keyword with the same spelling as its name appears in *init-plist*, the property for that keyword is used as the initial value.

Otherwise, if the default init plist specifies such a property, it is evaluated and the value is used. Otherwise, if the flavor definition specifies a default initialization form, it is evaluated and the value is used. The initialization form may not refer to any instance variables. It can find the new instance in **self** but should not invoke any operations on it and should not refer directly to any instance variables. It can get at instance variables using accessor macros created by the **:outside-accessible-instance-variables** option or the function **symeval-in-instance**.

If an instance variable does not get initialized either of these ways it is left nil; an **:init** method may initialize it (see below).

All remaining keywords and values specified in the **:default-init-plist** option to **defflavor**, that do not initialize instance variables and are not overridden by anything explicitly specified in *init-plist* are then merged into *init-plist* using **putprop**. The default init plist of the instantiated flavor is considered first, followed by those of all the component flavors in the standard order.

Then keywords appearing in the *init-plist* but not defined with the **:init-keywords** option or the **:inittable-instance-variables** option for some component flavor are collected. If the **:allow-other-keys** option is specified with a non-**nil** value (either in the original *init-plist* argument or by some default init plist) then these *unhandled* keywords are ignored. If the *return-unhandled-keywords* argument is non-**nil**, a list of these keywords is returned as the second value of **instantiate-flavor**. Otherwise, an error is signaled if any unrecognized init keywords are present.

If the *send-init-message-p* argument is supplied and non-**nil**, an **:init** message is sent to the newly-created instance, with one argument, the *init-plist*. **get** can be used to extract options from this property-list. Each flavor that needs initialization can contribute an **:init** method by defining a daemon.

The **:init** methods should not look on the *init-plist* for keywords that simply initialize instance variables (that is, keywords defined with **:inittable-instance-variables** rather than **:init-keywords**). The corresponding instance variables are already set up when the **:init** methods are called, and sometimes the keywords and their values may actually be missing from the *init-plist* if it is more efficient not to put them on. To avoid problems, always refer to the instance variables themselves rather than looking for the init keywords that initialize them.

(**:init** init-plist)
This operation is implemented on all flavor instances.

SIDE EFFECT: This function examines the init keywords and perform whatever initializations are appropriate. *init-plist* is the argument that was given to **instantiate-flavor**, and may be passed directly to **get** to examine the value of any particular init option.

The default definition of this operation does nothing. However, many flavors add **:before** and **:after** daemons to it.

(**instancep** object)

RETURNS: t if **object** is an instance of a flavor.

(**defwrapper** lambda-list macro-body-arg)

NOTE: This is complex and you may not be able to follow the description until you have tried to use flavors.

Sometimes the way the flavor system combines the methods of different flavors (the daemon system) is not powerful enough. In that case **defwrapper** can be used to define a macro that expands into code that is wrapped around the invocation of the methods. This is best explained by an example; suppose you needed a lock locked during the processing of the **:foo** operation on flavor **bar**, which takes two arguments, and you have a **lock-frobboz** special-form that knows how to lock the lock (presumably it generates an **unwind-protect**). **lock-frobboz** needs to see the first argument to the operation; perhaps that tells it what sort of operation is going to be performed (read or write).

```
(defwrapper (bar :foo) ((arg1 arg2) . body)
  `(lock-frobboz (self arg1)
     . ,body))
```

The use of the **body** macro-argument prevents the macro defined by **defwrapper** from knowing the exact implementation and allows several **defwrapper**'s from different flavors to be combined properly.

Note that the argument variables, **arg1** and **arg2**, are not referenced with commas before them. These may look like **defmacro** "argument" variables, but they are not. Those variables are not bound at the time the **defwrapper**-defined macro is expanded and the back-quoting is done; rather the result of that macro-expansion and back-quoting is code which, when a message is sent, will bind those variables to the arguments in the message as local variables of the combined method.

Consider another example. Suppose you thought you wanted a **:before** daemon, but found that if the argument was **nil** you needed to return from processing the message immediately, without executing the primary method. You could write a wrapper such as

```
(defwrapper (bar :foo) ((arg1) . body)
  '(cond ((null arg1))
         (t (print "About to do :FOO")
           . ,body)))
```

Suppose you need a variable for communication among the daemons for a particular operation; perhaps the **:after** daemons need to know what the primary method did, and it is something that cannot be easily deduced from just the arguments. You might use an instance variable for this, or you might create a special variable which is bound during the processing of the operation and used free by the methods.

```
(defvar *communication*)
(defwrapper (bar :foo) (ignore . body)
  '(let ((*communication* nil))
     . ,body))
```

Similarly you might want a wrapper that puts a **catch** around the processing of an operation so that any one of the methods could throw out in the event of an unexpected condition.

Like daemon methods, wrappers work in outside-in order; when you add a **defwrapper** to a flavor built on other flavors, the new wrapper is placed outside any wrappers of the component flavors. However, *all* wrappers happen before *any* daemons happen. When the combined method is built, the calls to the before-daemon methods, primary methods, and after-daemon methods are all placed together, and then the wrappers are wrapped around them. Thus, if a component flavor defines a wrapper, methods added by new flavors execute within that wrapper's context.

:around methods can do some of the same things that wrappers can. If one flavor defines both a wrapper and an **:around** method for the same operation, the **:around** method is executed inside the wrapper.

Be careful about inserting the body into an internal lambda-expression within the wrapper's code. Doing so interacts with the internals of the flavor system and requires knowledge of things not documented in the manual in order to work properly. It is much simpler to use an **:around** method instead.

(undefflavor 'flavor)
Undefines flavor *flavor*. All methods of the flavor are lost. *flavor* and all flavors that depend on it are no longer valid to instantiate.

If instances of the discarded definition exist, they continue to use that definition. When a message is sent to an object, the variable **self** is automatically bound to that object, for the benefit of methods which want to manipulate the object itself (as opposed to its instance variables).

(send s_instance 's_message [argument]...)
(funcall s_instance 's_message [argument]...)

NOTE: This is the way a message is passed to an instance of a flavor. **send** and **funcall** operate in essentially the same manner.

(send-self 's_message [argument]...)
(funcall-self 's_message [argument]...)

(lexpr-send-self message arguments.. list-of-arguments)
(lexpr-funcall-self operation arguments... list-of-arguments)
funcall-self is nearly equivalent to **funcall** with **self** as the first argument, but **funcall-self** is a little faster. The others are analogous.

(with-self-variables-bound body)
Within the body of this special form, all of **self**'s instance variables are bound as specials to the values inside **self**. (Normally this is true only of those instance variables that are specified in **:special-instance-variables** when **self**'s flavor was defined.) As a result, inside the body you can use **set**, **boundp** and **symeval**, etc., freely on the instance variables of **self**.

(recompile-flavor flavor-name [single-operation (use-old-combined-methods t) (do-dependents t)])
Updates the internal data of the flavor and any flavors that depend on it. If *single-operation* is supplied non-**nil**, only the methods for that operation are changed. The system does this when you define a new method that did not previously exist. If *use-old-combined-methods* is **t**, then the existing combined method functions are used if possible. New ones are generated only if the set of methods to be called has changed. This is the default. If *use-old-combined-methods* is **nil**, automatically-generated functions to call multiple methods or to contain code generated by wrappers are regenerated unconditionally. If *do-dependents* is **nil**, only the specific flavor you specified is recompiled. Normally all flavors that depend on it are also recompiled.

recompile-flavor affects only flavors that have already been compiled. Typically this means it affects flavors that have been instantiated, but does not bother with mixins. If this variable is non-**nil**, automatic recompilation of combined methods is turned off.

If you wish to make several changes each of which will cause recompilation of the same combined methods, you can use this variable to speed things up by making the recompilations happen only once. Set the variable to **t**, make your changes, and then set the variable back to **nil**. Then use **recompile-flavor** to recompile whichever combined methods need it.

NOTE: compile-flavor-methods and related functions are currently unimplemented in FRANZ LISP.

(get-handler-for 'object 'operation)
Given an object and an operation, this returns the object's method for that operation, or **nil** if it has none. When *object* is an instance of a flavor, this function can be useful to find which of that flavor's components supplies the method.

This is related to the **:handler** function specification. It is preferable to use the generic operation **:get-handler-for**.

(flavor-allows-init-keyword-p 'flavor-name 'keyword)

RETURNS: non-**nil** if the flavor named *flavor-name* allows *keyword* in the init options when it is instantiated, or **nil** if it does not. The non-**nil** value is the name of the component flavor that contributes the support of that keyword.

si:flavor-all-allowed-init-keywords flavor-name

RETURNS: a list of all the init keywords that may be used in instantiating *flavor-name*.

symeval-in-instance 'instance 'symbol ['no-error-p]

RETURNS: the value of the instance variable *symbol* inside *instance*. If there is no such instance variable, an error is signaled, unless *no-error-p* is non-**nil** in which case **nil** is returned.

set-in-instance 'instance 'symbol 'value

SIDE EFFECT: Sets the value of the instance variable *symbol* inside *instance* to *value*. If there is no such instance variable, an error is signalled.

describe-flavor flavor-name

SIDE EFFECT: Prints descriptive information about a flavor; it is self-explanatory. An important thing it tells you that can be hard to figure out yourself is the combined list of component flavors; this list is what is printed after the phrase 'and directly or indirectly depends on'.

Whenever a flavor instance is sent a message whose operation it does not handle, an error is signalled.

19.8. Defflavor Options

There are quite a few options to **defflavor**. They are all described here, although some are for very specialized purposes and not of interest to most users. Each option can be written in two forms; either the keyword by itself, or a list of the keyword and arguments to that keyword.

Several of these options declare things about instance variables. These options can be given with arguments which are instance variables, or without any arguments in which case they refer to all of the instance variables listed at the top of the **defflavor**. This is *not* necessarily all the instance variables of the component flavors, just the ones mentioned in this flavor's **defflavor**. When arguments are given, they must be instance variables that were listed at the top of the **defflavor**; otherwise they are assumed to be misspelled and an error is signalled. It is legal to declare things about instance variables inherited from a component flavor, but to do so you must list these instance variables explicitly in the instance variable list at the top of the **defflavor**.

:gettable-instance-variables

Enables automatic generation of methods for getting the values of instance variables. The operation name is the name of the variable, in the keyword package (i.e. it has a colon in front of it).

Note that there is nothing special about these methods; you could easily define them yourself. This option generates them automatically to save you the trouble of

writing out a lot of very simple method definitions. (The same is true of methods defined by the **:settable-instance-variables** option.) If you define a method for the same operation name as one of the automatically generated methods, the explicit definition overrides the automatic one.

:settable-instance-variables

Enables automatic generation of methods for setting the values of instance variables. The operation name is '**:set-**' followed by the name of the variable. All settable instance variables are also automatically made gettable and inittable. (See the note in the description of the **:gettable-instance-variables** option, above.)

In addition, **:case** methods are generated for the **:set** operation with suboperations taken from the names of the variables, so that **:set** can be used to set them.

:inittable-instance-variables

The instance variables listed as arguments, or all instance variables listed in this **defflavor** if the keyword is given alone, are made *inittable*. This means that they can be initialized through use of a keyword (a colon followed by the name of the variable) as an init-option argument to **make-instance**.

:special-instance-variables

The instance variables listed as arguments, or all instance variables listed in this **defflavor** if the keyword is given alone, will be bound dynamically when handling messages. (By default, instance variables are bound lexically with the scope being the method.) You must do this to any instance variables that you wish to be accessible through **symeval**, **set**, **boundp** and **makunbound**, since they see only dynamic bindings.

This should also be done for any instance variables that are declared globally special. If you omit this, the flavor system does it for you automatically when you instantiate the flavor, and gives you a warning to remind you to fix the **defflavor**.

NOTE: This option has no effect in Franz, as all instance variables are implicitly special in both interpreted and compiled code.

:init-keywords

The arguments are declared to be valid keywords to use in **instantiate-flavor** when creating an instance of this flavor (or any flavor containing it). The system uses this for error-checking: before the system sends the **:init** message, it makes sure that all the keywords in the init-plist are either inittable instance variables or elements of this list. If any is not recognized, an error is signalled. When you write a **:init** method that accepts some keywords, they should be listed in the **:init-keywords** option of the flavor.

If **:allow-other-keys** is used as an init keyword with a non-**nil** value, this error check is suppressed. Then unrecognized keywords are simply ignored.

:default-init-plist

The arguments are alternating keywords and value forms, like a property list. When the flavor is instantiated, these properties and values are put into the init-plist unless already present. This allows one component flavor to default an option to another component flavor. The value forms are only evaluated when and if they are used. For example,

```
(:default-init-plist :frob-array
                     (make-array 100))
```

would provide a default "frob array" for any instance for which the user did not provide one explicitly.

```
(:default-init-plist :allow-other-keys t)
```

prevents errors for unhandled init keywords in all instantiation of this flavor and other flavors that depend on it.

:required-init-keywords defflavor
: The arguments are init keywords which are to be required each time this flavor (or any flavor containing it) is instantiated. An error is signalled if any required init keyword is missing.

:required-instance-variables
: Declares that any flavor incorporating this one that is instantiated into an object must contain the specified instance variables. An error occurs if there is an attempt to instantiate a flavor that incorporates this one if it does not have these in its set of instance variables. Note that this option is not one of those that checks the spelling of its arguments in the way described at the start of this section (if it did, it would be useless).

  Required instance variables may be freely accessed by methods just like normal instance variables. The difference between listing instance variables here and listing them at the front of the **defflavor** is that the latter declares that this flavor "owns" those variables and accepts responsibility for initializing them, while the former declares that this flavor depends on those variables but that some other flavor must be provided to manage them and whatever features they imply.

:required-methods
: The arguments are names of operations that any flavor incorporating this one must handle. An error occurs if there is an attempt to instantiate such a flavor and it is lacking a method for one of these operations. Typically this option appears in the **defflavor** for a base flavor. Usually this is used when a base flavor does a **(send self ...)** to send itself a message that is not handled by the base flavor itself; the idea is that the base flavor will not be instantiated alone, but only with other components (mixins) that do handle the message. This keyword allows the error of having no handler for the message to be detected when the flavor instantiated or when **compile-flavor-methods** is done, rather than when the missing operation is used.

:required-flavors
: The arguments are names of flavors that any flavor incorporating this one must

include as components, directly or indirectly. The difference between declaring flavors as required and listing them directly as components at the top of the **defflavor** is that declaring flavors to be required does not make any commitments about where those flavors will appear in the ordered list of components; that is left up to whoever does specify them as components. The purpose of declaring a flavor to be required is to allow instance variables declared by that flavor to be accessed. It also provides error checking: an attempt to instantiate a flavor that does not include the required flavors as components signals an error. Compare this with **:required-methods** and **:required-instance-variables**.

For an example of the use of required flavors, consider the **ship** example given earlier, and suppose we want to define a **relativity-mixin** which increases the mass dependent on the speed. We might write,

```
(defflavor relativity-mixin () (moving-object))
(defmethod (relativity-mixin :mass) ()
  (// mass (sqrt (- 1 (^ (// (send self :speed)
                             *speed-of-light*)
                          2)))))
```

but this would lose because any flavor that had **relativity-mixin** as a component would get **moving-object** right after it in its component list. As a base flavor, **moving-object** should be last in the list of components so that other components mixed in can replace its methods and so that daemon methods combine in the right order. **relativity-mixin** has no business changing the order in which flavors are combined, which should be under the control of its caller. For example,

```
(defflavor starship ()
          (relativity-mixin long-distance-mixin ship))
```

puts **moving-object** last (inheriting it from **ship**).

So instead of the definition above we write,

```
(defflavor relativity-mixin () ()
        (:required-flavors moving-object))
```

which allows **relativity-mixin**'s methods to access **moving-object** instance variables such as **mass** (the rest mass), but does not specify any place for **moving-object** in the list of components.

It is very common to specify the *base flavor* of a mixin with the **:required-flavors** option in this way.

:included-flavors

The arguments are names of flavors to be included in this flavor. The difference between declaring flavors here and declaring them at the top of the **defflavor** is that when component flavors are combined, if an included flavor is not specified as a normal component, it is inserted into the list of components immediately after the last component to include it. Thus included flavors act like defaults. The important thing is that if an included flavor *is* specified as a component, its position in the list of components is completely controlled by that specification, independently of where the flavor that includes it appears in the list.

:included-flavors and **:required-flavors** are used in similar ways; it would have been reasonable to use **:included-flavors** in the **relativity-mixin** example above. The difference is that when a flavor is required but not given as a normal component, an error is signalled, but when a flavor is included but not given as a normal component, it is automatically inserted into the list of components at a reasonable place.

:no-vanilla-flavor

Normally when a flavor is instantiated, the special flavor **si:vanilla-flavor** is included automatically at the end of its list of components. The vanilla flavor provides some default methods for the standard operations which all objects are supposed to understand. These include **:print-self**, **:describe**, **:which-operations**, and several other operations.

If any component of a flavor specifies the **:no-vanilla-flavor** option, then **si:vanilla-flavor** is not included in that flavor. This option should not be used casually.

:default-handler

The argument is the name of a function that is to be called to handle any operation for which there is no method. Its arguments are the arguments of the **send** which invoked the operation, including the operation name as the first argument. Whatever values the default handler returns are the values of the operation.

Default handlers can be inherited from component flavors. If a flavor has no default handler, any operation for which there is no method signals a **sys:unclaimed-message** error.

:ordered-instance-variables

This option is mostly for esoteric internal system uses. The arguments are names of instance variables which must appear first (and in this order) in all instances of this flavor, or any flavor depending on this flavor. This is used for instance variables that are specially known about by microcode, and also in connection with the **:outside-accessible-instance-variables** option. If the keyword is given alone, the arguments default to the list of instance variables given at the top of this **defflavor**.

Removing any of the **:ordered-instance-variables**, or changing their positions in the list, requires that you recompile all methods that use any of the affected instance variables.

:outside-accessible-instance-variables

The arguments are instance variables which are to be accessible from outside of this flavor's methods. A macro (actually a subst) is defined which takes an object of this flavor as an argument and returns the value of the instance variable; **setf** may be used to set the value of the instance variable. The name of the macro is the name of the flavor concatenated with a hyphen and the name of the instance variable. These macros are similar to the accessor macros created by **defstruct**

This feature works in two different ways, depending on whether the instance variable has been declared to have a fixed slot in all instances, via the **:ordered-instance-**

variables option.

If the variable is not ordered, the position of its value cell in the instance must be computed at run time. This takes noticeable time, although less than actually sending a message would take. An error is signalled if the argument to the accessor macro is not an instance or is an instance that does not have an instance variable with the appropriate name. However, there is no error check that the flavor of the instance is the flavor the accessor macro was defined for, or a flavor built upon that flavor. This error check would be too expensive.

If the variable is ordered, the compiler compiles a call to the accessor macro into a subprimitive which simply accesses that variable's assigned slot by number. This subprimitive is only three or four times slower than **car**. The only error-checking performed is to make sure that the argument is really an instance and is really big enough to contain that slot. There is no check that the accessed slot really belongs to an instance variable of the appropriate name.

NOTE: In Franz, *setf* does not work on these functions.

:abstract-flavor

This option marks the flavor as one that is not supposed to be instantiated (that is, is supposed to be used only as a component of other flavors). An attempt to instantiate the flavor signals an error.

It is sometimes useful to do **compile-flavor-methods** on a flavor that is not going to be instantiated, if the combined methods for this flavor will be inherited and shared by many others. **:abstract-flavor** tells **compile-flavor-methods** not to complain about missing required flavors, methods or instance variables. Presumably the flavors that depend on this one and actually are instantiated will supply what is lacking.

NOTE: **:abstract-flavor** is not implemented in FRANZ LISP.

:method-combination

Specifies the method combination style to be used for certain operations. Each argument to this option is a list (*style order operation1 operation2...*). *operation1*, *operation2*, etc. are names of operations whose methods are to be combined in the declared fashion. *style* is a keyword that specifies a style of combination. *order* is a keyword whose interpretation is up to *style*; typically it is either **:base-flavor-first** or **:base-flavor-last**.

Any component of a flavor may specify the type of method combination to be used for a particular operation. If no component specifies a style of method combination, then the default style is used, namely **:daemon**. If more than one component of a flavor specifies the combination style for a given operation, then they must agree on the specification, or else an error is signalled.

:run-time-alternatives defflavor

:mixture defflavor

A run-time-alternative flavor defines a collection of similar flavors, all built on the same base flavor but having various mixins as well. Instantiation chooses a flavor of the collection at run time based on the init keywords specified, using an automatically generated instantiation flavor function.

A simple example would be

```
(defflavor foo () (basic-foo)
  (:run-time-alternatives
    (:big big-foo-mixin))
  (:init-keywords :big))
```

Then **(make-instance 'foo :big t)** makes an instance of a flavor whose components are **big-foo-mixin** as well as **foo**. But **(make-instance 'foo)** or **(make-instance 'foo :big nil)** makes an instance of **foo** itself. The clause **(:big big-foo-mixin)** in the **:run-time-alternatives** says to incorporate **big-foo-mixin** if **:big**'s value is **t**, but not if it is **nil**.

There may be several clauses in the **:run-time-alternatives**. Each one is processed independently. Thus, two keywords **:big** and **:wide** could independently control two mixins, giving four possibilities.

```
(defflavor foo () (basic-foo)
  (:run-time-alternatives
    (:big big-foo-mixin)
    (:wide wide-foo-mixin))
  (:init-keywords :big))
```

It is possible to test for values other than **t** and **nil**. The clause

```
(:size (:big big-foo-mixin)
       (:small small-foo-mixin)
       (nil nil))
```

allows the value for the keyword **:size** to be **:big**, **:small** or **nil** (or omitted). If it is **nil** or omitted, no mixin is used (that's what the second **nil** means). If it is **:big** or **:small**, an appropriate mixin is used. This kind of clause is distinguished from the simpler kind by having a list as its second element. The values to check for can be anything, but **eq** is used to compare them.

The value of one keyword can control the interpretation of others by nesting clauses within clauses. If an alternative has more than two elements, the additional elements are subclauses which are considered only if that alternative is selected. For example, the clause

```
(:etherial (t etherial-mixin)
           (nil nil
                (:size (:big big-foo-mixin)
                       (:small small-foo-mixin)
                       (nil nil))))
```

says to consider the **:size** keyword only if **:etherial** is **nil**.

:mixture is synonymous with **:run-time-alternatives**. It exists for compatibility with Zetalisp or other Lisp Machine systems.

:documentation
Specifies the documentation string for the flavor definition, which is made accessible through **(documentation** *flavorname* 'flavor).

This documentation can be viewed with the **describe-flavor** function.

19.9. Flavor Families

The following organization conventions are recommended for programs that use flavors.

A *base flavor* is a flavor that defines a whole family of related flavors, all of which have that base flavor as a component. Typically the base flavor includes things relevant to the whole family, such as instance variables, **:required-methods** and **:required-instance-variables** declarations, default methods for certain operations, **:method-combination** declarations, and documentation on the general protocols and conventions of the family. Some base flavors are complete and can be instantiated, but most are cannot be instantiated themselves. They serve as a base upon which to build other flavors. The base flavor for the *foo* family is often named **basic-***foo*.

A *mixin flavor* is a flavor that defines one particular feature of an object. A mixin cannot be instantiated, because it is not a complete description. Each module or feature of a program is defined as a separate mixin; a usable flavor can be constructed by choosing the mixins for the desired characteristics and combining them, along with the appropriate base flavor. By organizing your flavors this way, you keep separate features in separate flavors, and you can pick and choose among them. Sometimes the order of combining mixins does not matter, but often it does, because the order of flavor combination controls the order in which daemons are invoked and wrappers are wrapped. Such order dependencies should be documented as part of the conventions of the appropriate family of flavors. A mixin flavor that provides the *mumble* feature is often named *mumble*-**mixin**.

If you are writing a program that uses someone else's facility, using that facility's flavors and methods, your program may still define its own flavors, in a simple way. The facility provides a base flavor and a set of mixins: the caller can combine these in various ways depending on exactly what it wants, since the facility probably does not provide all possible useful combinations. Even if your private flavor has exactly the same components as a pre-existing flavor, it can still be useful since you can use its **:default-init-plist** to select options of its component flavors and you can define one or two methods to customize it "just a little".

19.10. Vanilla Flavor

The operations described in this section are a standard protocol, which all message-receiving objects are assumed to understand. The standard methods that implement this protocol are automatically supplied by the flavor system unless the user specifically tells it not to do so. These methods are associated with the flavor **si:vanilla-flavor**:

si:vanilla-flavor
: Unless you specify otherwise (with the **:no-vanilla-flavor** option to **defflavor**), every flavor includes the "vanilla" flavor, which has no instance variables but provides some basic useful methods.

:print-self stream prindepth escape-p
: The object should output its printed-representation to a stream. The printer sends this message when it encounters an instance or an entity. The arguments are the stream, the current depth in list-structure (for comparison with **prinlevel**), and whether escaping is enabled (a copy of the value of ***print-escape***. **si:vanilla-flavor** ignores the last two arguments and prints something like #<*flavor-name octal-address*>. The *flavor-name* tells you what type of object it is and the *octal-address* allows you to tell different objects apart.

:describe
: The object should describe itself, printing a description onto the ***standard-output*** stream. The **describe** function sends this message when it encounters an instance. **si:vanilla-flavor** outputs in a reasonable format the object, the name of its flavor, and the names and values of its instance-variables.

:set keyword value
: The object should set the internal value specified by *keyword* to the new value *value*. For flavor instances, the **:set** operation uses **:case** method combination, and a method is generated automatically to set each settable instance variable, with *keyword* being the variable's name as a keyword.

NOTE: This option is currently unimplemented in FRANZ LISP.

:which-operations
: The object should return a list of the operations it can handle. **si:vanilla-flavor** generates the list once per flavor and remembers it, minimizing consing and compute-time. If the set of operations handled is changed, this list is regenerated the next time someone asks for it.

:operation-handled-p operation
: *operation* is an operation name. The object should return **t** if it has a handler for the specified operation, **nil** if it does not.

:get-handler-for operation
: *operation* is an operation name. The object should return the method it uses to handle *operation*. If it has no handler for that operation, it should return **nil**. This is like the **get-handler-for** function, but, of course, you can use it only on objects known to accept messages.

:send-if-handles operation [arguments]...
: *operation* is an operation name and *arguments* is a list of arguments for the operation. If the object handles the operation, it should send itself a message with that operation and arguments, and return whatever values that message returns. If it doesn't handle the operation it should just return **nil**.

(:eval-inside-yourself form)
The argument is a form that is evaluated in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. It works to **setq** one of these special variables; the instance variable is modified. This is intended to be used mainly for debugging.

(:funcall-inside-yourself function &rest args)
function is applied to *args* in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. It works to **setq** one of these special variables; the instance variable is modified. This is a way of allowing callers to provide actions to be performed in an environment set up by the instance.

(:break)
break is called in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables.

19.11. Method Combination

When a flavor has or inherits more than one method for an operation, they must be called in a specific sequence. The flavor system creates a function called a *combined method* which calls all the user-specified methods in the proper order. Invocation of the operation actually calls the combined method, which is responsible for calling the others.

For example, if the flavor **foo** has components and methods as follows:

```
(defflavor foo () (foo-mixin foo-base))
(defflavor foo-mixin () (bar-mixin))

(defmethod (foo :before :hack) ...)
(defmethod (foo :after :hack) ...)

(defmethod (foo-mixin :before :hack) ...)
(defmethod (foo-mixin :after :hack) ...)

(defmethod (bar-mixin :before :hack) ...)
(defmethod (bar-mixin :hack) ...)

(defmethod (foo-base :hack) ...)
(defmethod (foo-base :after :hack) ...)
```

then the combined method generated looks like this (ignoring many important details not related to this issue):

```
(defmethod (foo :combined :hack) (&rest args)
  (apply #'(:method foo :before :hack) args)
  (apply #'(:method foo-mixin :before :hack) args)
  (apply #'(:method bar-mixin :before :hack) args)
  (multiple-value-prog1
      (apply #'(:method bar-mixin :hack) args)
    (apply #'(:method foo-base :after :hack) args)
    (apply #'(:method foo-mixin :after :hack) args)
    (apply #'(:method foo :after :hack) args)))
```

This example shows the default style of method combination, the one described in the introductory parts of this chapter, called **:daemon** combination. Each style of method combination defines which *method types* it allows, and what they mean. **:daemon** combination accepts method types **:before** and **:after**, in addition to *untyped* methods; then it creates a combined method which calls all the **:before** methods, only one of the untyped methods, and then all the **:after** methods, returning the value of the untyped method. The combined method is constructed by a function much like a macro's expander function, and the precise technique used to create the combined method is what gives **:before** and **:after** their meaning.

Note that the **:before** methods are called in the order **foo**, **foo-mixin**, **bar-mixin** and **foo-base**. (**foo-base** does not have a **:before** method, but if it had one that one would be last.) This is the standard ordering of the components of the flavor **foo** (see SEE ALSO(flavor-components)); since it puts the base flavor last, it is called **:base-flavor-last** ordering. The **:after** methods are called in the opposite order, in which the base flavor comes first. This is called **:base-flavor-first** ordering.

Only one of the untyped methods is used; it is the one that comes first in **:base-flavor-last** ordering. An untyped method used in this way is called a *primary* method.

Other styles of method combination define their own method types and have their own ways of combining them. Use of another style of method combination is requested with the **:method-combination** option to **defflavor**. Here is an example which uses **:list** method combination, a style of combination that allows **:list** methods and untyped methods:

```
(defflavor foo () (foo-mixin foo-base))
(defflavor foo-mixin () (bar-mixin))
(defflavor foo-base () ()
  (:method-combination (:list :base-flavor-last :win)))

(defmethod (foo :list :win) ...)
(defmethod (foo :win) ...)

(defmethod (foo-mixin :list :win) ...)

(defmethod (bar-mixin :list :win) ...)
(defmethod (bar-mixin :win) ...)

(defmethod (foo-base :win) ...)
```

yielding the combined method

```
(defmethod (foo :combined :win) (&rest args)
  (list
    (apply #'(:method foo :list :win) args)
    (apply #'(:method foo-mixin :list :win) args)
    (apply #'(:method bar-mixin :list :win) args)
    (apply #'(:method foo :win) args)
    (apply #'(:method bar-mixin :win) args)
    (apply #'(:method foo-base :win) args)))
```

The **:method-combination** option in the **defflavor** for **foo-base** causes **:list** method combination to be used for the **:win** operation on all flavors that have **foo-base** as a component, including **foo**. The result is a combined method which calls all the methods, including all the untyped methods rather than just one, and makes a list of the values they return. All the **:list** methods are called first, followed by all the untyped methods; and within each type, the **:base-flavor-last** ordering is used as specified. If the **:method-combination** option said **:base-flavor-first**, the relative order of the **:list** methods would be reversed, and so would the untyped methods, but the **:list** methods would still be called before the untyped ones. **:base-flavor-last** is more often right, since it means that **foo**'s own methods are called first and **si:vanilla-flavor**'s methods (if it has any) are called last.

A few specific method types, such as **:default** and **:around**, have standard meanings independent of the style of method combination, and can be used with any style. They are described in a table below.

Here are the standardly defined method combination styles.

:daemon
: The default style of method combination. All the **:before** methods are called, then the primary (untyped) method for the outermost flavor that has one is called, then all the **:after** methods are called. The value returned is the value of the primary method.

:daemon-with-or
: Like the **:daemon** method combination style, except that the primary method is wrapped in an **:or** special form with all **:or** methods. Multiple values can be returned from the primary method, but not from the **:or** methods (as in the **or** special form). This produces code like the following in combined methods:

```
(progn (foo-before-method)
       (multiple-value-prog1
         (or (foo-or-method)
             (foo-primary-method))
         (foo-after-method)))
```

This is useful primarily for flavors in which a mixin introduces an alternative to the primary method. Each **:or** method gets a chance to run before the primary method and to decide whether the primary method should be run or not; if any **:or** method returns a non-**nil** value, the primary method is not run (nor are the rest of the **:or** methods). Note that the ordering of the combination of the **:or** methods is controlled by the *order* keyword in the **:method-combination** option.

:daemon-with-and
: Like **:daemon-with-or** except that it combines **:and** methods in an **and** special form. The primary method is run only if all of the **:and** methods return non-**nil** values.

:daemon-with-override
: Like the **:daemon** method combination style, except an **or** special form is wrapped around the entire combined method with all **:override** typed methods before the combined method. This differs from **:daemon-with-or** in that the **:before** and **:after** daemons are run only if *none* of the **:override** methods returns non-**nil**. The combined method looks something like this:

```
(or (foo-override-method)
    (progn (foo-before-method)
           (foo-primary-method)
           (foo-after-method)))
```

:progn
: Calls all the methods inside a **progn** special form. Only untyped and **:progn** methods are allowed. The combined method calls all the **:progn** methods and then all the untyped methods. The result of the combined method is whatever the last of the methods returns.

:or
: Calls all the methods inside an **or** special form. This means that each of the methods is called in turn. Only untyped methods and **:or** methods are allowed; the **:or** methods are called first. If a method returns a non-**nil** value, that value is returned and none of the rest of the methods are called; otherwise, the next method is called. In other words, each method is given a chance to handle the message; if it doesn't want to handle the message, it can return **nil**, and the next method gets a chance to try.

:and
: Calls all the methods inside an **and** special form. Only untyped methods and **:and** methods are allowed. The basic idea is much like **:or**; see above.

:append
Calls all the methods and appends the values together. Only untyped methods and **:append** methods are allowed; the **:append** methods are called first.

:nconc
Calls all the methods and **nconc**'s the values together. Only untyped methods and **:nconc** methods are allowed, etc.

:list Calls all the methods and returns a list of their returned values. Only untyped methods and **:list** methods are allowed, etc.

:inverse-list
Calls each method with one argument; these arguments are successive elements of the list that is the sole argument to the operation. Returns no particular value. Only untyped methods and **:inverse-list** methods are allowed, etc.

If the result of a **:list**-combined operation is sent back with an **:inverse-list**-combined operation, with the same ordering and with corresponding method definitions, each component flavor receives the value that came from that flavor.

:pass-on
Calls each method on the values returned by the preceding one. The values returned by the combined method are those of the outermost call. The format of the declaration in the **defflavor** is:

(:method-combination (:pass-on (*ordering* . *arglist*))
. *operation-names*)

---

where *ordering* is **:base-flavor-first** or **:base-flavor-last**. *arglist* may include the **&aux** and **&optional** keywords.

Only untyped methods and **:pass-on** methods are allowed. The **:pass-on** methods are called first.

:case With **:case** method combination, the combined method automatically does a **selectq** dispatch on the first argument of the operation, known as the *suboperation*. Methods of type **:case** can be used, and each one specifies one suboperation that it applies to. If no **:case** method matches the suboperation, the primary method, if any, is called.

Example:
```
(defflavor foo (a b) ()
  (:method-combination (:case :base-flavor-last :win)))

This method handles (send a-foo :win :a):
(defmethod (foo :case :win :a) ()
  a)

This method handles (send a-foo :win :a*b):
(defmethod (foo :case :win :a*b) ()
  (* a b))

This method handles (send a-foo :win :something-else):
(defmethod (foo :win) (suboperation)
  (list 'something-random suboperation))
```

:case methods are unusual in that one flavor can have many **:case** methods for the same operation, as long as they are for different suboperations.

The suboperations **:which-operations**, **:operation-handled-p**, **:send-if-handles** and **:get-handler-for** are all handled automatically based on the collection of **:case** methods that are present.

Methods of type **:or** are also allowed. They are called just before the primary method, and if one of them returns a non-**nil** value, that is the value of the operation, and no more methods are called.

NOTE: **:case** method combination is currently unimplemented in FRANZ LISP.

Here is a list of all the method types recognized by the standard styles of method combination.

(no *If no type is given to* **defmethod**, a primary method is created. This is the most common type of method.

:before

:after These are used for the before-daemon and after-daemon methods used by **:daemon** method combination.

:default
If there are no untyped methods among any of the flavors being combined, then the **:default** methods (if any) are treated as if they were untyped. If there are any untyped methods, the **:default** methods are ignored.

Typically a base-flavor defines some default methods for certain of the operations understood by its family. When using the default kind of method combination these default methods are suppressed if another component provides a primary method.

:or, :and
These are used for **:daemon-with-or** and **:daemon-with-and** method combination. The **:or** methods are wrapped in an **or**, or the **:and** methods are wrapped in an **and**, together with the primary method, between the **:before** and **:after** methods.

:override
Allows the features of **:or** method combination to be used together with daemons. If you specify **:daemon-with-override** method combination, you may use **:override** methods. The **:override** methods are executed first, until one of them returns non-**nil**. If this happens, that method's value(s) are returned and no more methods are used. If all the **:override** methods return **nil**, the **:before**, primary and **:after** methods are executed as usual.

In typical usages of this feature, the **:override** method usually returns **nil** and does nothing, but in exceptional circumstances it takes over the handling of the operation.

:or, :and, :progn, :list, :inverse-list, pass-on, :append, :nconc.
Each of these methods types is allowed in the method combination style of the same name. In those method combination styles, these typed methods work just like untyped ones, but all the typed methods are called before all the untyped ones. These method types can be used with any method combination style; they have standard meanings independent of the method combination style being used.

:wrapper
This is used internally by **defwrapper**.

:combined
This is used internally for automatically-generated *combined* methods.

The most common form of combination is **:daemon**. One thing may not be clear: when do you use a **:before** daemon and when do you use an **:after** daemon? In some cases the primary method performs a clearly-defined action and the choice is obvious: **:before :launch-rocket** puts in the fuel, and **:after :launch-rocket** turns on the radar tracking.

In other cases the choice can be less obvious. Consider the **:init** message, which is sent to a newly-created object. To decide what kind of daemon to use, we observe the order in which daemon methods are called. First the **:before** daemon of the instantiated flavor is called, then **:before** daemons of successively more basic flavors are called, and finally the **:before** daemon (if any) of the base flavor is called. Then the primary method is called. After that, the **:after** daemon for the base flavor is called, followed by the **:after** daemons at successively less basic flavors.

Now, if there is no interaction among all these methods, if their actions are completely independent, then it doesn't matter whether you use a **:before** daemon or an **:after** daemon. There is a difference if there is some interaction. The interaction we are talking about is usually done through instance variables; in general, instance variables are how the methods of different component flavors communicate with each other. In the case of the **:init** operation, the *init-plist* can be used as well. The important thing to remember is that no method knows beforehand which other flavors have been mixed in to form this flavor; a method cannot make any assumptions about how this flavor has been combined, and in what order the various components are mixed.

This means that when a **:before** daemon has run, it must assume that none of the methods for this operation have run yet. But the **:after** daemon knows that the **:before** daemon for each of the other flavors has run. So if one flavor wants to convey information to the other, the first one should "transmit" the information in a **:before** daemon, and the second one should "receive" it in an **:after** daemon. So while the **:before** daemons are run, information is "transmitted"; that is, instance variables get set up. Then, when the **:after** daemons are run, they can look at the

instance variables and act on their values.

In the case of the **:init** method, the **:before** daemons typically set up instance variables of the object based on the init-plist, while the **:after** daemons actually do things, relying on the fact that all of the instance variables have been initialized by the time they are called.

The problems become most difficult when you are creating a network of instances of various flavors that are supposed to point to each other. For example, suppose you have flavors for "buffers" and "streams", and each buffer should be accompanied by a stream. If you create the stream in the **:before :init** method for buffers, you can inform the stream of its corresponding buffer with an init keyword, but the stream may try sending messages back to the buffer, which is not yet ready to be used. If you create the stream in the **:after :init** method for buffers, there will be no problem with stream creation, but some other **:after :init** methods of other mixins may have run and made the assumption that there is to be no stream. The only way to guarantee success is to create the stream in a **:before** method and inform it of its associated buffer by sending it a message from the buffer's **:after :init** method. This scheme--creating associated objects in **:before** methods but linking them up in **:after** methods--often avoids problems, because all the various associated objects used by various mixins at least exist when it is time to make other objects point to them.

Since flavors are not hierarchically organized, the notion of levels of abstraction is not rigidly applicable. However, it remains a useful way of thinking about systems.

19.12. Implementation of Flavors

An object that is an instance of a flavor is implemented as a closure whose associated function is a message-dispatch function. The variables closed new are **self, .own-flavor.** (containing the name of the flavor), and the instance variables. The flavor name can be used to find an instance-descriptor, which is a **defstruct** that appears on the **si:flavor** property of the flavor name. It contains, among other things, the name of the flavor, the size of an instance, the table of methods for handling operations, and information for accessing the instance variables.

defflavor creates such a data structure for each flavor, and links them together according to the dependency relationships between flavors.

A message is sent to an instance simply by calling it as a function, with the first argument being the operation. The evaluator binds **self** to the object and binds those instance variables that are supposed to be special to the value cells in the instance. Then it passes on the operation and arguments to a funcallable hash table taken from the flavor-structure for this flavor.

When the funcallable hash table is called as a function, it hashes the first argument (the operation) to find a function to handle the operation. If there is only one method to be invoked, this function is that method; otherwise it is an automatically-generated function called the combined method. which calls the appropriate methods in the right order. If there are wrappers, they are incorporated into this combined method. A consequence of the implementation of flavors as closures in Franz is that all instance variables are implicitly special, whether or not they are so declared, and this holds for both interpreted and compiled code. Flavor methods operate compiled, as well as interpreted, although compiling does not get around the overhead of making the closure bindings for each message.

19.12.1. Order of Definition

There is a certain amount of freedom to the order in which you do **defflavor**'s, **defmethod**'s, and **defwrapper**'s. This freedom is designed to make it easy to load programs containing complex flavor structures without having to do things in a certain order. It is considered important that not all the methods for a flavor need be defined in the same file. Thus the partitioning of a program into files can be along modular lines.

The rules for the order of definition are as follows.

Before a method can be defined (with **defmethod** or **defwrapper**) its flavor must have been defined (with **defflavor**). This makes sense because the system has to have a place to remember the method, and because it has to know the instance-variables of the flavor if the method is to be compiled.

When a flavor is defined (with **defflavor**) it is not necessary that all of its component flavors be defined already. This is to allow **defflavor**'s to be spread between files according to the modularity of a program, and to provide for mutually-dependent flavors. Methods can be defined for a flavor some of whose component flavors are not yet defined; however, in certain cases compiling those methods may produce a warning that an instance variable was declared special (because the system did not realize it was an instance variable). If this happens, you should fix the problem and recompile.

The methods automatically generated by the **:gettable-instance-variables** and **:settable-instance-variables defflavor** options are generated at the time the **defflavor** is done.

The first time a flavor is instantiated, or when **compile-flavor-methods** is done, the system looks through all of the component flavors and gathers various information. At this point an error is signaled if not all of the components have been **defflavor**'ed. This is also the time at which certain other errors are detected, for instance lack of a required instance-variable (see the **:required-instance-variables defflavor** option. The combined methods are generated at this time also, unless they already exist.

After a flavor has been instantiated, it is possible to make changes to it. Such changes affect all existing instances if possible. This is described more fully immediately below.

19.12.2. Changing a Flavor

You can change anything about a flavor at any time. You can change the flavor's general attributes by doing another **defflavor** with the same name. You can add or modify methods by doing **defmethod**'s. If you do a **defmethod** with the same flavor-name, operation (and suboperation if any), and (optional) method-type as an existing method, that method is replaced by the new definition.

These changes always propagate to all flavors that depend upon the changed flavor. Normally the system propagates the changes to all existing instances of the changed flavor and its dependent flavors. However, this is not possible when the flavor has been changed so drastically that the old instances would not work properly with the new flavor. This happens if you change the number of instance variables, which changes the size of an instance. It also happens if you change the order of the instance variables (and hence the storage layout of an instance), or if you change the component flavors (which can change several subtle aspects of an instance). The system does not

keep a list of all the instances of each flavor, so it cannot find the instances and modify them to conform to the new flavor definition. Instead it gives you a warning message, on the ***error-output*** stream, to the effect that the flavor was changed incompatibly and the old instances will not get the new version. The system leaves the old flavor data-structure intact (the old instances continue to point at it) and makes a new one to contain the new version of the flavor. If a less drastic change is made, the system modifies the original flavor data-structure, thus affecting the old instances that point at it. However, if you redefine methods in such a way that they only work for the new version of the flavor, then trying to use those methods with the old instances won't work.

19.13. Property List Operations

It is often useful to associate a property list with an abstract object, for the same reasons that it is useful to have a property list associated with a symbol. This section describes a mixin flavor that can be used as a component of any new flavor in order to provide that new flavor with a property list. For more details and examples, see the general discussion of property lists. The usual property list functions (**get**, **putprop**, etc.) are not implemented by sending the instance the corresponding message.

The mixin flavor **si:property-list-mixin** provides the basic operations on property lists:

(si:property-list-mixin :get property-name &optional default)
Looks up the object's *property-name* property. If it finds such a property, it returns the value; otherwise it returns *default*.

(si:property-list-mixin :getl)
Like the **:get** operation, except that the argument is a list of property names. The **:getl** operation searches down the property list until it finds a property whose property name is one of the elements of *property-name-list*. It returns the portion of the property list begining with the first such property that it found. If it doesn't find any, it returns **nil**.

(si:property-list-mixin :putprop value property-name)
Gives the object a *property-name* property of *value*.

(send *object* :set :get *property-name value*)

---

also has this effect.

(si:property-list-mixin :remprop property-name)

RETURNS: It returns one of the cells spliced out, whose car is the former value of the property that was just removed. If there was no such property to begin with, the value is **nil**.

SIDE EFFECT: Removes the object's *property-name* property, by splicing it out of the property list.

(si:property-list-mixin :get-location-or-nil property-name)
(si:property-list-mixin :get-location property-name)

RETURNS: a locative pointer to the cell in which this object's *property-name* property is stored.

NOTE: If there is no such property, **:get-location-or-nil** returns **nil**, but **:get-location** adds a cell to the property list and initialized to **nil**, and a pointer to that cell is returned.

(si:property-list-mixin :push-property)

WHERE: the *property-name* property of the object should be a list (note that **nil** is a list and an absent property is **nil**). This operation sets the *property-name* property of the object to a list whose car is *value* and whose cdr is the former *property-name* property of the list. This is analogous to doing

---

(push *value* (get *object property-name*))

(si:property-list-mixin :property-list)

RETURNS: the list of alternating property names and values that implements the property list.

(si:property-list-mixin :property-list-location)

RETURNS: a locative pointer to the cell in the instance which holds the property list data.

(si:property-list-mixin :set-property-list list)
Sets the list of alternating property names and values that implements the property list to *list*. So does

---

(send *object* :set :property-list *list*)

(si:property-list-mixin :property-list list)
This initializes the list of alternating property names and values that implements the property list to *list*.

19.14. Copying Instances

There are no built-in techniques to copy instances because there are too many questions raised about what should be copied. These include:

* Do you or do you not send an **:init** message to the new instance? If you do, what init-plist options do you supply?

* If the instance has a property list, you should copy the property list (e.g. with **copy-list**) so that **putprop** or **remprop** on one of the instances does not affect the properties of the other instance.

* If the instance is a pathname, the concept of copying is not even meaningful. Pathnames are *interned*, which means that there can only be one pathname object with any given set of instance-variable values.

* If the instance is a stream connected to a network, some of the instance variables represent an agent in another host elsewhere in the network. Should the copy talk to the same agent, or should a new agent be constructed for it?

* If the instance is a stream connected to a file, should copying the stream make a copy of the file or should it make another stream open to the same file? Should the choice depend on whether the file is open for input or for output?

In general, you can see that in order to copy an instance one must understand a lot about the instance. One must know what the instance variables mean so that the values of the instance variables can be copied if necessary. One must understand what relations to the external environment the instance has so that new relations can be established for the new instance. One must even understand what the general concept 'copy' means in the context of this particular instance, and whether it means anything at all.

Copying is a generic operation, whose implementation for a particular instance depends on detailed knowledge relating to that instance. Modularity dictates that this knowledge be contained in the instance's flavor, not in a "general copying function". Thus the way to copy an instance is to send it a message, as in **(send object :copy)**. It is up to you to implement the operation in a suitable fashion, such as

```
(defflavor foo (a b c) ()
  (:inittable-instance-variables a b))

(defmethod (foo :copy) ()
  (make-instance 'foo :a a :b b))
```

The flavor system chooses not to provide any default method for copying an instance, and does not even suggest a standard name for the copying message, because copying involves so many semantic issues.

If a flavor supports the **:reconstruction-init-plist** operation, a suitable copy can be made by invoking this operation and passing the result to **make-instance** along with the flavor name. This is because the definition of what the **:reconstruction-init-plist** operation should do requires it to address all the problems listed above. Implementing this operation is up to you, and so is making sure that the flavor implements sufficient init keywords to transmit any information that is to be copied.

19.15. Miscellaneous Cautions

Since the Lisp-Machine compatibility package is used by the MIT-based flavor system, the read-macro definitions of the slash (/)character are changed to that of the backslash (\). This is of relatively slight importance unless you use the slash in names. You do, of course use it in Unix file names. Instead of using "/usr/lib/lisp" you will have to use "//usr//lib//lisp".

As mentioned earlier, if you are using a version of FRANZ LISP with packages installed, all quotes used with symbols beginning with colons are unnecessary (although not harmful).

APPENDIX A

Index to FRANZ LISP Functions

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.

package – The value of this symbol is the current package. See chapter 17.

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.

APPENDIX D

Lxref: The Lisp Cross Reference Program

Lxref reads cross reference files written by the Lisp compiler *liszt* and prints a cross reference listing on the standard output. *Liszt* will create a cross reference file during compilation when it is given the **+x** switch. Cross reference files usually end in '.x' and consequently *lxref* will append a '.x' to the file names given if necessary.

The *lxref* command line looks like:

+ + lxref [+ N] xref-file ... [+ a source-file ...] [+ A source-file]

The first option to *lxref* is a decimal integer, N, which sets the *ignorelevel*. If a function is called more than *ignorelevel* times, the cross reference listing will just print the number of calls instead of listing each one of them. The default for *ignorelevel* is 50.

The **+a** option causes *lxref* to put limited cross reference information in the sources named. *lxref* will scan the source and when it comes across a definition of a function (that is a line beginning with *'(def'* it will precede that line with a list of the functions which call this function, written as a comment preceded by *';.. '*. All existing lines beginning with *';.. '* will be removed from the file. If the source file contains a line beginning *';.-'* then this will disable this annotation process from this point on until a *';.+'* is seen (however, lines beginning with *';.. '* will continue to be deleted). After the annotation is done, the original file *'foo.l'* is renamed to *'#.foo.l'"* and the new file with annotation is named *'foo.l'*

The **+A** switch, is like **+a** except it associates up to 2 function names per function. For example if you have something "(def-md s-foo md-foo (x)" then lxref will write in the file the names of the functions that call either s-foo or md-foo.

APPENDIX E

Reconfiguring Lisp

The file */lisp/franz/h/config.h* contains the definitions of the parameters for lisp. In order to modify this file you should have a familiarity with the C programming language.

A few of these parameters can be changed, others have values determined by the machine and operating system. Some of the parameters are on a per-machine basis. The ones for the Tektronix 4404 are surrounded by "#ifdef pegasus ... #endif"

The important parameters that you may wish to alter are:

TTSIZE
: the value is the maximum size that lisp can grow. It is measured in 512 byte 'pages', thus the default value of 6120 means that the maximum size is slightly move than 3 megabytes. There is an 18 byte overhead in static table space for each potential lisp page.

NAMESIZE
: the number of entries in the 'namestack' which is the stack used for holding function arguments and local variables. It is rare that a correctly functioning program will exceed the default size of 3072 entries, so if you get a namestack overflow, make sure that your program isn't in a infinite loop before changing this value and rebuilding lisp.

XstackSize
: the number of entries in the alternate stack. This stack is used for bignum arithmetic and fasl. If you get an 'out of alternate stack' message, you can increase this parameter.

To rebuild lisp, change to the directory */lisp/franz/68k* and type "update updatefile". The result will be a new version of lisp named "nlisp".

under /lisp/franz/68k

4400
FRANZ LISP GRAPHICS

First Printing JUL 1985

Tektronix®
COMMITTED TO EXCELLENCE

CONTENTS

4400 FRANZ LISP GRAPHICS

1. Introduction

The Lisp library directory (**/lisp/lib**) includes files (**gelib.c**, **gelib.r**, **gelib.l**, **gelib.o**) which provide an interface to the 4404 C Graphics and Events Library. (See the *4404 AIS Reference Manual.*) Lisp structures that are equivalent to the C display structures are defined, and a Lisp function interfaces to each function in the C library.

Additional graphics support is provided by the files of the Lisp examples directory (**/lisp/examples**).

demo.l	demo.o
draw.l	draw.o
form.l	form.o
menu.l	menu.o
object.l	object.o
telib.l	telib.o

This support includes functions and related structures for setting graphics modes, drawing primitives, painting text, using the mouse, and menus, definition of cursor and halftone Forms, an example of flavors for graphical objects, and a terminal emulator interface. Reading this code is one of the best ways to see how the graphics interface works.

These files are subject to change and are not supported by Tektronix.

2. The C Graphics and Events Library Interface

The Lisp display structures used in the interface to the C Graphics and Events Library correspond directly to underlying C structures defined in **/lib/include/graphics.h**. In Lisp, the **cstructs** module is used by the **gelib** module to create and manipulate these structures, which are garbage-collected like any other Lisp types. All structures (except the actual screen bitmap) are maintained directly in Lisp. Structures are passed to the external functions of the graphics library, which are loaded automatically when the **gelib** module is included in a Lisp program.

The Lisp functions of this interface have the same names as the C library functions. The same number and types of arguments have been retained whenever possible. The semantics of the Lisp functions may differ in values returned.

An example of a simple bitblt operation follows the description of the functions and structures of the Lisp graphics interface. The main purpose of this example is to demonstrate the creation and access functions generated for Lisp C structures.

2.1 Graphics Interface Functions

(BitBlt bbcom**)**

Perform the bitblt command described in the Bbcom structure argument. The structure contains the source and destination rectangles, clipping regions, halftone mask, and combination rule.

(ClearScreen)
Set the full screen bitmap to zeros. If the screen is set to normal video, this will result in a white screen. The terminal emulator is not affected by this call, i.e., the terminal emulator's idea of where to place its next character is unchanged.

(CursorTrack t_or_nil)
Force the cursor to track the mouse. If the argument is t, then moving the mouse will cause the cursor to track the mouse position, otherwise the mouse will have no affect on the cursor. The previous mode is returned (1 for tracking, 0 for non-tracking).

(CursorVisible t_or_nil)
Make the cursor visible or invisible. The cursor is made visible if the argument is t, otherwise it is blanked. The previous mode is returned (1 for visible, 0 for invisible).

(DisplayVisible t_or_nil)
Make the display visible or blanked. The display is made visible if the argument is t, otherwise it is blanked. The previous mode is returned (positive integer for visible, 0 for invisible). When the display is blanked, the screen goes black, and no screen output of any kind is possible until the display is again made visible. The display should only be blanked with (DisplayVisible nil) in a program that includes a subsequent (DisplayVisible t).

(EClearAlarm)
Clear any pending alarms that the process has requested.

(EGetCount)
Return the number of event values in the event buffer waiting to be processed. A negative result indicates an error.

(EGetNewCount)
Return the number of event values in the event buffer which have occurred since the previous call to this function.

(EGetNext)
Return the next value in the event buffer. Since some events require one value and some require three values, this is either a complete event, an event header, or half of a long time event parameter. The event type (see **EGetType**) will be a fixnum in the range -1 to 5, inclusive. A negative value signals an error. Type values 1, 2, 3, and 4 indicate that the event parameter is embedded in the event value (see **EGetParam**). Otherwise (types 0 and 5) the embedded parameter field is ignored, and the parameter is represented by the next two values in the event buffer.

(EGetParam event)
Return the parameter portion of an event value. The event argument should be a value returned by (EGetNext) of type 1 through 4, inclusive (see **EGetType**).

(EGetTime)
Return the time, in milliseconds, since the system was powered up. A returned value of 0 indicates an error.

(EGetType event)
Return the type portion of an event value. The event argument should be an event value returned by (EGetNext).

(ESetAlarm time)
Request a signal when the specified time, in milliseconds, is reached.

(ESetSignal)
Request the event manager to signal the current process when events occur. The event signal is disabled after being issued.

(EventDisable)
Disable event processing, i.e., turn off the event manager.

(EventEnable)
Enable event processing, i.e., turn on the event manager. Any subsequent user input action will cause event values to be created. The terminal emulator is not affected (if the terminal is enabled when events are enabled, then the keyboard will start producing events in addition to ANSI character strings). If only mouse events are desired, then keyboard events can be turned off (see **SetKBCode**).

(ExitGraphics)
Map the bit-mapped display out of the address space of the calling process. This is all that happens. Cursor, panning, event, keyboard and other modes are not affected. If a return to the display state that existed prior to **InitGraphics** is desired, then it is necessary to save that display state, and then restore it (see **SaveDisplay-State** and **RestoreDisplayState**).

(FormCreate width height)
Allocate memory for a Form and its associated bit-map in the address space of the Lisp process. Returns the new Form. Forms created with this function are handled by normal Lisp garbage collection.

(GetButtons)
Returns a fixnum which indicates the state of the mouse buttons. The button states are reported in the low three bits of the fixnum, where bit 0 is the right button, bit 1 is the middle button, and bit 2 is the left button. If the bit is a 0 then the button is up, if the bit is a 1 then the button is down (depressed). The button values are defined as Lisp constants (see below).

(GetCPosition point)
Get the position where the cursor is currently displayed. If cursor/mouse tracking is enabled, i.e., (CursorTrack t), then this is the same as **GetMPosition**. Returns a Point with x and y values in the range 0 to 1023, inclusive.

(GetCursor form)
Returns a Form which is the current cursor image. The argument must be a 16x16 bit Form.

(GetMBounds point1 point2)
Get the limits on mouse motion. Returns a list of two Points with x and y values in the range -32768 to 32767, inclusive. The mouse is limited within the rectangle defined by the first Point at the upper left and the second Point at the lower right.

(GetMPosition point)
Get the position where the mouse is currently pointing. If cursor/mouse tracking is enabled, i.e., (CursorTrack t), then this is the same as **GetCPosition**. Returns a Point with x and y values in the range 0 to 1023, inclusive.

(GetViewport point)

Get the position which the panning hardware is displaying as the upper left corner of the 640x480 display. Returns a Point with x and y values in the range 0 to 1023, inclusive.

(InitGraphics t_or_nil)

Map the bit-mapped display into the address space of the calling process and put the display in graphics mode. If the argument is nil, all other modes are unchanged. If the argument is t, then in addition, the display is cleared, made visible, and set to normal video (black on white) with both mouse and joydisk panning enabled. Returns a Form which defines the screen bit-map.

(PaintLine bbcom point)

Paint a line on the display. A sequence of bitblt operations is performed while stepping a pixel at a time toward the specified position. If one of the exclusive OR rules is specified, and the source is null (ones), the response will instead be as if the line was drawn by the above stepping method to a hidden bit-map, and then that hidden bit-map was combined with the destination bit-map according to the specified rule.

(PanCursorEnable t_or_nil)

Enable screen panning using the cursor. If the argument is t, then auto-panning with the cursor is enabled, otherwise it is disabled. The previous mode is returned (1 for cursor auto-panning enabled, 0 for cursor auto-panning disabled).

(PanDiskEnable t_or_nil)

Enable screen panning using the joydisk. If the argument is t, then auto-panning with the joydisk is enabled, otherwise it is disabled. The previous mode is returned (1 for joydisk auto-panning enabled, 0 for joydisk auto-panning disabled).

(PointToRC rowcol point)

Convert the screen coordinates of a point to the row and column indices that define the terminal emulator character cell which contains that point. Returns a Rowcol with these values.

(ProtectCursor rect1 rect2)

Tell the operating system that graphics operations will be occurring in one or both of the screen areas defined by the two Rects (either Rect may be nil). The operating system will respond by removing the cursor from the screen if it is in either of the two areas. This instruction and its release (ReleaseCursor) should be used if the user is writing or reading directly from the screen. This cursor protection is already included in the routines of this library which draw on the screen.

(RCToRect rect row col)

Given row and column indices which define a terminal emulator character cell, returns the Rect which describes that cell.

(ReleaseCursor)

Tell the operating system to restore the cursor if it was removed due to a **ProtectCursor** call. This call should be used to match every **ProtectCursor** call.

(RestoreDisplayState dispstate)

The state defined by the Dispstate argument is re-established. This includes the coordinates of the viewport, the mouse bounds, the current cursor, the keyboard code, and the modes for display, terminal emulator, cursor, panning, tracking,

screensaver, and video.

(**SaveDisplayState** dispstate)
Copy significant attributes of the current display state into the Dispstate argument. These attributes will include at least the coordinates of the viewport, the mouse bounds, the current cursor, the keyboard code, and the modes for display, terminal emulator, cursor, panning, tracking, screensaver, and video. Returns the modified Dispstate.

(**ScreenSaverEnable** t_or_nil)
Enable the screen saver timeout, which causes the screen to be blanked after 10 minutes of keyboard or mouse inactivity. If the argument is t, then the timeout is enabled, otherwise it is disabled. The previous mode is returned (1 for screen saver enable, 0 for screensaver disabled).

(**SetCPosition** point)
Display the cursor at the specified psition. If cursor/mouse tracking is enabled, i.e., (CursorTrack t), this is the same as **SetMPosition**. The x and y values of the Point argument must be in the range of 0 to 1023, inclusive.

(**SetCursor** form)
Install a new cursor. The cursor argument is a Form which includes the 16x16 bit representation for the new cursor.

(**SetKBCode** code)
Tell the keyboard to output ANSI character strings, if the code is 1, or event codes in addition to ANSI character strings, if the code is 0. Event processing must be enabled before asking for event codes, and the terminal must be enabled to generate ANSI. Enabling events automatically forces the keyboard into ANSI-plus-event mode. The call (SetKBCode 1) would normally be used after enabling the event mechanism, to force the keyboard back to ANSI-only mode, while leaving the mouse generating events.

(**SetMBounds** point1 point2)
Set the limits on mouse motion to be the rectangle defined by point1 at the upper left and point2 at the lower right. The x and y values of these Points may be in the range -32768 to 32767, inclusive.

(**SetMPosition** point)
Position the mouse at the specified point. If cursor/mouse tracking is enabled, i.e., (CursorTrack t), this is the same as **SetCPosition**. The x and y values of the Point argument must be in the range 0 to 1023, inclusive.

(**SetViewport** point)
Sets the panning hardware to display the upper left corner of the 640x480 display at the specified position. The x and y values defined in the point must be in the physical range of the screen, i.e., x in the range 0 to 383, inclusive, and y in the range 0 to 644, inclusive. Values out of these ranges have no effect.

(**TerminalEnable** t_or_nil)
Enable the terminal emulator. If the argument is t, then the terminal emulator is enabled, otherwise it is disabled. The previous mode is returned (1 for terminal emulator enabled, 0 for terminal emulator disabled). If the terminal emulator is disabled, then the normal terminal emulator functions of transmitting and displaying keyboard input are no longer performed automatically. The terminal should only

by disabled with (TerminalEnable nil) in a program that manages its own character input and echoing, if needed, and that includes a subsequent (TerminalEnable t).

(VideoNormal t_or_nil)

Set the video mode of the display. The mode is set to normal video (black on white) if the argument is t, and to inverse video (white on black) otherwise. The previous mode is returned (1 for normal, 0 for inverse).

2.2 Graphics Interface Structures

Structures implemented for the Lisp Graphics and Events Library Interface are described below. At the right of each structure name are two columns. Each entry in the first column is the name of a field in the given structure. The field type is shown in the second column, to the right of the field name. When these Lisp C structures are declared, creation and access functions are created automatically. See the next section for a few examples, and the Foreign Function Interface chapter of the **Lisp Programmers Reference** for more information.

Point	x	short
	y	short

The Point structure is used to represent the location of one pixel in a bitmap. The screen origin of (0,0) is the upper left corner of the screen, with x and y values increasing right and down. For the 4404, the visible portion of the screen is a 640x480 bit viewport into a 1024x1024 bit frame buffer.

Rowcol	row	int
	col	int

The Rowcol structure is used to represent one character cell. The character cell (0,0) is at the upper left with row and col values increasing down and right. The 4404 has 32 lines of text, with 80 characters per line.

Rect	x	short
	y	short
	w	short
	h	short

The Rect structure is used to represent a rectangular region. The x and y values define the upper left corner of the region in screen pixel coordinates. The width and height in pixels are defined by w and h.

Form	addr	*vectori
	w	short
	h	short
	offsetw	short
	offseth	short
	inc	short

The Form structure is used to represent a rectangular bitmap. The addr is a Lisp immediate vector which contains the bitmap itself (the one exception to this is the screen bitmap Form, in which the addr is a pointer to the non-Lisp memory address used for the screen bitmap). Width and height in pixels are stored in w and h. The offset fields are usually used only in cursor Forms, to designate the "hot-spot" of the cursor. The inc field is the number of bytes (always even) in one row of the bit map.

Bbcom	srcform	*Form
	destform	*Form
	srcpoint	Point
	destrect	Rect
	cliprect	Rect
	halftoneform	*Form
	rule	short

The Bbcom structure is used to represent the arguments for a bitblt operation. The srcform is a pointer to the source Form. The destform is a pointer to the destination Form. The srcpoint, an embedded Point, is the location in the source Form where copying begins. The destrect, an embedded Rect, is the destination Rect of the destination Form. The cliprect, an embedded Rect, is the clipping Rect of the destination Form. The halftoneform is a pointer to the halftone Form to be combined with the source Form. The rule, a short in the range 0 to 15, inclusive, is the combination rule for the bitblt operation.

Dispstate	statebits	long
	viewp	Point
	ulmouseb	Point
	lrmouseb	Point
	curarray	(array 16) short
	keycode	char
	b1_reserved	char
	lineincr	short
	dispwidth	short
	dispheight	short
	viewwidth	short
	viewheight	short
	1_reserved	(array 3) long

The Dispstate structure is used to represent various attributes of the display state to saved and restored. These attributes include the coordinates of the viewport, the mouse bounds, the current cursor, the keyboard code, and the modes for display, terminal emulator, cursor, panning, tracking, screensaver, and video.

2.3 Graphics Interface Special Variables

Below are the special variables defined in the Lisp Graphics and Events Library Interface. At the right of each variable name is its type (or value, if known).

bbZero	0
bbSandD	1
bbSandDn	2
bbS	3
bbSnandD	4
bbD	5
bbSxorD	6
bbSorD	7
bbnSorD	8
bbnSxorD	9
bbDn	10
bbSorDn	11
bbSn	12
bbSnorD	13
bbnSandD	14
bbOnes	15

These are the legal values for the combination rule of a bitblt operation. The names of each is a meaningful abbreviation for the rule it represents. S is for source, and D is for destination. The embedded boolean operators and, or, and xor have the usual meanings. Not is represented by n. An n following an S or D means the negation of just the Source or Destination, while a preceding n negates the full expression that follows. Thus "bbSnorD" is (or (not Source) Destination) and "bbnSxorD" is (not (exclusive-or Source Destination)).

DRAW	0
ERASE	1
INVERT	2

These values are defined as constants, but are not currently used.

M_LEFT	4
M_MIDDLE	2
M_RIGHT	1
M_ANY	7

These values are used to represent the various states of the mouse buttons. For example the value of (equal (M_LEFT (GetButtons))) is t if the left button is down and the other buttons are up. The values for middle and right are similar. The value M_ANY is only returned by **GetButtons** if all three buttons were down. However, the functions described in this document which accept M_ANY as an argument use a different test when this value is passed, i.e., (plusp (GetButtons)).

ScrWidth	1024
ScrHeight	1024
ViewWidth	640
ViewHeight	480

These variables represent the dimension of the full screen bitmap and the visible viewport. Values shown are for the 4404.

DS_DISPON	#x0001	[1=display enabled, 0=disabled]
DS_SCRSAVE	#x0002	[1=screen saver enabled, 0=disabled]
DS_VIDEO	#x0004	[1=video normal, 0=video inverse]
DS_TERMEM	#x0008	[1=terminal emulator enabled, 0=disabled]
DS_CAPSLOCK	#x0010	[1=caps lock on, 0=off]
DS_CURSOR	#x0100	[1=cursor enabled, 0=disabled]
DS_TRACK	#x0200	[1=cursor tracks mouse, 0=no tracking]
DS_PANCUR	#x0400	[1=cursor panning enabled, 0=disabled]
DS_PANDISK	#x0800	[1=joydisk panning enabled, 0=disabled]
DS_KBEVENTS	#x10000	[1=keyboard generates event codes, 0=ANSI]

These variables represent the values for specific bits in the statebits field of Disp-state structure. The hex value identifies the bit, and the values in brackets to the right indicate the semantics for each possible bit value.

****origin** Point

This Point is the upper left corner of every Form, with x and y equal to 0.

****maxpoint** **Point**

This Point, with x and y equal to 1023, is the lower right corner of ****screen**, the actual screen bitmap.

****maxrect** Rect

This Rect has the dimensions of ****screen**, the physical screen bitmap. For the 4404, ****maxrect** has x and y values equal to 0, and w and h values equal to 1024.

****visrect** Rect

This Rect has the dimensions of the visible portion of the screen. For the 4404, ****visrect** has x and y values equal to 0, w equal to 640, and h equal to 480.

****screen** Form

This is the Form which holds the actual screen bitmap. For the 4404, ******screen has an addr value equal to the address of the bitmap, w and h equal to 1024, offsetw and offseth equal to 0, and inc equal to 128. Changes to ****screen** are directly mapped to the actual screen.

2.4 A Simple Example of Cstructs and BitBlt

This example is primarily intended to explain the creation and access functions generated automatically by Lisp C-structure definitions. For comprehensive explanation of bitblt, see the Smalltalk "blue book" (*Smalltalk-80: The Language and its Implementation*, pp. 329-362).

=> (load 'gelib)
nil

This loads the Lisp and external functions of the graphics interface.

=> (setq bb (make-Bbcom destform **screen cliprect **visrect rule bbOnes))
#<vectori 34>

A creation function is generated for each c-structure, and given the name of the structure prefixed with "make-". Here a Bbcom is created with **make-Bbcom**. Structure fields may be initialized by including alternating field names and values in the function call. Fields not specified are initialized to nil or 0. Note that cliprect is embedded in the Bbcom structure, so the values of ****visrect** are copied.

```
=> (setq r (Bbcom->cliprect bb))
#<vectori 8>
```

This sets **r** to the cliprect field of the Bbcom structure. A new Rect is allocated, and given values copied from the Bbcom cliprect.

```
=> (describe-cs r)
cstructs record
name: Rect  occupying 8 bytes
  field: x  offset: 0  C-type: short
    value: 0
  field: y  offset: 2  C-type: short
    value: 0
  field: w  offset: 4  C-type: short
    value: 640
  field: h  offset: 6  C-type: short
    value: 480
access functions: (*Rect->x *Rect->y *Rect->w *Rect->h
  Rect->x Rect->y Rect->w Rect->h)
nil
```

The function **describe-cs** may be used to inspect the values of Lisp C structures. Inspection of **r** shows that the subfield values of ****visrect** were used for the cliprect of **bb**, and then copied from **bb** to **r**.

```
=> (Bbcom->cliprect.w bb)
640
```

The dot notation is used to access subfields of embedded structures.

```
=> (Form->w (Bbcom->destform bb))
1024
```

Each of the Form fields (source, destination, and halftone) of a Bbcom is a lispval pointer to a Form structure or nil, rather than an embedded structure. This is how the width field of the destination Form is accessed. Note that the dot notation is not appropriate, because the structure is not embedded.

```
=> (setf (Bbcom->destrect bb) **visrect)
t
```

The function **setf** is used with a structure access function and a value to set the value of a field. Here the destination Rect is given the value of ****visrect**.

```
=> (describe-cs (Bbcom->desrect bb))
cstructs record
name: Rect  occupying 8 bytes
  field: x  offset: 0  C-type: short
    value: 0
  field: y  offset: 2  C-type: short
    value: 0
  field: w  offset: 4  C-type: short
    value: 640
  field: h  offset: 6  C-type: short
    value: 480
access functions: (*Rect->x *Rect->y *Rect->w *Rect->h
  Rect->x Rect->y Rect->w Rect->h)
nil
```

The access function **Bbcom->destrect** is used here with **describe-cs** to inspect the values of the destination Rect of **bb** directly.

```
=> (BitBlt bb)
t
```

Here the bitblt operation is performed. If you have been typing in this example as you read, you will have noticed that your screen just reversed color. All bits in the visible portion of the screen bitmap were set to 1 (black if in normal video mode).

The bitblt operation described above is one of the simplest possible bitblt operations. Since the combination rule is **bbOnes**, the values for source Form, source Point, and halftone Form were not relevant and never set. Any different values for these fields would have given the same result. Values for the other fields, however, are always mandatory. The destination Form, destination Rect, clipping Rect and combination rule must all have meaningful values if a bitblt operation is to be performed correctly.

3. Graphics Support

The functions, macros, structures and variables described in the remaining sections of this document are provided as examples of the Lisp graphics interface, and for interim use until a more complete Lisp display library becomes available. All of the following may be subject to change.

Functions with many options use the **&key** option for arguments (see the "Program Forms" chapter of the Lisp Manual). In the descriptions that follow, this is indicated with "KEY-ARG-PAIRS" in the argument list. A list of the keywords supported for the particular function immediately follows the argument list.

The formats for description of structures and variables are the same as are used in the previous Lisp Graphics Interface sections.

3.1 Graphics Mode

The functions, macros, and variables below (defined in the file **/lisp/examples/draw.l**) are used for setting characteristics of the Lisp graphics environment.

****graphrect** Rect

This global variable is the current graphics region, used as the default clipping Rect. Its value is set by **init-graphics-mode** and by **set-scrolling-region**.

(init-graphics-mode KEY-ARG-PAIRS)

:cursor-loc	initial graphics-cursor Point
:cursor-p	graphics-cursor visible or invisible
:lines	number of lines in text scrolling region
:mouse-bounds	list of two Points defining mouse bounds
:pan-cursor-p	pan cursor scrolling enable or disable
:pan-disk-p	pan disk scrolling enable or disable
:video-normal-p	normal or inverse video
:viewport-origin	upper left display Point

Turn on Lisp graphics mode and set the current default clipping Rect or graphics region (****graphrect**). Defaults for keyword arguments are normal viewport (Set-Viewport **origin) and video mode (VideoNormal t), cursor and disk panning disabled, and a visible graphics-cursor in the upper right of the display. The default graphics region is the entire screen bitmap. Macros are provided for common configurations (see below), **full-screen-graphics**, **viewport-graphics**, and **split-screen-graphics**. When a text scrolling region is defined, ****graphrect** is set to the remainder of the visible screen. At the first call to **init-graphics-mode**, the previous display state is saved. When the next call to **exit-graphics-mode** is made, the saved display state is restored. A change from one graphics mode to another, i.e., two calls to **init-graphics-mode** without an intervening call to **exit-graphics-mode**, leaves the previously saved display state unchanged.

(exit-graphics-mode)

Restore full-screen text-scrolling and the display state saved at the time of the first call to **init-graphics-mode**. The screen is cleared and the text cursor is located in the upper left-hand corner of the display. This is a no-op if not already in graphics mode.

(set-scrolling-region KEY-ARG-PAIRS)

:lines	lines in text scrolling region

Set the size of the text scrolling region and the value of ****graphrect**. The default for :lines is 32, which restores the normal scrolling region for the terminal emulator. Otherwise a text region of the size indicated is created at the bottom of the screen. The text region is cleared, and the cursor moved to the upper left-hand corner of the text region. The variable ****graphrect** is set to the portion of the visible screen which is not used for text. The graphics region is not cleared.

(full-screen-graphics)

Set the graphics region to ****maxrect**, the full screen bitmap, with cursor and disk panning enabled. Normal terminal emulator interactions will continue to scroll the screen, so this mode would generally be used when all input and output is handled graphically.

(viewport-graphics)

Set the graphics region to ****visrect**, the visible portion of the screen, with cursor and disk panning disabled. Normal terminal emulator interactions will continue to scroll the screen, so this mode would generally be used when all input and output is handled graphically.

(split-screen-graphics lines)

Create a text scrolling region at the bottom of the screen, and set the graphics region to the remainder of the screen. The argument determines the number of text lines in the scrolling region. A line drawn across the screen divides the regions. Cursor and disk panning are disabled, and the mouse is limited to the graphics region.

(clear-display)

Erase the graphics region by setting all bits to 0.

(clear-text-region)

Erase the scrolling text region, and home the cursor to the upper left-hand corner of the region.

3.2 Drawing

The functions below (defined in the file **/lisp/examples/draw.l**) are used for drawing lines, circles, rectangles, and boxes.

(draw-line point1 point2 KEY-ARG-PAIRS)

:bb	Bbcom with destform, destrect, cliprect and rule set
:cliprect	clipping Rect
:destform	destination Form
:rule	bitblt combination rule
:width	number of pixels in line width

Draw a line from point1 to point2. If :bb is specified, then the :cliprect, :destform, and :width values are ignored. Otherwise, defaults include ****screen** for :destform, ****graphrect** for :cliprect and **bbSorD** for :rule.

(draw-lines points KEY-ARG-PAIRS)

:width	number of pixels in line width
:closed-form-p	first and last Point connected or not connected.

Draw a line connecting each Point in the points argument to its successor. The defaults for keyword arguments are 1 for :width and nil for :closed-form-p.

(draw-circle center-point radius)

Draw a circle centered at center-point with the specified radius. The circle is drawn on an intermediate Form, then bitblted on to ****screen** with **bbSorD**.

(draw-rectangle rect KEY-ARG-PAIRS)

:bb	Bbcom with destform, destrect, cliprect and rule set
:cliprect	clipping Rect
:destform	destination Form
:halftone	halftone Form
:rule	bitblt combination rule

Draw a solid rectangle at the region specified by rect. Defaults include ****screen** for :destform, ****graphrect** for :cliprect, nil for :halftone, and **bbS** for :rule.

(draw-box rect KEY-ARG-PAIRS)

:cliprect	clipping Rect
:destform	destination Form
:width	number of pixels in border width

Draw a box with the interior region specified by rect. The border of the box is drawn around this region. Defaults include ****screen** for :destform, ****graphrect** for :cliprect, and 1 for :width.

3.3 Mouse

The functions below (defined in the file **/lisp/examples/draw.l**, except **rect-from-user**, defined in **/lisp/examples/form.l**) use the mouse. See also **menu-choose**, in the section on menus.

(**mouse-in-region** rect)
Return t if the mouse cursor is positioned within the specified Rect, otherwise return nil.

(**wait-mouse-click** button)
Wait for the press and release of the specified mouse button (**M_LEFT**, **M_MIDDLE**, **M_RIGHT**, or **M_ANY**). The button pressed and the location of the mouse are returned as multiple values.

(**rect-from-user** &optional minwidth minheight)
Wait for the user to select a rectangular region of the display with the left mouse button. Pressing the button specifies the upper left-hand corner of the region, and releasing the button specifies the lower right-hand corner. The specified Rect and a Bbcom are returned as multiple values. The returned Bbcom defines a bitblt operation which will restore the screen pixel values in the region to what they were before selection. The optional arguments are used to specify a minimum height and width.

3.4 Text

The functions and variables below (defined in the file **/lisp/examples/menu.l**) are used for painting text onto the screen and other bitmap Forms. These are independent of the terminal emulator, which is discussed in a different section.

****font** Font

This variable is used to reference the current Font. A font contains the same character images used by the terminal emulator has been defined. When the file **/lisp/examples/menu.l** is loaded, ****font** is initialized to this Font.

(**paint-string** x y string &optional font rule destform)
Paint the specified string onto a Form. The upper left corner of the first character image is located at (x, y). Defaults include ****font** for font, **bbS** for rule, and ****screen** for destform. The default font includes images for all ASCII character codes, but in Lisp only certain characters are legal for strings. Use **paint-char** instead of **paint-string** when it is necessary to circumvent this restriction.

(**paint-char** x y char &optional font rule destform)
Paint the character with ASCII value char onto a Form. The upper left corner of the character image is located at (x, y). Defaults include ****font** for font, **bbS** for rule, and ****screen** for destform.

(**string-pixel-size** item font)

Return the width in pixels required to paint item (a symbol or string) with the specified Font.

3.5 Menus

The functions below (defined in the file **/lisp/lib/examples/menu.l**) are for the creation and use of pop-up menus.

(**make-menu** item-list &optional selector cliprect-var)

Create a menu from item-list. A menu is a symbol with associated structures and values maintained on a property list. The item-list is a list of items, each of which is associated with one slot of the menu. Each item is either a string, a symbol, or a list with a string or symbol at the head of the list. The value displayed in the menu is the string or the print name of the symbol. If the item is a string or symbol, then the item itself is returned if selected. If the item is a list, then the second item of the list is returned. The optional selector indicates the mouse button used to select from the menu, which defaults to **M_ANY**. The cliprect-var is a quoted symbol to be evaluated when the menu is used, to return the Rect in which the menu should be displayed. The default cliprect-var is '****maxrect**.

(**make-icon-menu** item-list &optional selector cliprect-var)

Create a menu from item-list. This is identical to **make-menu** (see above), except that Forms take the place of strings. A menu is a symbol with associated structures and values maintained on a property list. The item-list is a list of items, each of which is associated with one slot of the menu. Each item is either a Form, or a list with a Form at the head of the list. The value displayed in the menu is the Form. If the item is a Form, then the item itself is returned if selected. If the item is a list, then the second item of the list is returned. The optional selector indicates the mouse button used to select from the menu, which defaults to **M_ANY**. The cliprect-var is a quoted symbol to be evaluated when the menu is used, to return the Rect in which the menu should be displayed. The default cliprect-var is '****maxrect**.

(**menu-choose** menu)

Pop up a menu and wait for a selection. A menu is an object returned by **make-menu** or **make-icon-menu**. The value returned from selecting a menu item is described above. The menu is centered at the location of the graphics-cursor. Nil is returned if no item is selected, i.e., if the selector is pressed and released outside of the menu.

3.6 Cursors and Halftones

The variables below (defined in the file **/lisp/examples/form.l**) are Forms and lists of Forms which represent cursors and halftones.

****cursors** list of Forms

The variable ****cursors** is a list of the cursors described below. A cursor is a 16x16 bit Form.

CornerCursor	Form
CrosshairCursor	Form
DownCursor	Form
NormalCursor	Form
OriginCursor	Form
ReadCursor	Form
SquareCursor	Form
UpCursor	Form
WaitCursor	Form
WriteCursor	Form
XeqCursor	Form

These cursor Forms are equivalent to the standard Smalltalk cursors.

****halftones** list of Forms

This variable is a list of the halftones described below. A halftone is a 16x16 bit Form.

BlackHalftone	Form
DarkGrayHalftone	Form
GrayHalftone	Form
LightGrayHalftone	Form
VeryLightGrayHalftone	Form

These halftone Forms are equivalent to the standard Smalltalk halftones.

4. Terminal Emulator Interface

An interface to the terminal emulator (defined in **/lisp/examples/telib.l**) is provided as a Lisp package. The Lisp functions of this interface are used to generate the escape-prefixed and other special character strings necessary to issue terminal emulator commands. See Section 10 of the *4404 AIS Reference Manual* for more information. Lisp functions which use the terminal emulator interface include **set-scrolling-region**, and **clear-text-region**.

5. Smalltalk Forms for Lisp

The Smalltalk bit editor may be used to create a Form that can be read and used for graphics in Lisp. To create the Form **newCursor.f**, use the following steps:

1. Invoke smalltalk.
2. Open up a workspace.
3. Enter the following Smalltalk statements (where <- is replaced by the true Smalltalk backarrow):

```
aForm <- Form new extent: 16@16.
aForm <- aForm bitEdit.
```

4. Select those statements and DoIt.
5. Use the biteditor to create the cursor Form desired.
6. Use the middle button to 'accept' the Form.
7. Enter the following Smalltalk statement:

```
aForm writeOn: 'newCursor.f'.
```

8. Select the statement and DoIt.
9. Exit Smalltalk.

The function **ReadForm** (defined in **/lisp/lib/gelib.l**) will make the Smalltalk Form available to Lisp. To replace the current cursor with the new Form in **newCursor.f**,

1. Invoke lisp.
2. (include gelib)
3. (InitGraphics t)
4. (CursorVisible t)
5. (setq oldcursor (GetCursor (FormCreate 16 16)))
6. (SetCursor (ReadForm "newCursor.f"))
7. To restore the normal cursor, (SetCursor oldcursor)

6. Graphics Examples

Demonstration functions (defined in **/lisp/examples/demo.l**) which use the Lisp graphics interface and other Lisp graphics support functions are included. To run these examples,

1. Invoke lisp.

2. (load '/lisp/examples/demo)
   Other necessary files are loaded automatically.
   This takes awhile.

3. (demos)
   The screen is divided into two regions.
   The upper region is used for graphics, and
   the lower region is used for instructions
   and explanation. A menu will pop up, and
   instructions will appear in the text region.
   The examples themselves are self-explanatory.

INDEX

FRANZ LISP VERSION 42 INSTALLATION NOTES

These installation notes provide important information about your new release of Franz Lisp. The major sections are: Installation, Release Notes, and Franz Lisp Files. Read this document carefully **before** installing Franz Lisp on your system.

1. INSTALLATION

This version of Franz Lisp is compatible with Tektronix 4404 Operating System versions 1.5 and later. It is **incompatible** with earlier versions of the operating system and C libraries. Be certain that you are running a compatible version of the operating system before installing this release of Franz Lisp.

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

1. At the system prompt "+#", log in as the system manager by typing:

   login system

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

   restore -l

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

4. 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. See the section *FRANZ LISP FILES* below for a description of the files included in the release.

2. RELEASE NOTES

2.1 Introduction

This is a description of the changes between Franz Lisp Versions 41 and 42. There are many new features and some incompatibilities between these versions, but the conversion of Lisp programs to Version 42 Franz Lisp should be simple.

2.2 Incompatibilities

There are two major changes that result in incompatibilities between Versions 42 and 41: reader syntax for packages and fasl format. It is very important to read the following sections about packages and fasl format.

2.3 New Features and Enhancements

2.4 Packages

This new feature provides multiple name spaces for Lisp symbols, and is completely compatible with Common Lisp packages.

Important: the syntax of the character ":" has changed, and programs which use ":" in atom names will have to be changed. Please see chapter 17 for details about packages.

2.5 Fasl format

The format of fasl files (output of the Lisp compiler, *liszt*) has changed in an incompatible way. Compiled Franz Lisp programs created from earlier versions of the compiler will not load into Version 42. **All Franz Lisp programs should be recompiled.**

2.6 Keyword Arguments

The Lisp function *defun* has been enhanced to allow keyword arguments, as described in chapter 8.

2.7 Defstruct

This is the structure package from Lisp Machine Lisp and Common Lisp. It is described in chapter 14.

2.8 Flavors

A new implementation of flavors, object-oriented programming for Lisp, has been added to Franz Lisp. See chapter 19 for a description of flavors.

2.9 Foreign Function Interface

The Franz Lisp to foreign function interface has been greatly enhanced. See chapter 18 for more details.

2.10 Lisp Compiler Speedups

For MC68000 versions of Franz Lisp, the compiler has been improved in the area of fixnum arithmetic. All versions of the compiler now allow vector bounds checking to be turned off via the declarations: *(declare (vector-bounds-chk nil))*.

2.11 Closures

Lisp Machine closures have been fully implemented. See chapter 2 for more details.

2.12 Reader Syntax

The reader syntax has been modified for Common Lisp compatibility. See chapter 8 for more details.

2.13 Documentation Additions and Errata

char-rindex (p. 2-15)

> *char-rindex* returns the position of the first occurrence of stx_char from the end of t_string.

Exponential notation (p. 3-1)

> Exponential format for flonums is accepted by the Lisp reader, even though the manual does not mention it. For example, 1e10 returns 10000000000.0.

cerror (p. 4-10)

The manual is in error when it states that the function cerror returns nil. The error is continuable, via the tpl command *?ret*, and may return particular values. For example, to return a value of 10 from a call to cerror *?ret 10* would be typed at the *c{n}* prompt.

readdir (p. 5-8)

The function *readdir* is in the system package, not in the Lisp package, so must be called as *sys:readdir* from the user package.

signal (p. 6-6)

The function *signal* catches all signals, and not just the four that the manual indicates.

Reader (p. 7-1)

There is a new syntax class in the reader, *vpackage*. The character *:* is in this class.

Closures (p. 8-11)

The function *closurep* should be added to section 8.3.2 of the manual. It returns **t** if the sole argument is a closure.

Also, if a symbol given to *symeval-in-closure* or *set-in-closure* is not one of the symbols closed over in the closure, then the effect of the function call is identical to *symeval* and *set*, respectively.

cfasl (p. 18-3)

There are two definitions of the function *cfasl*. the correct one is found on page 5-2.

2.14 Function List

The following is a list of all functions which are either new or changed in the new version of Franz Lisp. Functions with no comment next to them are new, otherwise the comment should explain its status.

**invmod*
allsym
apropos
apropos-list
ash
c-declare
case
cerror
char-index
char-rindex
charcnt
cli:error
closure
closurep
defun — changed to include &key option
defsetf
defsubst

describe-cs	
do-all-symbols	
do-external-symbols	
do-symbols	
double-to-float	
escape-exploden	
fboundp	
feature-present	
ffasl	
fileopen	
find-all-symbols	
find-package	
find-symbol	
float-to-double	
format	
frexp	removed
fseek	
gcbefore	
getenv	
hashtabstat	removed
if	additional new form consistent with Common Lisp
implodes	
import	
in-package	
initsym	
integer-length	
intern	functionality expanded to include packages
keywordp	
let-closed	
*list**	
list-all-packages	
litatom	changed
logand	
logandc1	
logandc2	
logbitp	
logcount	
logeqv	
logior	
logiorc1	
logiorc2	
lognand	
lognor	
lognot	
logtest	
logxor	
make-package	
make-vector-float	replaced by new-vectori-float
map-over-oblist	removed
oblist	changed for packages
nequal	

new-vectori-double
new-vectori-float
newsym
oldsym
package-name
package-nicknames
package-shadowing-symbols
package-use-list
package-used-by-list
packagep
pop
primep
provide
push
pushnew
qualify-escape-exploden
qualify-explode
qualify-explodec
qualify-exploden
rassq
readdir
readline
remsym
rename-package
set-in-closure
shadow
shadowing-import
sprintf
sstatus ignoreeof removed
str=
strcat
string
substrp
symbol-function
symbol-name
symbol-package
symbol-plist
symbol-value
symeval-in-closure
symstat
sys:getpwnam
sys:getpwuid
sys:getuid
tilde-expand
truename
unexport
unintern
unless
unuse-package
use-package
vectori-double

vectori-float	
vref-float	replaced by vrefi-float
vrefi-double	
vrefi-float	
vset-float	replaced by vseti-float
vseti-double	
vseti-float	
vsize-double	
vsize-float	
when	
wide-print-list	
with-keywords	replaced by defun &key option
y-or-n-p	

3. FRANZ LISP FILES

The following files are distributed with Franz Lisp version 42. Files ending in the extension *.l* are human-readable Lisp source code. Files ending in the extension *.o* are compiled Lisp object code.

3.1 LISP EXECUTABLE FILES

The followings files are found in */bin*:

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

3.2 LISP HELP FILES

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

lisp
liszt
lxref

3.3 LISP SOURCE FILES

The following files, which are loaded into the standard Lisp interpreter, are found in */lisp/franz/lispsrc*:

array.l	Array functions.
autoload.l	Manages autoload of functions.
buildlisp.l	Used to build the Lisp system from the C kernel.
charmac.l	Backquote and sharp sign macros.
cl.l	Common Lisp functions.

cli.l	Common Lisp incompatible functions.
closure.l	Fclosure functions.
common0.l *common1.l* *common2.l* *common3.l*	Most Lisp-coded Lisp functions are in *common files*.
compiler.l	Compiler symbols.
filep.l	File functions accessed through the *tpl* top level.
format.l	String formatting, compatible with Zetalisp.
load.l	File functions.
macros.l	Common macros for Franz Lisp.
package.l	Package functions.
syntax.l	Contains the setsyntax function.
sysint.l	Contains the *system-internal* or *si* package.
system.l	Contains system-interface functions in the *system* or *sys* package.
tpl.l	Franz Lisp top level.
vector.l	Vector handling functions.
version.l	Franz Lisp version info.

3.3.1 System Rebuild

The following files are used when rebuilding/reconfiguring Lisp (see Appendix E of the Franz Lisp Reference Manual):

In */lisp/franz*:

userdata.c

In */lisp/franz/68k*:

loadlisp.l

ulisp.r

updatefile

3.3.2 Other Lisp Source Files

The following files are found in */lisp/franz/h*:

68kframe.h	aout.h	aout_peg.h	catchfram.h	chars.h
config.h	debug.h	dfuncs.h	flavors.h	frame.h
gc.h	global.h	hooks.h	hpagsiz.h	ioext.h
lconf.h	lfuncs.h	lispo.h	lispo_mitas.h	lstructs.h
ltypes.h	module.h	package.h	public.h	savelisp.h
sigtab.h	space.h	structs.h	types.h	vaxframe.h

3.4 LISP LIBRARY FILES

The following files are found in */lisp/lib*:

arun.c, arun	Used to generate autorun files.
as	Assembler for *liszt* files.
cmuedit.l, cmuedit.o	Code for an interactive structure editor. Loaded when edit functions are called.
cstructs.l, cstructs.o	Functions providing C structure compatibility. Loaded when c-declare function is called.
describe.l, describe.o	Functions to describe any Lisp object, including *flavors*.
fix.l, fix.o	Fix package that is autoloaded when the function debug is invoked.
flavors.l, flavors.o	Flavor system, object definition and creation.
gelib.c, gelib.r *gelib.l, gelib.o*	Graphics library interface functions.
lmhacks.l, lmhacks.o	Miscellaneous functions compatible with Zetalisp.
machacks.l, machacks.o	Maclisp compatibility package. Autoloaded when the *+m* option is specified for *liszt*.
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.
struct.l, struct.o	Structure package.
structini.l	Macros necessary for compiling the structure package.
trace.l, trace.o	Trace package. Loaded when trace function invoked.
vanilla.l, vanilla.o	Definition of vanilla flavors and methods.

3.5 ADDITIONAL LISP FILES

Graphics support and demo programs are distributed in */lisp/examples*. They are subject to change.

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

In */lisp/lib*:

cmuenv.l	Loads *cmumacs*, *cmufncs*, *cmutop*, and *cmufile* for a cmu environment.
cmufile.l, cmufile.o	Functions for Cmu file package.
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.
loop.l, loop.o	Loop macro.
ucido.l, ucido.o	UCI Lisp do loop.
ucifnc.l, ucifnc.o	UCI Lisp compatibility package.