The FRANZ LISP Manual

by

John K. Foderaro

Keith L. Sklower

Kevin Layer

June 1983

A document in
four movements

Overture

*A chorus of students under the direction of Richard Fateman have contributed to building FRANZ LISP from a mere melody into a full symphony . The major contributors to the initial system were Mike Curry, John Breedlove and Jeff Levinsky. Bill Rowan added the garbage collector and array package. Tom London worked on an early compiler and helped in overall system design. Keith Sklower has contributed much to FRANZ LISP, adding the bignum package and rewriting most of the code to increase its efficiency and clarity. Kipp Hickman and Charles Koester added hunks. Mitch Marcus added *rset, evalhook and evalframe. Don Cohen and others at Carnegie-Mellon made some improvements to evalframe and provided various features modelled after UCI/CMU PDP-10 Lisp and Interlisp environments (editor, debugger, top-level). John Foderaro wrote the compiler, added a few functions, and wrote much of this manual. Of course, other authors have contributed specific chapters as indicated. Kevin Layer modified the compiler to produce code for the Motorola 68000, and helped make FRANZ LISP pass ''Lint''.*

This manual may be supplemented or supplanted by local chapters representing alterations, additions and deletions. We at U.C. Berkeley are pleased to learn of generally useful system features, bug fixes, or useful program packages, and we will attempt to redistribute such contributions.

Work reported herein was supported in part by the U. S. Department of Energy, Contract DE-AT03-76SF00034, Project Agreement DE-AS03-79ER10358, and the National Science Foundation under Grant No. MCS 7807291

Score

First Movement *(allegro non troppo)*

Second Movement *(Largo)*

Third Movement *(Scherzo)*

Final Movement *(allegro)*

CHAPTER 1

FRANZ LISP

1.1. FRANZ LISP[†] was created as a tool to further research in symbolic and algebraic manipulation, artificial intelligence, and programming languages at the University of California at Berkeley. Its roots are in a PDP-11 Lisp system which originally came from Harvard. As it grew it adopted features of Maclisp and Lisp Machine Lisp. Substantial compatibility with other Lisp dialects (Interlisp, UCILisp, CMULisp) is achieved by means of support packages and compiler switches. The heart of FRANZ LISP is written almost entirely in the programming language C. Of course, it has been greatly extended by additions written in Lisp. A small part is written in the assembly language for the current host machines, VAXen and a couple of flavors of 68000. Because FRANZ LISP is written in C, it is relatively portable and easy to comprehend.

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

This document is a reference manual for the FRANZ LISP system. It is not a Lisp primer or introduction to the language. Some parts will be of interest primarily to those maintaining FRANZ LISP at their computer site. There is an additional document entitled *The Franz Lisp System,* by John Foderaro, which partially describes the system implementation. FRANZ LISP, as delivered by Berkeley, includes all source code and machine readable version of this manual and system document. The system document is in a single file named "franz.n" in the "doc" subdirectory.

This document is divided into four Movements. In the first one we will attempt to describe the language of FRANZ LISP precisely and completely as it now stands (Opus 38.69, June 1983). In the second Movement we will look at the reader, function types, arrays and exception handling. In the third Movement we will look at several large support packages written to help the FRANZ LISP user, namely the trace package, compiler, fixit and stepping package. Finally the fourth movement contains an index into the other movements. In the rest of this chapter we shall examine the data types of FRANZ LISP. The conventions used in the description of the FRANZ LISP functions will be given in §1.3 -- it is very important that these conventions are understood.

1.2. Data Types FRANZ LISP has fourteen data types. In this section we shall look in detail at each type and if a type is divisible we shall look inside it. There is a Lisp function *type* which will return the type name of a lisp object. This is the official FRANZ LISP name for that type and we will use this name and this name only in the manual to avoid confusing the reader. The types are listed in terms of importance rather than alphabetically.

[†] It is rumored that this name has something to do with Franz Liszt [Frants List] (1811-1886) a Hungarian composer and keyboard virtuoso. These allegations have never been proven.

1.2.0. lispval This is the name we use to describe any Lisp object. The function *type* will never return ‘lispval’.

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

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

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

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

A symbol also has a hashlink used to link symbols together in the oblist -- this field is inaccessible to the lisp user.

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

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

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

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

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

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

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

1.2.5. flonum A flonum is a double precision real number. On the VAX, the range is $\pm 2.9\times 10^{-37}$ to $\pm 1.7\times 10^{38}$. There are approximately sixteen decimal digits of precision. Other machines may have other ranges.

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. Should calculation with bignums give a result which can be represented as a fixnum, then the fixnum representation will be 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.** The user 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 which operate on the symbol's print name will also work on strings. The default reader syntax is set so that a sequence of characters surrounded by double quotes is a string.

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

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

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

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

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

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

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

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

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

1.3. Documentation The conventions used in the following chapters were 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 any. The arguments all have names which begin with a letter or letters and an underscore. The letter(s) gives the allowable type(s) 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
a	array
e	value
p	port (or nil)
h	hunk

In the first line of a function description, those arguments preceded by a quote mark are evaluated

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

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

CHAPTER 2

Data Structure Access

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

2.1. Lists

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

2.1.1. list creation

(cons 'g_arg1 'g_arg2**)**

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

(xcons 'g_arg1 'g_arg2**)**

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

(ncons 'g_arg**)**

EQUIVALENT TO: *(cons 'g_arg nil)*

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

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

(append 'l_arg1 'l_arg2**)**

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

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

(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 for 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 ! and !!. ! g_eform evaluates g_form and the value is inserted in the place of the call; !! l_form evaluates l_form and the value is spliced into the place of the call.

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

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

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

(bignum-to-list ’b_arg**)**

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

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

(list-to-bignum ’l_ints**)**

WHERE: l_ints is a list of fixnums.

RETURNS: a bignum constructed of the given fixnums.

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

2.1.2. list predicates

(dtpr 'g_arg**)**

RETURNS: t iff g_arg is a list cell.

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

(listp 'g_arg**)**

RETURNS: t iff 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: *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 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 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 will be used. If g_nodups is non-nil, an element will not be inserted if an equal element is already in the list. *insert* does 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 will be used. u_comparefn should be thought of as "less than or equal". *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* will be 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 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 a 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** will be returned for 'false' and something other than **nil** (often **t**) will be returned for 'true'.

(arrayp 'g_arg**)**

RETURNS: t iff g_arg is of type array.

(atom 'g_arg**)**

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

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

(bcdp 'g_arg**)**

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

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

(bigp 'g_arg**)**

RETURNS: t iff g_arg is a bignum.

(dtpr 'g_arg**)**

RETURNS: t iff g_arg is a list cell.

NOTE: that (dtpr '()) is nil.

(hunkp 'g_arg**)**

RETURNS: t iff g_arg is a hunk.

(listp 'g_arg**)**

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

(stringp 'g_arg**)**

RETURNS: t iff g_arg is a string.

(symbolp 'g_arg**)**

RETURNS: t iff g_arg is a symbol.

(valuep 'g_arg**)**

RETURNS: t iff g_arg is a value cell

(vectorp 'v_vector**)**

RETURNS: **t** iff the argument is a vector.

(vectorip 'v_vector**)**

RETURNS: **t** iff the argument is an immediate-vector.

(type 'g_arg**)**
(typep 'g_arg**)**

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

(signp s_test 'g_val**)**

RETURNS: t iff 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 ≤ 0
e	g_val $= 0$
n	g_val $\neq 0$
ge	g_val ≥ 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 the exact same place in memory. Lisp objects which 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 will be interned.]

(neq ’g_x ’g_y**)**

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

(equal ’g_arg1 ’g_arg2**)**
(eqstr ’g_arg1 ’g_arg2**)**

RETURNS: t iff 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 will always work.
–> (eq 1023 1023)
t
–> (eq 1024 1024)
nil
–> (equal 1024 1024)
t
```

(not ’g_arg**)**
(null ’g_arg**)**

RETURNS: t iff 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. To remind ourselves of the difference, a string is a null terminated sequence of characters, stored as compactly as possible. Strings are used as constants in FRANZ LISP. They *eval* to themselves. A symbol has additional structure: a value, property list, function binding, as well as its external representation (or print-name). If a symbol is given to one of the string manipulation functions below, its print name will be used as the string.

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

2.3.1. symbol and string creation

(concat [’stn_arg1 ...]**)**
(uconcat [’stn_arg1 ...]**)**

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

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

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

(concatl ’l_arg**)**

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

(implode ’l_arg**)**
(maknam ’l_arg**)**

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

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

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

(gensym ['s_leader]**)**

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

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

(copysymbol 's_arg 'g_pred**)**

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

(ascii 'x_charnum**)**

WHERE: x_charnum is between 0 and 255.

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

(intern 's_arg**)**

RETURNS: s_arg

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

(remob 's_symbol**)**

RETURNS: s_symbol

SIDE EFFECT: s_symbol is removed from the oblist.

(rematom 's_arg**)**

RETURNS: t if s_arg is indeed an atom.

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

NOTE: This function does *not* check to see if s_arg is on the oblist or is referenced anywhere. Thus calling *rematom* on an atom in the oblist may result in disaster when that atom cell is reused!

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 iff the 'name' of st_arg1 is alphabetically less than the name of st_arg2. If st_arg is a symbol then its 'name' is its print name. If st_arg is a string, then its 'name' is the string itself.

2.3.3. symbol and string accessing

(symeval ’s_arg**)**

RETURNS: the value of symbol s_arg.

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

(get_pname ’s_arg**)**

RETURNS: the string which is the print name of s_arg.

(plist ’s_arg**)**

RETURNS: the property list of s_arg.

(getd ’s_arg**)**

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

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

(getchar ’s_arg ’x_index**)**
(nthchar ’s_arg ’x_index**)**
(getcharn ’s_arg ’x_index**)**

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

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

(substring ’st_string ’x_index [’x_length]**)**
(substringn ’st_string ’x_index [’x_length]**)**

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

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

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

2.3.4. symbol and string manipulation

(set ’s_arg1 ’g_arg2**)**

RETURNS: g_arg2.

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

(setq s_atm1 ’g_val1 [s_atm2 ’g_val2 ... ...]**)**

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

RETURNS: the last g_val*i*.

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

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

(desetq sl_pattern1 ’g_exp1 [... ...]**)**

RETURNS: g_expn

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

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

(setplist ’s_atm ’l_plist**)**

RETURNS: l_plist.

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

(makunbound ’s_arg**)**

RETURNS: s_arg

SIDE EFFECT: the value of s_arg is made ‘unbound’. If the interpreter attempts to evaluate s_arg before it is again given a value, an unbound variable error will occur.

(aexplode ’s_arg**)**
(explode ’g_arg**)**
(aexplodec ’s_arg**)**
(explodec ’g_arg**)**
(aexploden ’s_arg**)**
(exploden ’g_arg**)**

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

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

```
–> (setq x ’|quote this \| ok?|)
|quote this \| ok?|
–> (explode x)
(q u o t e |\\| | | t h i s |\\| | | |\\| |\|| |\\| | | o k ?)
; note that |\\| just means the single character: backslash.
; and |\|| just means the single character: vertical bar
; and | | means the single character: space

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

2.4. Vectors

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

2.4.1. vector creation

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

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

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

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

Vectors may be created by specifying multiple initial values:

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

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

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

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

2.4.2. vector reference

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

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

(vprop 'Vv_vect**)**

RETURNS: The Lisp property associated with a vector.

(vget 'Vv_vect 'g_ind**)**

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

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

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

2.4.3. vector modfication

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

RETURNS: the datum.

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

(vsetprop 'Vv_vect 'g_value**)**

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

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

(vputprop 'Vv_vect 'g_value 'g_ind**)**

RETURNS: g_value.

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

2.5. Arrays

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

2.5.1. array creation

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

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

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

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

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

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

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

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

2.5.2. array predicate

(arrayp ’g_arg**)**

RETURNS: t iff g_arg is of type array.

2.5.3. array accessors

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

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

(arrayref ’a_name ’x_ind**)**

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

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

(arraycall s_type ’as_array ’x_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.

```
; We will create a 3 by 4 array of general lisp objects
–> (array ernie t 3 4)
array[12]

; the array header is stored in the function definition slot of the
; symbol ernie
–> (arrayp (getd ’ernie))
t
–> (arraydims (getd ’ernie))
(t 3 4)

; store in ernie[2][2] the list (test list)
–> (store (ernie 2 2) ’(test list))
(test list)

; check to see if it is there
–> (ernie 2 2)
(test list)

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

2.5.4. array manipulation

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

RETURNS: the second argument to the function.

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

(store ’l_arexp ’g_val**)**

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

RETURNS: g_val

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

(fillarray ’s_array ’l_itms**)**

RETURNS: s_array

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

2.6. Hunks

Hunks are vector-like objects whose size can range from 1 to 128 elements. Internally, hunks are allocated in sizes which 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*) which will overwrite the distinguished symbol.

2.6.1. hunk creation

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

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

NOTE: the maximum size of a hunk is 128.

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

(makhunk ’xl_arg**)**

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

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

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

(*makhunk ’x_arg**)**

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

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

2.6.2. hunk accessor

(cxr ’x_ind ’h_hunk**)**

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

(hunk-to-list ’h_hunk**)**

RETURNS: a list consisting of the elements of h_hunk.

2.6.3. hunk manipulators

(rplacx ’x_ind ’h_hunk ’g_val**)**
(*rplacx ’x_ind ’h_hunk ’g_val**)**

RETURNS: h_hunk

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

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

(hunksize ’h_arg**)**

RETURNS: the size of the hunk h_arg.

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

2.7. Bcds

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

(getdisc ’y_bcd**)**
(getentry ’y_bcd**)**

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

(putdisc ’y_func ’s_discipline**)**

RETURNS: s_discipline

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

2.8. Structures

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

2.8.1. assoc list

An ‘assoc list’ (or alist) is a common lisp data structure. It has the form

((key1 . value1) (key2 . value2) (key3 . value3) ... (keyn . valuen))

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

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

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

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

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

NOTE: sassoc is written as a macro.

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

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

NOTE: sassq is written as a macro.

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

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

; we 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 will always work
-> (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: new list structure is returned to prevent modification of l_exp. When a substitution is made, a copy of the value to substitute in is not made.

2.8.2. property list

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

(plist 's_name**)**

RETURNS: the property list of s_name.

(setplist 's_atm 'l_plist**)**

RETURNS: l_plist.

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

(get 'ls_name 'g_ind**)**

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

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

(getl 'ls_name 'l_indicators**)**

RETURNS: the property list ls_name beginning at the first indicator which 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 it 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
; use of a disembodied property list:
-> (get '(nil fateman rjf sklower kls foderaro jkf) 'sklower)
kls
```

2.8.3. tconc structure

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

(tconc 'l_ptr 'g_x**)**

WHERE: l_ptr is a tconc structure.

RETURNS: l_ptr with g_x added to the end.

(lconc 'l_ptr 'l_x**)**

WHERE: l_ptr is a tconc structure.

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

```
; A tconc structure can be initialized in two  ways.
; nil can be given to tconc in which case tconc will generate
; 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)                         ; no change
-> (lconc foo '(2 3 4))
((1 2 3 4) 4)
```

2.8.4. fclosures

An fclosure is a functional object which 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 which 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* will not change.

(fclosure-function ’v_fclosure**)**

RETURNS: the functional object part of the fclosure.

(fclosurep ’v_fclosure**)**

RETURNS: t iff the argument is an fclosure.

(symeval-in-fclosure ’v_fclosure ’s_symbol**)**

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

(set-in-fclosure ’v_fclosure ’s_symbol ’g_newvalue**)**

RETURNS: g_newvalue.

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

2.9. Random functions

The following functions don’t fall into any of the classifications above.

(bcdad ’s_funcname**)**

RETURNS: a fixnum which 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 which 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 will only macroexpand expressions which 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* is dependent on the type of the g_arg*i* although one will 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. What this means is 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 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 which x points to. The former method is used almost all of the time, the latter is used very rarely and has the potential to 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 will point to the same box. If you now *(replace x 12345)*, then y will also have 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 will 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 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 being put on property lists, among others. Setf will allow the user 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 will be 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 will be used.

CHAPTER 3

Arithmetic Functions

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

An attempt to divide or to generate a floating point result outside of the range of floating point numbers will cause a floating exception signal from the UNIX operating system. The user 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 will be converted to a bignum. If any of the arguments are flonums, the partial sum will be converted to a flonum when that argument is processed and the result will thus be a flonum. Currently, if in the process of doing the addition a bignum must be converted into a flonum an error message will result.

(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 details and restrictions of data type coercion. A divide by zero will cause a floating exception interrupt -- see the description of the *signal* function.

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

RETURNS: the integer part of i_x / i_y.

(Divide 'i_dividend 'i_divisor**)**

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

NOTE: this is restricted to integer division.

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

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

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

3.2. predicates

(numberp 'g_arg**)**

(numbp 'g_arg**)**

RETURNS: t iff g_arg is a number (fixnum, flonum or bignum).

(fixp 'g_arg**)**

RETURNS: t iff g_arg is a fixnum or bignum.

(floatp 'g_arg**)**

RETURNS: t iff g_arg is a flonum.

(evenp 'x_arg**)**

RETURNS: t iff x_arg is even.

(oddp 'x_arg**)**

RETURNS: t iff x_arg is odd.

(zerop 'g_arg**)**

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

(onep 'g_arg**)**

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

(plusp 'n_arg**)**

RETURNS: t iff n_arg is greater than zero.

(minusp 'g_arg**)**

RETURNS: t iff g_arg is a negative number.

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

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

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

(= ’fx_arg1 ’fx_arg2**)**

(=& ’x_arg1 ’x_arg2**)**

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

3.3. Trignometric Functions

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

(cos ’fx_angle**)**

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

(sin ’fx_angle**)**

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

(acos ’fx_arg**)**

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

(asin ’fx_arg**)**

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

(atan ’fx_arg1 ’fx_arg2**)**

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

3.4. Bignum/Fixnum Manipulation

(haipart bx_number x_bits**)**

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

(haulong bx_number**)**

RETURNS: the number of significant bits in bx_number.

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

(bignum-leftshift bx_arg x_amount**)**

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

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

(sticky-bignum-leftshift 'bx_arg 'x_amount**)**

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

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

3.5. Bit Manipulation

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

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

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

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

(lsh 'x_val 'x_amt**)**

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

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

(rot 'x_val 'x_amt**)**

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

3.6. Other Functions

As noted above, some of the following functions are inherited from the host math library, with all their virtues and vices.

(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 will be 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* will round down. Currently, if n_arg is a flonum larger than the size of a fixnum, this will fail.

(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 will occur.

(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 will have 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

(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 will cease with the first nil encountered.

(apply 'u_func 'l_args**)**

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

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

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

; we will define plus1 as a macro which will be equivalent to add1
–> (def plus1 (macro (arg) (list 'add1 (cadr arg))))
plus1
–> (plus1 3)
4

; now if we apply a macro we obtain the form it changes to.
–> (apply 'plus1 '(plus1 3))
(add1 3)

; if we 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 will work
–> (apply 'plus1 '(foo 3))
(add1 3)
```

(arg [’x_numb]**)**

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

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

(break [g_message [’g_pred]]**)**

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

RETURNS: the value of *(*break ’g_pred ’g_message)*

(*break ’g_pred ’g_message**)**

RETURNS: nil immediately if g_pred is nil, else 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 which allows one to interactively debug a program. To continue execution from a break you can use the *return* function. to return to top level or another break level, you can use *retbrk* or *reset*.

(caseq ’g_key-form l_clause1 ...**)**

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

NOTE: The way caseq works is that it evaluates g_key-form, yielding a value we will call 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. 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 the value thrown during the evaluation of g_exp.

SIDE EFFECT: this first sets up a 'catch frame' on the lisp runtime stack. Then it begins to evaluate g_exp. If g_exp evaluates normally, its value is returned. If, however, a value is thrown during the evaluation of g_exp then this *catch will return 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 will not catch an error but a catch whose tag is the error type will catch that type of error. See Chapter 10 for more information.

(comment [g_arg ...]**)**

RETURNS: the symbol comment.

NOTE: This does absolutely nothing.

(cond [l_clause1 ...]**)**

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

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

(cvttointlisp)

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

(cvttofranzlisp)

SIDE EFFECT: FRANZ LISP's default syntax is reinstated. One would 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, 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 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 above 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, see the definitions of nthcdr and nth in /usr/lib/lisp/common2.l

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

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

RETURNS: s_name

SIDE EFFECT: This defines the function s_name.

NOTE: this exists for Maclisp compatibility, it is just a macro which changes the defun form to the def form. An s_mtype of fexpr is converted to nlambda and of expr to lambda. Macro remains the same. If ls_arg1 is a non-nil symbol, then the type is assumed to be lexpr and ls_arg1 is the symbol which is bound to the number of args when the function is entered.
For compatibility with the Lisp Machine Lisp, there are three types of optional parameters that can occur in ls_argl: *&optional* declares that the following symbols are optional, and may or may not appear in the argument list to the function, *&rest symbol* declares that all

forms in the function call that are not accounted for by previous lambda bindings are to be assigned to *symbol*, and *&aux form1 ... formn* declares that the *formi* are either symbols, in which case they are lambda bound to **nil**, or lists, in which case the first element of the list is lambda bound to the second, evaluated element.

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

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

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

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

RETURNS: s_variable.

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

SIDE EFFECT: This declares s_variable to be special. If g_init is present and s_variable is unbound when the file is read in, s_variable will be 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 will be 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 will be evaluated only once and the do will return 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 which follow constitute the do body. A do body is like a prog body and thus may have labels and one may 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 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 which can be used when there is only one do variable. It is described next.

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

(do s_name g_init g_repeat g_test g_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

(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, it looks for a 'version' property under file*i*'s property list. Thus 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, franzlisp.

SIDE EFFECT: *environment-maclisp* sets the environment to that which 'liszt -m' would generate.

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* will return s_value instead of nil. If the second arg is given, then it must be nil (MAClisp compatibility).

(error ['s_message1 ['s_message2]]**)**

RETURNS: nothing (it never returns).

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

(errset g_expr [s_flag]**)**

RETURNS: a list of one element, which is the value resulting from evaluating g_expr. If an error occurs during the evaluation of g_expr, then the locus of control will return to the *errset* which will then return 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 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 this 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. We continually evaluate g_car until we end up with a symbol with a non-null function binding or a non-symbol. Call what we end up with: 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 we shall call g_functype, must be either lambda, nlambda, macro or lexpr. If g_func is a binary, then its discipline, which we shall call 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 will determine 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 which 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* to order to evaluate variables in the same context as this evaluation. If type is 'eval' then expression will have the form *(function-name arg1 ...)*. If type is 'apply' then expression will have the form *(function-name (arg1 ...))*. np-index and lbot-index are pointers into the argument stack (also known as the *namestack* array) at the time of call. lbot-index points to the first argument, np-index points one beyond the last argument.
In order for there to be enough information for *evalframe* to return, you must call *(*rset t)*.

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

(evalhook 'g_form 'su_evalfunc ['su_funcallfunc]**)**

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

NOTE: As explained in §14.4, the function *eval* may pass the job of evaluating a form to a user 'hook' function when various switches are set. The hook function normally prints the form to be evaluated on the terminal and then evaluates it by calling *evalhook*. *Evalhook* does the lambda binding mentioned above 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* will 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.

(execе '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 will not return, instead the lisp system will be overlaid by the new command.

(freturn 'x_pdl-pointer 'g_retval**)**

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

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

(frexp 'f_arg**)**

RETURNS: a list cell *(exponent . mantissa)* which represents the given flonum

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

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

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

NOTE: If u_func is a macro or nlambda then there should be only one g_arg. *funcall* is the function which the evaluator uses to evaluate lists. If *foo* is a lambda or lexpr or array, then *(funcall 'foo 'a 'b 'c)* is equivalent to *(foo 'a 'b 'c)*. If *foo* is a 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 *funcall*ing the *(car l_form)* on the already evaluated arguments in the *(cdr l_form)* after lambda binding 'funcallhook' to su_funcallfunc and, if it is given, lambda binding 'evalhook' to su_evalhook.

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* will actually be 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 an symbol with a function binding otherwise u_func is returned.

(getdisc 'y_func**)**

RETURNS: the discipline of the machine coded function (either lambda, nlambda or macro).

(go g_labexp**)**

WHERE: g_labexp is either a symbol or an expression.

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

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

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

NOTE: The various forms of *if* are intended to be a more readable conditional statement, to be used in place of *cond*. There are two varieties of *if*, with keywords, and without. 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 optional else-clause. The predicate is evaluated, and if the result is non-nil, the then-clause will be performed, in the sense described below. Otherwise, (i.e. the result of the predicate evaluation was precisely nil), the else-clause will be performed.

Then-clauses will either consist entirely of the single keyword **thenret**, or will start with the keyword **then**, and be 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 than 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 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* will do 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 element of list arg are stack 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.

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

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

(mfunction t_entry 's_disc**)**

RETURNS: a lisp object of type binary composed of t_entry and s_disc.

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

(oblist)

RETURNS: a list of all symbols on the oblist.

(or [g_arg1 ...]**)**

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

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

(prog l_vrbls g_exp1 ...**)**

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

NOTE: the local variables are lambda-bound to nil, then the g_exp*i* are evaluated from left to right. This is a prog body (obviously) and this means than 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 which 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 will result in an illegal memory reference.

(purep 'g_exp**)**

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

(putd 's_name 'u_func**)**

RETURNS: u_func

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

(return ['g_val]**)**

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

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

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

NOTE: This function is just like *caseq* (see above), 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.

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* will still be evaluated. This is useful if g_protected does something sensitive which must be cleaned up whether or not g_protected completes.

CHAPTER 5

Input/Output

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

If a port argument is not supplied to a function which requires one, or if a bad port argument (such as nil) is given, then FRANZ LISP will use 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 will be 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 which goes to the standard output will also go to the port **ptport** if it is a valid port. Output destined for the standard output will not reach the standard output if the symbol **ˆw** is non nil (although it will still go to **ptport** if **ptport** is a valid port).

Some of the functions listed below reference files directly. FRANZ LISP has borrowed a convenient shorthand notation from */bin/csh*, 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.

FRANZ LISP keeps a cache of user home directory information, to minimize searching the password file. Tilde-expansion is performed in the following functions: *cfasl, chdir, fasl, ffasl, fileopen, infile, load, outfile, probef, sys:access, sys:unlink*.

(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 will be set to point to st_entry, so that when the lisp function s_funcname is called, st_entry will be 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 will be searched to resolve external references. The form of st_library should be something like "-lm". The C library (" -lc ") is always searched so when loading in a C file you probably won't need to specify a library. For Fortran files, you should specify "-lF77" and if you are doing any I/O, the library entry should be "-lI77 -lF77". For Pascal files "-lpc" is required.

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 which 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 UNIX 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 which are just printed literally and it may contain special formatting commands preceded by a percent sign. The complete set of formatting characters is described in the UNIX 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.

(ex [s_filename]**)**
(vi [s_filename]**)**
(exl [s_filename]**)**
(vil [s_filename]**)**

RETURNS: nil

SIDE EFFECT: The lisp system starts up an editor on the file named as the argument. It will try appending .l to the file if it can't find it. The functions *exl* and *vil* will load the file after you finish editing it. These functions will also remember the name of the file so that on subsequent invocations, you don't need to provide the argument.

NOTE: These functions do not evaluate their argument.

(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* will 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 will be appended. If g_warn is non nil and if a function is loaded from the file which is already defined, then a warning message will be 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 will call *fasl* if it finds a file with the same root name and a '.o' extension. In most cases the user would be better off using the function *load* rather than calling *fasl* directly.

(ffasl 'st_file 'st_entry 'st_funcname ['st_discipline ['st_library]]**)**

RETURNS: the binary object created.

SIDE EFFECT: the Fortran object file st_file is loaded into the lisp system. St_entry should be an entry point in the file just loaded. A binary object will be created and its entry field will be set to point to st_entry. The discipline field of the binary will be set to st_discipline or "subroutine" by default. If st_library is present and non-null, then after st_file is loaded, the libraries given in st_library will be searched to resolve external references. The form of st_library should be something like "-lS -ltermcap". In any case, the standard Fortran libraries will be searched also to resolve external references.

NOTE: in F77 on Unix, the entry point for the fortran function foo is named '_foo_'.

(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 will be at position x_pos.

(filestat 'st_filename**)**

RETURNS: a vector containing various numbers which the UNIX operating system assigns to files. if the file doesn't exist, an error is invoked. Use *probef* to determine if the file exists.

NOTE: The individual entries can be accesed by mnemonic functions of the form filestat:*field*, where field may be any of atime, ctime, dev, gid, ino, mode,mtime, nlink, rdev, size, type, uid. See the UNIX 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 will return 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.

(fileopen 'st_filename 'st_mode**)**

RETURNS: a port for reading or writing (depending on st_mode) the file st_name.

SIDE EFFECT: the given file is opened (or created if opened for writing and it doesn't yet exist).

NOTE: this function call provides a direct interface to the operating system's fopen function. The mode may be more than just "r" for read, "w" for write or "a" for append. The modes "r+", "w+" and "a+" permit both reading and writing on a port provided that *fseek* is done between changes in direction. See the UNIX manual description of fopen for more details. This routine does not look through a search path for a given file.

(fseek ’p_port ’x_offset ’x_flag**)**

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

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

(infile ’s_filename**)**

RETURNS: a port ready to read s_filename.

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

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

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

which will set myport to the port to read from if the file exists or will print a message if it couldn’t open it and also set 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* will search 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 which *load* uses is the value of *(status load-search-path)*. The default is (|.| /usr/lib/lisp) which means look in the current directory first and then /usr/lib/lisp. The file which *load* looks for depends on the last two characters of s_filename. If s_filename ends with ".l" then *load* will only look for a file name s_filename and will assume that this is a FRANZ LISP source file. If s_filename ends with ".o" then *load* will only look for a file named s_filename and will assume that this is a FRANZ LISP object file to be *fasl*ed in. Otherwise, *load* will first look for s_filename.o, then s_filename.l and finally s_filename itself. If it finds s_filename.o it will assume that this is an object file, otherwise it will assume 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 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 will be opened in append mode, that is the current contents will not be lost and the next data will be 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 don't print

(princ 'g_arg ['p_port]**)**

EQUIVALENT TO: patom.

(print 'g_arg ['p_port]**)**

RETURNS: nil

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

(probef 'st_file**)**

RETURNS: t iff the file st_file exists.

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

(pp-form 'g_form ['p_port]**)**

RETURNS: t

SIDE EFFECT: g_form is pretty-printed to the port p_port (or poport if p_port is not given). This is the function which *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, c.f. *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 will occur if the reader is given an ill formed expression. The most common error is too many right parentheses (note that this is not considered an error in Maclisp).

(readc [’p_port [’g_eof]]**)**

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

(readlist ’l_arg**)**

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

(removeaddress ’s_name1 [’s_name2 ...]**)**

RETURNS: nil

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

(resetio)

RETURNS: nil

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

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

RETURNS: t

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

(sload ’s_file**)**

SIDE EFFECT: the file s_file (in the current directory) is opened for reading and each form is read, printed and evaluated. If the form is recognizable as a function definition, only its name will be 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 narrow down 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 will also drain the port.

(terpri [’p_port]**)**

EQUIVALENT TO: terpr.

(tilde-expand ’st_filename**)**

RETURNS: a symbol whose pname is the tilde-expansion of the argument, (as discussed at the beginning of this chapter). If the argument does not begin with a tilde, the argument itself is returned.

(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 actually read the character, it just peeks at it.

(tyo ’x_char [’p_port]**)**

RETURNS: x_char.

SIDE EFFECT: the character whose fixnum representation is x_code, is printed as a 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* will read it first.

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

(username-to-dir ’st_name**)**

RETURNS: the home directory of the given user. The result is stored, to avoid unnecessarily searching the password file.

(zapline)

RETURNS: nil

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

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

CHAPTER 6

System Functions

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

(allocate 's_type 'x_pages**)**

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

RETURNS: x_pages.

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

(argv 'x_argnumb**)**

RETURNS: a symbol whose pname is the x_argnumb*th* argument (starting at 0) on the command line which 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 will occasionally miss the names of compiled lisp functions due to incomplete information on the stack. If you are tracing compiled code, then *baktrace* won't be able to interpret the stack unless *(sstatus translink nil)* was done. See the function *showstack* for another way of printing the lisp runtime stack. This misspelling is from Maclisp.

(chdir 's_path**)**

RETURNS: t iff the system call succeeds.

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

NOTE: *chdir* follows the standard UNIX conventions, if s_path does not begin with a slash, the default path is changed to the current path with s_path appended. *Chdir* employs tilde-expansion (discussed in Chapter 5).

(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 will be in a lisp in the same state as when the dumplisp was done.

NOTE: dumplisp will fail if one tries to write over the current running file. UNIX 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 is discussed in §12.3.2.1 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).

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 below), 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                               ;; we 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 doesn't have to call *gc* since garbage collection occurs automatically whenever internal free lists are exhausted.

(gcafter s_type**)**

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

NOTE: this function is called by the garbage collector after a garbage collection which was caused by running out of data type s_type. This function should determine if more space need be allocated and if so should allocate it. There is a default gcafter function but users who want control over space allocation can define their own -- but note that it must be an nlambda.

(getenv 's_name**)**

RETURNS: a symbol whose pname is the value of s_name in the current UNIX environment. If s_name doesn't exist in the current environment, a symbol with a null pname is returned.

(hashtabstat)

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

NOTE: the oblist is stored a hash table of buckets. Ideally there would be the same number of symbols in each bucket.

(help [sx_arg]**)**

SIDE EFFECT: If sx_arg is a symbol then the portion of this manual beginning with the description of sx_arg is printed on the terminal. If sx_arg is a fixnum or the name of one of the appendicies, that chapter or appendix is printed on the terminal. If no argument is provided, *help* prints the options that it recognizes. The program 'more' is used to print the manual on the terminal; it will stop after each page and will continue after the space key is pressed.

(include s_filename**)**

RETURNS: nil

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

NOTE: this is similar to load except 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 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.

(monitor [’xs_maxaddr]**)**

RETURNS: t

SIDE EFFECT: If xs_maxaddr is t then profiling of the entire lisp system is begun. If xs_maxaddr is a fixnum then profiling is done only up to address xs_maxaddr. If xs_maxaddr is not given, then profiling is stopped and the data obtained is written to the file ’mon.out’ where it can be analyzed with the UNIX ’prof’ program.

NOTE: this function only works if the lisp system has been compiled in a special way, otherwise, an error is invoked.

(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* will return 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 *opval* associates with a data type using *opval*.
If s_arg is *pagelimit* then *opval* will return (and set if g_newval is given) the maximum amount of lisp data pages it will allocate. This limit should remain small unless you know your program requires lots of space as this limit will catch 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 (it first tries /bin/csh, then it tries /bin/sh if /bin/csh doesn't exist). 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 two or three arguments are given, **process* starts the process and then returns a list which, depending on the value of g_readp and g_writep, may contain i/o ports for communcating with the new process. If g_writep is non-null, then a port will be created which the lisp program can use to send characters to the new process. If g_readp is non-null, then a port will be created which 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* were written to perform the common functions.

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

RETURNS: a port which 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 which 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 which is the exit code when s_prgm dies. If the optional arguments are present, it returns a fixnum which is the process id of the child.

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

SIDE EFFECT: If s_frompipe and s_topipe are given, they are bound to ports which are pipes which 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 iff 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*th*s of a second. The first number includes the second number. The amount of time used by garbage collection is not recorded until the first call to ptime. This is done to prevent overhead when the user is not interested in garbage collection times.

(reset)

SIDE EFFECT: the lisp runtime stack is cleared and the system restarts at the top level by executing a *(funcall top-level nil)*.

(restorelisp 's_name**)**

SIDE EFFECT: this reads in file s_name (which was created by *savelisp*) and then does a *(reset)*.

NOTE: This is only used on VMS systems where *dumplisp* cannot be used.

(retbrk ['x_level]**)**

WHERE: x_level is a small integer of either sign.

SIDE EFFECT: The default error handler keeps a notion of the current level of the error caught. If x_level is negative, control is thrown to this default error handler whose level is that many less than the present, or to *top-level* if there aren't enough. If x_level is non-negative, control is passed to the handler at that level. If x_level is not present, the value -1 is taken by default.

(*rset 'g_flag**)**

RETURNS: g_flag

SIDE EFFECT: If g_flag is non nil then the lisp system will maintain 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)*

(savelisp 's_name**)**

RETURNS: t

SIDE EFFECT: the state of the Lisp system is saved in the file s_name. It can be read in by *restorelisp*.

NOTE: This is only used on VMS systems where *dumplisp* cannot be used.

(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 will actually allocate 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 the most that showstack will show is the function name and it may miss some functions.

(signal 'x_signum 's_name**)**

RETURNS: nil if no previous call to signal has been made, or the previously installed s_name.

SIDE EFFECT: this declares that the function named s_name will handle the signal number x_signum. If s_name is nil, the signal is ignored. Presently only four UNIX 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 will append 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 which no one wants to handle, a *reset* will be 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 will be 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 UNIX that a segmentation violation or bus error should cause a core dump. If g_val is non nil then FRANZ LISP will catch those errors and print a message advising the user to reset.

NOTE: The initial value for this flag is nil, and only those knowledgeable of the innards of the lisp system should ever set this flag non nil.

(sstatus dumpmode x_val**)**

RETURNS: x_val

SIDE EFFECT: All subsequent *dumplisp*'s will be done in mode x_val. x_val may be either 413 or 410 (decimal).

NOTE: the advantage of mode 413 is that the dumped Lisp can be demand paged in when first started, which will make it start faster and disrupt other users less. The initial value is 413.

(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 gcstrings g_val**)**

RETURNS: g_val

SIDE EFFECT: if g_val is non-null, and if string garbage collection was enabled when the lisp system was compiled, string space will be garbage collected.

NOTE: the default value for this is nil since in most applications garbage collecting strings is a waste of time.

(sstatus ignoreeof g_val**)**

RETURNS: g_val

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

(sstatus nofeature g_val**)**

RETURNS: g_val

SIDE EFFECT: g_val is removed from the status features list if it was 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 will not cause the fast links to be set up. If g_val is the symbol *on* then all possible transfer table entries will be linked and the flag will be 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 reader will be 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 below

(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 below.

(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 which are present in this system. You add to this list with *(sstatus feature 'g_val)* and test if feature g_feat is present with *(status feature 'g_feat)*.

(status isatty)

RETURNS: t iff the standard input is a terminal.

(status localtime)

RETURNS: a list of fixnums representing the current time.

EXAMPLE: *(status localtime)* = (3 51 13 31 6 81 5 211 1)
means 3*rd* second, 51*st* minute, 13*th* hour (1 p.m), 31*st* day, month 6 (0 = January), year 81 (0 = 1900), day of the week 5 (0 = Sunday), 211*th* day of the year and daylight savings time is 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 which transfer table entries point to but which are not defined at this point.

NOTE: Some of the undefined functions listed could be arrays which have yet to be created.

(status version)

RETURNS: a string which is the current lisp version name.

EXAMPLE: *(status version)* = "Franz Lisp, Opus 38.61"

(syscall 'x_index ['xst_arg1 ...]**)**

RETURNS: the result of issuing the UNIX system call number x_index with arguments xst_arg*i*.

NOTE: The UNIX system calls are described in section 2 of the UNIX Programmer's manual. If xst_arg*i* is a fixnum, then its value is passed as an argument, if it is a symbol then its pname is passed and finally if it is a string then the string itself is passed as an argument. Some useful syscalls are:
(syscall 20) returns process id.
(syscall 13) returns the number of seconds since Jan 1, 1970.
(syscall 10 'foo) will unlink (delete) the file foo.

(sys:access 'st_filename 'x_mode**)**
(sys:chmod 'st_filename 'x_mode**)**
(sys:gethostname)
(sys:getpid)
(sys:getpwnam 'st_username**)**
(sys:link 'st_oldfilename 'st_newfilename**)**
(sys:time)
(sys:unlink 'st_filename**)**

NOTE: We have been warned that the actual system call numbers may vary among different UNIX systems. Users concerned about portability may wish to use this group of functions. Another advantage is that tilde-expansion is performed on all filename arguments. These functions do what is described in the system call section of your UNIX manual.

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

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

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

(top-level)

RETURNS: nothing (it never returns)

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

(wait)

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

CHAPTER 7

The Lisp Reader

7.1. Introduction

The *read* function is responsible for converting a stream of characters into a Lisp expression. *Read* is table driven and the table it uses is called a *readtable.* The *print* function does the inverse of *read*; it converts a Lisp expression into a stream of characters. Typically the conversion is done in such a way that if that stream of characters were read by *read*, the result would be an expression equal to the one *print* was 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 above.

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 below.

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 will be 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: always escape a symbol with this character in it, only escape a symbol if this is the only character in the symbol, and only escape a symbol if this is the first character in the symbol. [note: The printer will always escape a symbol which, if printed out, would look 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 which don't start tokens (such as blanks and tabs). Collecting means gathering the characters which 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 will continue 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 will continue collecting until it runs into a character whose syntax class has the *separator* property. (That last character will be pushed back into the input buffer and will be the first character read next time.) Now the reader goes into the processing phase, checking to see if the token it read is a number or symbol. It is important to note that after the first character is collected the component of the syntax class which tells the reader to stop collecting is the *separator* property, not the character class.

If the character class of the character which 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* will act like *ccharacter* if the following token is not a *separator.* The processing which 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 ˆH !#$%&*,/:;<=>?@ˆ_‘{}˜
standard readtable:A-Z a-z ˆH !$%&*/:;<=>?@ˆ_{}˜

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 which 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 [e.g. (a . (b . nil)) is the same as (a b)]. *cperiod* is also used in numbers as described above.

cseparator raw readtable:ˆI-ˆM esc space
standard readtable:ˆI-ˆM 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 to 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 will not separate symbols.

cstring-delimiter raw readtable:"
standard readtable:"

This is the same as *csymbol-delimiter* except 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 which 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 below.

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 to 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 is 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 will only be 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. The user may add syntax classes with *add-syntax-class*. For each syntax class, we list the properties of the class and which characters have this syntax class by default. 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 ˆH !#$%&*,/:;<=>?@ˆ_‘{}˜
ccharacter standard readtable:A-Z a-z ˆH !$%&*/:;<=>?@ˆ_{}˜

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

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

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

vright-bracket raw readtable:]
cright-bracket standard readtable:]
escape-always

vperiod raw readtable:.
cperiod standard readtable:.
escape-when-unique

vseparator raw readtable:ˆI-ˆM esc space
cseparator standard readtable:ˆI-ˆM esc space
escape-always

vsingle-quote raw readtable:´
csingle-quote standard readtable:´
escape-always

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	
vsplicing-macro	raw readtable:none
csplicing-macro	standard readtable:#;
escape-always	
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	
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	

7.6. Character Macros

Character macros are user written functions which 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 which 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 wasn't 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 couldn't 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 (i.e. not collecting elements of list), then it is illegal for a splicing macro to return more then one element in the list. The major advantage of a splicing macro over a normal macro is the ability of the splicing macro to return nothing. The comment character (usually ;) is a splicing macro bound to a function which reads to the end of the line and always returns nil. Here is the previous example written as a splicing macro

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

7.6.1.3. Infix

Infix macros are passed a *conc* structure representing what has been read so far. Briefly, a tconc structure is a single list cell whose car points to a list and whose cdr points to the last list cell in that list. The interpretation by the reader of the value returned by an infix macro depends on whether the macro is called while the reader is constructing a list or whether it is called at the top level of the reader. If the macro is called while a list is being constructed, then the value returned should be a tconc structure. The car of that structure replaces the list of elements that the reader has been collecting. If the macro is called at top level, then it will be 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 (i.e. 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 would be defined using *cmacro*, *csplicing-macro*, or *cinfix-macro* and 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, i.e the macro character is preceeded by zero or more *cseparator*s and followed by a *separator.* A syntax class for a *when unique* macro would be defined using *csingle-macro*, *csingle-splicing-macro*, or *csingle-infix-macro* and not including the *separator* property. The syntax classes so defined are **vsingle-macro**, **vsingle-splicing-macro**, and **vsingle-infix-macro**.

7.7. Functions

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

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

RETURNS: t

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

NOTE: The symbolic syntax codes are new to Opus 38. For compatibility, s_synclass can be one of the fixnum syntax codes which 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 Opus 38. It supercedes *(status syntax)* which 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 classes mentioned above. l_properties may contain one of the escape properties: *escape-always*, *escape-when-unique*, or *escape-when-first*. l_properties may contain the *separator* property. After a syntax class has been defined with *add-syntax-class*, the *setsyntax* function can be used to give characters that syntax class.

```
; Define a non-separating macro character.
; This type of macro character is used in UCI-Lisp, and
; it corresponds to a  FIRST MACRO in Interlisp
-> (add-syntax-class 'vuci-macro '(cmacro escape-when-first))
vuci-macro
->
```

CHAPTER 8

Functions, Fclosures, and Macros

8.1. valid function objects

There are many different objects which can occupy the function field of a symbol object. Table 8.1, on the following page, shows all of the possibilities, how to recognize them, and where to look for documentation.

8.2. functions

The basic Lisp function is the lambda function. When a lambda function is called, the actual arguments are evaluated from left to right and are lambda-bound to the formal parameters of the lambda function.

An nlambda function is usually used for functions which are invoked by the user at top level. Some built-in functions which evaluate their arguments in special ways are also nlambdas (e.g *cond*, *do*, *or*). When an nlambda function is called, the list of unevaluated arguments is lambda bound to the single formal parameter of the nlambda function.

Some programmers will use an nlambda function when they are not sure how many arguments will be passed. Then, the first thing the nlambda function does is map *eval* over the list of unevaluated arguments it has been passed. This is usually the wrong thing to do, as it will not work compiled if any of the arguments are local variables. The solution is to use a lexpr. When a lexpr function is called, the arguments are evaluated and a fixnum whose value is the number of arguments is lambda-bound to the single formal parameter of the lexpr function. The lexpr can then access the arguments using the *arg* function.

When a function is compiled, *special* declarations may be needed to preserve its behavior. An argument is not lambda-bound to the name of the corresponding formal parameter unless that formal parameter has been declared *special* (see §12.3.2.2).

Lambda and lexpr functions both compile into a binary object with a discipline of lambda. However, a compiled lexpr still acts like an interpreted lexpr.

8.3. macros

An important feature of Lisp is its ability to manipulate programs as data. As a result of this, most Lisp implementations have very powerful macro facilities. The Lisp language's macro facility can be used to incorporate popular features of the other languages into Lisp. For example, there are macro packages which allow one to create records (ala Pascal) and refer to elements of those records by the field names. The *struct* package imported from Maclisp does this. Another popular use for macros is to create more readable control structures which expand into *cond*, *or* and *and*. One such example is the If macro. It allows you to write

```
(If (equal numb 0) then (print 'zero) (terpr)
 elseif (equal numb 1) then (print 'one) (terpr)
```

informal name	object type	documentation
interpreted lambda function	list with *car* *eq* to lambda	8.2
interpreted nlambda function	list with *car* *eq* to nlambda	8.2
interpreted lexpr function	list with *car* *eq* to lexpr	8.2
interpreted macro	list with *car* *eq* to macro	8.3
fclosure	vector with *vprop* *eq* to fclosure	8.4
compiled lambda or lexpr function	binary with discipline *eq* to lambda	8.2
compiled nlambda function	binary with discipline *eq* to nlambda	8.2
compiled macro	binary with discipline *eq* to macro	8.3
foreign subroutine	binary with discipline of ‘‘subroutine’’†	8.5
foreign function	binary with discipline of ‘‘function’’†	8.5
foreign integer function	binary with discipline of ‘‘integer-function’’†	8.5
foreign real function	binary with discipline of ‘‘real-function’’†	8.5
foreign C function	binary with discipline of ‘‘c-function’’†	8.5
foreign double function	binary with discipline of ‘‘double-c-function’’†	8.5
foreign structure function	binary with discipline of ‘‘vector-c-function’’†	8.5
array	array object	9

Table 8.1

else (print ’|I give up|))

which expands to

```
(cond
   ((equal numb 0) (print ’zero) (terpr))
   ((equal numb 1) (print ’one) (terpr))
   (t (print ’|I give up|)))
```

†Only the first character of the string is significant (i.e ‘‘s’’ is ok for ‘‘subroutine’’)

8.3.1. macro forms

A macro is a function which 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, we 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 chops off the car of the argument with *cdr*, cons' a *car* at the beginning of the list and returns *(car ’(a b c))*, which *eval* evaluates. The value *a* is returned as the value of *(first ’(a b c))*. Thus whenever *eval* tries to evaluate a list whose car has a macro definition it ends up doing (at least) two operations, the first of which is a call to the macro to let it macro expand the form, and the other 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 will expand is to use *apply* as shown on the third input line above.

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 we find ourselves often writing code like *(setq stack (cons newelt stack)*. We could define a macro named *push* to do this for us. 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)* will expand to the form mentioned above. The same macro written using defmacro would be:

```
−> (defmacro push (value stack)
    (list ’setq ,stack (list ’cons ,value ,stack)))
push
```

Defmacro allows you to name the arguments of the macro call, and makes the macro definition look more like a function definition.

8.3.3. the backquote character macro

The default syntax for FRANZ LISP has four characters with associated character macros. One is semicolon for comments. Two others are the backquote and comma which are used by the backquote character macro. The fourth is the sharp sign macro described in the next section.

The backquote macro is used to create lists where many of the elements are fixed (quoted). This makes it very useful for creating macro definitions. In the simplest case, a backquote acts just like a single quote:

```
–>‘(a b c d e)
(a b c d e)
```

If a comma precedes an element of a backquoted list then that element is evaluated and its value is put in the list.

```
–>(setq d ’(x y z))
(x y z)
–>‘(a b c ,d e)
(a b c (x y z) e)
```

If a comma followed by an at sign precedes an element in a backquoted list, then that element is evaluated and spliced into the list with *append*.

```
–>‘(a b c ,@d e)
(a b c x y z e)
```

Once a list begins with a backquote, the commas may appear anywhere in the list as this example shows:

```
–>‘(a b (c d ,(cdr d)) (e f (g h ,@(cddr d) ,@d)))
(a b (c d (y z)) (e f (g h z x y z)))
```

It is also possible and sometimes even useful to use the backquote macro within itself. As a final demonstration of the backquote macro, we shall define the first and push macros using all the power at our disposal: defmacro and the backquote macro.

```
–>(defmacro first (list) ‘(car ,list))
first
–>(defmacro push (value stack) ‘(setq ,stack (cons ,value ,stack)))
stack
```

8.3.4. sharp sign character macro

The sharp sign macro can perform a number of different functions at read time. The character directly following the sharp sign determines which function will be done, and following Lisp s-expressions may serve as arguments.

8.3.4.1. conditional inclusion

If you plan to run one source file in more than one environment then you may want to 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* will only be read in if *when* is true. A symbol in *when* is true only if it appears in the *(status features)* list.

```
; suppose we want to write a program which references a file
; and which can run at ucb, ucsd and cmu where the file naming conventions
; are different.
;
–> (defun howold (name)
        (terpr)
        (load #+(or ucb ucsd) "/usr/lib/lisp/ages.l"
                #+cmu "/usr/lisp/doc/ages.l")
        (patom name)
        (patom " is ")
        (print (cdr (assoc name agefile)))
        (patom "years old")
        (terpr))
```

The form

#–when what

is equivalent to

#+(not when) what

8.3.4.2. fixnum character equivalents

When working with fixnum equivalents of characters, it is often hard to remember the number corresponding to a character. The form

#/c

is equivalent to the fixnum representation of character c.

```
; a function which returns t if the user types y else it returns nil.
;
–> (defun yesorno nil
        (progn (ans)
                (setq ans (tyi))
                (cond ((equal ans #/y) t)
                        (t nil))))
```

8.3.4.3. read time evaluation

Occasionally you want to express a constant as a Lisp expression, yet you don't 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.

```
; a function to test if any of bits 1 3 or 12 are set in a fixnum.
;
−> (defun testit (num)
        (cond ((zerop (boole 1 num #.(+ (lsh 1 1) (lsh 1 3) (lsh 1 12))))
                 nil)
                (t t)))
```

8.4. fclosures

Fclosures are a type of functional object. The purpose is to remember the values of some variables between invocations of the functional object and to protect this data from being inadvertently overwritten by other Lisp functions. Fortran programs usually exhibit this behavior for their variables. (In fact, some versions of Fortran would require the variables to be in COMMON). Thus it is easy to write a linear congruent random number generator in Fortran, merely by keeping the seed as a variable in the function. It is much more risky to do so in Lisp, since any special variable you picked, might be used by some other function. Fclosures are an attempt to provide most of the same functionality as closures in Lisp Machine Lisp, to users of FRANZ LISP. Fclosures are related to closures in this way:

(fclosure ’(a b) ’foo) <==>
        (let ((a a) (b b)) (closure ’(a b) ’foo))

8.4.1. an example

```
% lisp
Franz Lisp, Opus 38.60
−>(defun code (me count)
  (print (list ’in x))
  (setq x (+ 1 x))
  (cond ((greaterp count 1) (funcall me me (sub1 count))))
  (print (list ’out x)))
code
−>(defun tester (object count)
  (funcall object object count) (terpri))
tester
−>(setq x 0)
0
−>(setq z (fclosure ’(x) ’code))
fclosure[8]
−> (tester z 3)
(in 0)(in 1)(in 2)(out 3)(out 3)(out 3)
nil
−>x
0
```

The function *fclosure* creates a new object that we will call an fclosure, (although it is actually a vector). The fclosure contains a functional object, and a set of symbols and values for the symbols. In the above example, the fclosure functional object is the function code. The set of symbols and values just contains the symbol ‘x’ and zero, the value of ‘x’ when the fclosure was created.

When an fclosure is funcall’ed:

1) The Lisp system lambda binds the symbols in the fclosure to their values in the fclosure.

2) It continues the funcall on the functional object of the fclosure.

3) Finally, it un-lambda binds the symbols in the fclosure and at the same time stores the current values of the symbols in the fclosure.

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

```
–> (setq zz (fclosure ’(x) ’code))
fclosure[1]
–> (tester zz 2)
(in 0)(in 1)(out 2)(out 2)
–> (tester zz 2)
(in 2)(in 3)(out 4)(out 4)
–> (tester z 3)
(in 3)(in 4)(in 5)(out 6)(out 6)(out 6)
```

8.4.2. useful functions

Here are some quick some summaries of functions dealing with closures. They are more formally defined in §2.8.4. To recap, fclosures are made by *(fclosure ’l_vars ’g_funcobj)*. l_vars is a list of symbols (not containing nil), g_funcobj is any object that can be funcalled. (Objects which can be funcalled, include compiled Lisp functions, lambda expressions, symbols, foreign functions, etc.) In general, if you want a compiled function to be closed over a variable, you must declare the variable to be special within the function. Another example would be:

(fclosure ’(a b) #’(lambda (x) (plus x a)))

Here, the #’ construction will make the compiler compile the lambda expression.

There are times when you want to share variables between fclosures. This can be done if the fclosures are created at the same time using *fclosure-list*. The function *fclosure-alist* returns an assoc list giving the symbols and values in the fclosure. The predicate *fclosurep* returns t iff its argument is a fclosure. Other functions imported from Lisp Machine Lisp are *symeval-in-fclosure, let-fclosed,* and *set-in-fclosure.* Lastly, the function *fclosure-function* returns the function argument.

8.4.3. internal structure

Currently, closures are implemented as vectors, with property being the symbol fclosure. The functional object is the first entry. The remaining entries are structures which point to the symbols and values for the closure, (with a reference count to determine if a recursive closure is active).

8.5. foreign subroutines and functions

FRANZ LISP has the ability to dynamically load object files produced by other compilers and to call functions defined in those files. These functions are called *foreign* functions.* There are seven types of foreign functions. They are characterized by the type of result they return, and by differences in the interpretation of their arguments. They come from two families: a group suited for languages which pass arguments by reference (e.g. Fortran), and a group suited for languages which pass arguments by value (e.g. C).

There are four types in the first group:

subroutine
This does not return anything. The Lisp system always returns t after calling a subroutine.

function
This returns whatever the function returns. This must be a valid Lisp object or it may cause the Lisp system to fail.

integer-function
This returns an integer which the Lisp system makes into a fixnum and returns.

real-function
This returns a double precision real number which the Lisp system makes into a flonum and returns.

There are three types in the second group:

c-function
This is like an integer function, except for its different interpretation of arguments.

double-c-function
This is like a real-function.

vector-c-function
This is for C functions which return a structure. The first argument to such functions must be a vector (of type vectori), into which the result is stored. The second Lisp argument becomes the first argument to the C function, and so on

A foreign function is accessed through a binary object just like a compiled Lisp function. The difference is that the discipline field of a binary object for a foreign function is a string whose first character is given in the following table:

letter	type
s	subroutine
f	function
i	integer-function
r	real-function.
c	c-function
v	vector-c-function
d	double-c-function

Two functions are provided for setting-up foreign functions. *Cfasl* loads an object file into the Lisp system and sets up one foreign function binary object. If there are more than one function in an object file, *getaddress* can be used to set up additional foreign function objects.

Foreign functions are called just like other functions, e.g *(funname arg1 arg2)*. When a function in the Fortran group is called, the arguments are evaluated and then examined. List, hunk

*This topic is also discussed in Report PAM-124 of the Center for Pure and Applied Mathematics, UCB, entitled ‘‘Parlez-Vous Franz? An Informal Introduction to Interfacing Foreign Functions to Franz LISP’’, by James R. Larus and symbol arguments are passed unchanged to the foreign function. Fixnum and flonum arguments are copied into a temporary location and a pointer to the value is passed (this is because Fortran uses call by reference and it is dangerous to modify the contents of a fixnum or flonum which something else might point to). If the argument is an array object, the data field of the array object is passed to the foreign function (This is the easiest way to send large amounts of data to and receive large amounts of data from a foreign function). If a binary object is an argument, the entry field of that object is passed to the foreign function (the entry field is the address of a function, so this amounts to passing a function as an argument).

When a function in the C group is called, fixnum and flownum arguments are passed by value. For almost all other arguments, the address is merely provided to the C routine. The only exception arises when you want to invoke a C routine which expects a ‘‘structure’’ argument. Recall that a (rarely used) feature of the C language is the ability to pass structures by value. This copies the structure onto the stack. Since the Franz’s nearest equivalent to a C structure is a vector, we provide an escape clause to copy the contents of an immediate-type vector by value. If the property field of a vectori argument, is the symbol ‘‘value-structure-argument’’, then the binary data of this immediate-type vector is copied into the argument list of the C routine.

The method a foreign function uses to access the arguments provided by Lisp is dependent on the language of the foreign function. The following scripts demonstrate how how Lisp can interact with three languages: C, Pascal and Fortran. C and Pascal have pointer types and the first script shows how to use pointers to extract information from Lisp objects. There are two functions defined for each language. The first (cfoo in C, pfoo in Pascal) is given four arguments, a fixnum, a flonum-block array, a hunk of at least two fixnums and a list of at least two fixnums. To demonstrate that the values were passed, each ?foo function prints its arguments (or parts of them). The ?foo function then modifies the second element of the flonum-block array and returns a 3 to Lisp. The second function (cmemq in C, pmemq in Pascal) acts just like the Lisp *memq* function (except it won’t 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 *italic.*

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 38.60
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
#63000-"integer-function"
–> **(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
#63200-"integer-function"
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 will be much shorter since in Fortran you can't 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 38.60
–> (cfasl ’ch8auxf.o ’_ffoo_ ’ffoo "real-function" "-lF77 -lI77")
/usr/lib/lisp/nld -N -A /usr/local/lisp -T 63000 ch8auxf.o -e _ffoo_
-o /tmp/Li11066.0 -lF77 -lI77 -lc
#6307c-"real-function"

–> (array test fixnum-block 2)
array[2]
–> (store (test 0) 10)
10
–> (store (test 1) 11)
11
–> (ffoo 385 (getd ’test) 5.678)
 a= 385, b(1)=   10, b(2)=   11 c=5.6780
1.234559893608093
–> (test 0)
22
```

CHAPTER 9

Arrays and Vectors

Arrays and vectors are two means of expressing aggregate data objects in FRANZ LISP. Vectors may be thought of as sequences of data. They are intended as a vehicle for user-defined data types. This use of vectors is still experimental and subject to revision. As a simple data structure, they are similar to hunks and strings. Vectors are used to implement closures, and are useful to communicate with foreign functions. Both of these topics were discussed in Chapter 8. Later in this chapter, we describe the current implementation of vectors, and will advise the user 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 we will describe how array references are handled by the lisp system.

The structure of an array object is given in §1.3.10 and reproduced here for your convenience.

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 we will call foo_arr_obj). First the evaluator will evaluate and stack the values of *a* and *b*. Next it will stack the array object foo_arr_obj. Finally it will call 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 will also be called upon to store elements in the array. For example, *(store (foo a b) c)* will automatically expand to (foo c a b) and when the evaluator is called to evaluate this, it will evaluate the arguments *c*, *b* and *a*. Then it will stack the array object (which is stored in the function cell of foo) and call 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 which 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 above. 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 will not be marked by the garbage collector (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 will be 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 arrays in other languages: a block of homogeneous data elements which 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 we want, so we evaluate the value cell and return it 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 the exact 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 could implement would be hashed arrays. The subscript could be anything, not just a number. The access function would hash the subscript and use 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 shortcommings with 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 an *Blobit*." Thus a **vector** is an arbitrary 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 which will be common to all *blobits*. On the other hand, for each individual blobits there are particulars which are likely to change, (height, weight, 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. Once again we summarize the structure of a vector in tabular form:

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 the user should know:

†Aliasing is when two variables are share the same storage location. For example if the copying mentioned weren't 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.

9.5.1. size The user is not free to alter this. It is noted when the vector is created, and is used by the garbage collector. The garbage collector will coallesce two free vectors, which are neighbors in the heap. Internally, this is kept as the number of bytes of data. Thus, a vector created by (*vector* ’foo), has a size of 4.

9.5.2. property Currently, we expect the property to be either a symbol, or a list whose first entry is a symbol. The symbols **fclosure** and **structure-value-argument** are magic, and their effect is described in Chapter 8. If the property is a (non-null) symbol, the vector will be printed out as <symbol>[<size>]. Another case is if the property is actually a (disembodied) property-list, which contains a value for the indicator **print**. The value is taken to be a Lisp function, which the printer will invoke with two arguments: the vector and the current output port. Otherwise, the vector will be printed as vector[<size>]. We have vague (as yet unimplemented) ideas about similar mechanisms for evaluation properties. Users are cautioned against putting anything other than nil in the property entry of a vector.

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 pointer to the first data cell, i.e. 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 below). The user should never return a pointer to any other part of a vector, as this may cause the garbage collector to follow an invalid pointer.

9.6. immediate-vectors Immediate-vectors are similar to vectors. They differ, in that binary data are stored in space directly within the vector. Thus the garbage collector will preserve the vector itself (if used), and will only traverse 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 will avoid 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	vseti-byte	fixnum
	vrefi-word	vseti-word	fixnum
	vrefi-long	vseti-long	fixnum
property	vprop	vsetprop	lispval
		vputprop	
size	vsize	–	fixnum
	vsize-byte		fixnum
	vsize-word		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 the user to handle in a number of ways the errors which 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 will return to the errset which will return 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 which 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, assuming that the error is continuable, return to the function which 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 which indicates the general classification of the error. This classification may determine which function handles this error.

(2) unique id - This is a fixnum unique to this error.

(3) continuable - If this is non-nil then this error is continuable. There are some who feel that every error should be continuable and the reason that some (in fact most) errors in FRANZ LISP are not continuable is due to the laziness of the programmers.

(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 which 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 which is done 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 we return the *car* of the list to the function which called the error. Presumably the function will use 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, so we go on to step (2). Something

special happens before we call the **ER%all** error handler which does not happen in any of the other cases we will describe below. To help insure that we don't get infinitely recursive errors 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 which 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 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, and then we go 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* we go to step (4).

(4) If the symbol **ER%tpl** has a value then it is the name of an error handler which is called in a manner similar to that discussed above. If it chooses not to handle the error, we go to step (5).

(5) At this point it has been determined that the user doesn't 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 which noticed the error. Otherwise the error handler has decided not to handle the error and we go on to something else.

10.4. Default aids

There are two standard error handlers which will probably handle the needs of most users. One of these is the lisp coded function *break-err-handler* which is the default value of **ER%tpl.** Thus when all other handlers have ignored an error, *break-err-handler* will take over. It will print out the error message and go into a read-eval-print loop. The other standard error handler is *debug-err-handler*. This handler is designed to be connected to **ER%all**and is useful if your program uses *errset* and you want to look at the error before it is thrown up to the *errset*.

10.5. Autoloading

When *eval*, *apply* or *funcall* are told to call an undefined function, an **ER%undef** error is signaled. The default handler for this error is *undef-func-handler*. This function checks the property list of the undefined function for the indicator autoload. If present, the value of that indicator should be the name of the file which contains the definition of the undefined function. *Undef-func-handler* will load the file and check if it has defined the function which caused the error. If it has, the error handler will return and the computation will continue as if the error did not occur. This provides a way for the user to tell the lisp system about the location of commonly used functions. The trace package sets up an autoload property to point to /usr/lib/lisp/trace.

10.6. Interrupt processing

The UNIX operating system provides one user interrupt character which defaults to ˆC.[†] The user 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 were in an inconsistent state, the processing of the interrupt may be delayed until a safe time. When the first ˆC 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 doesn't respond to the first ˆC, another ˆC should be typed. This will cause all of the transfer tables to be cleared forcing all calls from compiled code to go through the *qlinker* function where the interrupt flag will be checked. If the lisp system still doesn't respond, a third ˆC will cause an immediate interrupt. This interrupt will not necessarily be 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 Joseph Lister Trace Package

The Joseph 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 /usr/lib/lisp/trace.l). Although not normally loaded in the Lisp system, the package will be loaded in when the first call to *trace* is made.

(trace [ls_arg1 ...]**)**

WHERE: the form of the ls_arg*i* is described below.

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 function definitions of the functions to trace are modified.

The ls_arg*i* can have one of the following forms:

foo - when foo is entered and exited, the trace information will be printed.

(foo break) - when foo is entered and exited the trace information will be printed. Also, just after the trace information for foo is printed upon entry, you will be put in a special break loop. The prompt is ‘T>’ 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 ˆD or (tracereturn) and execution will continue. Note that ˆD will work only on UNIX systems.

(foo if expression) - when foo is entered and the expression evaluates to non-nil, then the trace information will be printed for both exit and entry. If expression evaluates to nil, then no trace information will be printed.

(foo ifnot expression) - when foo is entered and the expression evaluates to nil, then the trace information will be 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 will be evaluated. Exit trace information will be printed when foo exits.

---

†*Lister, Joseph* 1st Baron Lister of Lyme Regis, 1827-1912; English surgeon: introduced antiseptic surgery.

(foo evalout expression) - when foo is entered, entry trace information will be printed. When foo exits, and before the exit trace information is printed, expression will be 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 a &. This allows you to trace functions with massive arguments or results.

The following trace options permit one to have greater control over each action which takes place when a function is traced. These options are only meant to be used by people who need special hooks into the trace package. Most people 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 will be bound to the name of the function being traced, foo in this case. The second argument will be 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 will be bound to the name of the function being traced, foo in this case. The second argument will be 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 will usually be an expression, not just the name of a function, and that expression will be specific to the evaluation of function foo. The argument list to be applied will be available as T-arglist.

(foo printargs prfunc) - this tells *trace* to used prfunc to print the arguments to be applied to the function foo. prfunc should be a lambda of one argument. You might want to use this option if you wanted a print function which could handle circular lists. This option will work 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 will work 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 will be printed with the level printer. Next it will enter a trace break loop, allowing you to evaluate any lisp expressions. When you exit the trace break loop, foo will be applied to its arguments and the resulting value will be printed, again using the level printer. Bar is also traced, and each time bar is entered, an entering bar message will be printed and then the value of xyzzy will be printed. Next bar will be applied to its arguments and the result will be printed. If you tell *trace* to trace a function which is already traced, it will first *untrace* 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, will be in effect.

If the symbol $tracemute is given a non nil value, printing of the function name and arguments on entry and exit will be 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 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 which has been traced).

When 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* will do 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 will run 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. returns a list of functions untraced.

(untrace [s_arg1 ...]**)**

RETURNS: a list of the functions which 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 which when brought into the lisp system using *fasl* will have the same effect as bringing in the corresponding lisp coded source module with *load* with one important exception, functions will be 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 will also be stored in the object module produced by Liszt and will be evaluated at fasl time.

As is almost universally true of Lisp compilers, the main pass of Liszt is written in Lisp. A subsequent pass is the assembler, for which we use the standard UNIX assembler.

12.2. Running the compiler

The compiler is normally run in this manner:
% **liszt foo**
will compile 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 will be placed in the file foo.o if no fatal errors were detected. All messages which Liszt generates go to the standard output. Normally each function name is printed before it is compiled (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 (i.e 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. The user may define his 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 will invoke *eval* on each form*i* as it examines it. If *load* is in the list then the compile will recursively call itself to compile each form*i* as it examines it. Note that if *compile* and *load* are in the time list, then the compiler will both evaluate and compile 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), 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 needn't worry about declaring lexprs either. It is important to declare nlambdas or define them before calling them. Another attribute you can declare for a function is localf which makes the function 'local'. A local function's name is known only to the functions defined within the file itself. The advantage of a local function is that is can be entered and exited very quickly and it can have the same name as a function in another file and there will be no name conflict.

Variables may be declared special or unspecial. When a special variable is lambda bound (either in a lambda, prog or do expression), its old value is stored away on a stack for the duration of the lambda, prog or do expression. This takes time and is often not necessary. Therefore the default classification for variables is unspecial. Space for unspecial variables is dynamically allocated on a stack. An unspecial variable can only be accessed from within the function where it is created by its presence in a lambda, prog or do expression variable list. It is possible to declare that all variables are special as will be shown below.

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 above, declare is much like *(eval-when (compile) ...)*. Thus if the compiler sees *(declare (foo bar))* and foo is defined, then it will evaluate *(foo bar)*. If foo is not defined then an undefined declare attribute warning will be 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 doesn't 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 will evaluate the definition, thus defining the macro withing the running Lisp compiler. Furthermore, if the variable *macros* is set to a non nil value, then the macro definition will also be translated to machine language and thus will be 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 will macro expand it. It will not macro expand arguments to a nlambda unless the characteristics of the nlambda is 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 which 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 won't have to worry about all that detail as files which work interpreted will work compiled. Following is a list of steps you should follow to insure that a file will compile 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 which has not been defined in the current file, which does not currently have a function binding, and whose type has not been declared then it will assume that the function needs its arguments evaluated (i.e. it is a lambda or lexpr) and will generate 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 which are used for communicating values between functions. These variables must be declared special at the beginning of a file. In most cases there won't be many special declarations but if you fail to declare a variable special that should be, the

compiled code could fail in mysterious ways. Let's look at a common problem, 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)))

We can see that if we load in these two definitions then (aaa 3 4) is the same as (add 3 4) and will give us 7. Suppose we compile the file containing these definitions. When Liszt compiles aaa, it will assume that both glob and loc are local variables and will allocate space on the temporary stack for their values when aaa is called. Thus the values of the local variables glob and loc will 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 find 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 will automatically declare glob to be special and it will print a warning to that effect. Thus subsequent uses of glob will 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 will be lambda bound to 3 bbb will be called and will return 7. However (aaa 3 4) will fail since when bbb is called, glob will be 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 will work however. The macro *listify* can be used to put all of some of a lexprs arguments in a list which then can be passed to other functions.

12.5. Compiler options

The compiler recognizes a number of options which are described below. The options are typed anywhere on the command line preceded by a minus 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. Before scanning the command line for options, liszt looks for in the environment for the variable LISZT, and if found scans its value as if it was a string of options. The meaning 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
will evaluate the above s-expression. Note that the shell requires that the arguments be surrounded by single quotes.

i Compile this program in interlisp compatibility mode. This is not implemented yet.

m Compile this program in Maclisp mode. The reader syntax will be changed to the Maclisp syntax and a file of macro definitions will be loaded in (usually named /usr/lib/lisp/machacks). This switch brings us sufficiently close to Maclisp to allow us to compile Macsyma, a large Maclisp program. However Maclisp is a moving target and we can't guarantee that this switch will allow you to compile any given program.

o Select a different object or assembler language file name. For example
% liszt foo -o xxx.o
will compile foo and into xxx.o instead of the default foo.o, and
% liszt bar -S -o xxx.s
will compile to assembler language into xxx.s instead of bar.s.

p place profiling code at the beginning of each non-local function. If the lisp system is also created with profiling in it, this allows function calling frequency to be determined (see *prof(1)*)

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 will cause a lisp system to be invoked and the object file *fasl*ed in. This is known as ‘autorun’ and is described below.

S Create an assembler language file only.
% liszt -S foo
will create the file assembler language file foo.s and will not attempt to assemble it. If this option is not specified, the assembler language file will be put in the temporary disk area under a automatically generated name based on the lisp compiler's process id. Then if there are no compilation errors, the assembler will be 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 an 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 which 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 will contain a small piece of bootstrap code at the beginning, and the object file will be made executable. Now, when the name of the object file is given to the UNIX command interpreter (shell) to run, the bootstrap code at the beginning of the object file will cause a lisp system to be started and the first action the lisp system will take is to *fasl* in the object file which 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 which 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 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 would be wasted *fasl*ing it in each time it was 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.

---

we want to replace the unix date program with one written in lisp.

```
% cat lispdate.l
(defun mydate nil
    (patom "The date is ")
    (patom (status ctime))
    (terpr)
    (exit 0))
(setq user-top-level 'mydate)

% liszt -r lispdate
Compilation begins with Lisp Compiler 5.2
source: lispdate.l, result: lispdate.o
mydate
%Note: lispdate.l: Compilation complete
%Note: lispdate.l:  Time: Real: 0:3, CPU: 0:0.28, GC: 0:0.00 for 0 gcs
%Note: lispdate.l: Assembly begins
%Note: lispdate.l: Assembly completed successfully
3.0u 2.0s 0:17 29%

 We change the name to remove the ".o", (this isn't necessary)
% mv lispdate.o lispdate

 Now we test it out
% lispdate
The date is Sat Aug  1 16:58:33 1981
%
```

12.7. pure literals

Normally the quoted lisp objects (literals) which 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 will never be true, since the *car* of *(a b)* should always be *a*. However if you run this function twice, it will print '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 above 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 put in space which needn't be looked at by the garabage 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 will act 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 which 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 will initially point to the internal function *qlinker*. It may some time in the future point to the function *foo* if certain conditions are satisfied (more on this below).

[2] function name − This is a pointer to the symbol *foo*. This will be used by *qlinker.*

When a call is made to the function *foo* the call will actually be made to the address in the transfer table entry and will end up in the *qlinker* function. *Qlinker* will determine that *foo* was 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* will modify the function address part of the transfer table to point directly to the function *foo*. Finally *qlinker* will call *foo* directly . The next time a call is made to *foo* the call will go directly to *foo* and not through *qlinker*. This will result 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 above has already been done. The solution to these problems is to use *(sstatus translink value)*. If value is

nil All transfer tables will be cleared, i.e. all function addresses will be set to point to *qlinker*. This means that the next time a function is called *qlinker* will be called and will look at the current definition. Also, no fast links will be set up since *(status translink)* will be nil. The end result is that *showstack* and *baktrace* will work and the function definition at the time of call will always be 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* will make new fast links if the situation arises (which isn't likely unless you *fasl* in another file).

t This or any other value not previously mentioned will just make *(status translink)* be non nil, and as a result fast links will be made by *qlinker* if the called function is compiled.

12.9. Fixnum functions

The compiler will generate inline arithmetic code for fixnum only functions. Such functions include +, −, *, /, \, 1+ and 1−. The code generated will be much faster than using *add*, *difference*, etc. However it will only work 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 which called it. The address field of the *calls* contains the address of the transfer table entry.

CHAPTER 13

The CMU User Toplevel and the File Package

This documentation was written by Don Cohen, and the functions described below were imported from PDP-10 CMULisp.

Non CMU users note: this is not the default top level for your Lisp system. In order to start up this top level, you should type *(load 'cmuenv)*.

13.1. User Command Input Top Level

The top-level is the function that reads what you type, evaluates it and prints the result. The *newlisp* top-level was inspired by the CMULisp top-level (which was inspired by interlisp) but is much simpler. The top-level is a function (of zero arguments) that can be called by your program. If you prefer another top-level, just redefine the top-level function and type "(reset)" to start running it. The current top-level simply calls the functions tlread, tleval and tlprint to read, evaluate and print. These are supposed to be replaceable by the user. The only one that would make sense to replace is tlprint, which currently uses a function that refuses to go below a certain level and prints "...]" when it finds itself printing a circular list. One might want to prettyprint the results instead. The current top-level numbers the lines that you type to it, and remembers the last n "events" (where n can be set but is defaulted to 25). One can refer to these events in the following "top-level commands":

TOPLEVEL COMMAND SUMMARY

??	prints events - both the input and the result. If you just type "??" you will see all of the recorded events. "?? 3" will show only event 3, and "?? 3 6" will show events 3 through 6.
redo	pretends that you typed the same thing that was typed before. If you type "redo 3" event number 3 is redone. "redo -3" redoes the thing 3 events ago. "redo" is the same as "redo -1".
ed	calls the editor and then does whatever the editor returns. Thus if you want to do event 5 again except for some small change, you can type "ed 5", make the change and leave the editor. "ed -3" and "ed" are analogous to redo.

Finally, you can get the value of event 7 with the function (valueof 7). The other interesting feature of the top-level is that it makes outermost parentheses superfluous for the most part. This works the same way as in CMULisp, so you can use the help for an explanation. If you're not sure and don't want to risk it you can always just include the parentheses.

(top-level)

SIDE EFFECT: *top-level* is the LISP top level function. As well as being the top level function with which the user interacts, it can be called recursively by the user or any function. Thus, the top level can be invoked from inside the editor, break package, or a user function to make its commands available to the user.

NOTE: The CMU FRANZ LISP top-level uses *lineread* rather than *read.* The difference will not usually be noticeable. The principal thing to be careful about is that input to the function or system being called cannot appear on the same line as the top-level call. For example, typing *(editf foo)fP on one line will edit foo and evaluate P, not edit foo and execute the p command in the editor. top-level specially recognizes the following commands:*

(valueof *'g_eventspec***)**

RETURNS: the value(s) of the event(s) specified by g_eventspec. If a single event is specified, its value will be returned. If more than one event is specified, or an event has more than one subevent (as for *redo,* etc), a list of vlaues will be returned.

13.2. The File Package

Users typically define functions in lisp and then want to save them for the next session. If you do *(changes)*, a list of the functions that are newly defined or changed will be printed. When you type *(dskouts)*, the functions associated with files will be saved in the new versions of those files. In order to associate functions with files you can either add them to the *filefns* list of an existing file or create a new file to hold them. This is done with the *file* function. If you type *(file new)* the system will create a variable called *newfns.* You may add the names of the functions to go into that file to *newfns.* After you do *(changes)*, the functions which are in no other file are stored in the value of the atom *changes.* To put these all in the new file, *(setq newfns (append newfns changes))*. Now if you do *(changes)*, all of the changed functions should be associated with files. In order to save the changes on the files, do *(dskouts)*. All of the changed files (such as NEW) will be written. To recover the new functions the next time you run FRANZ LISP, do *(dskin new)*.

```
Script started on Sat Mar 14 11:50:32 1981
$ newlisp
Welcome to newlisp...
1.(defun square (x) (* x x))        ; define a new function
square
2.(changes)                                 ; See, this function is associated
                                            ; with no file.
<no-file>     (square)nil
3.(file 'new)                               ; So let's declare file NEW.
new
4.newfns                            ; It doesn't have anything on it yet.
nil
5.(setq newfns '(square))           ; Add the function associated
(square)                            ; with no file to file NEW.
6.(changes)                                 ; CHANGES magically notices this fact.

new          (square)nil
7.(dskouts)                         ; We write the file.
creating new
(new)
8.(dskin new)                               ; We read it in!
(new)
14.Bye
$
script done on Sat Mar 14 11:51:48 1981
```

(changes s_flag**)**

RETURNS: Changes computes a list containing an entry for each file which defines atoms that have been marked changed. The entry contains the file name and the changed atoms defined therein. There is also a special entry for changes to atoms which are not defined in any known file. The global variable *filelst* contains the list of "known" files. If no flag is passed this result is printed in human readable form and the value returned is t if there were any changes and nil if not. Otherwise nothing is printed and the computer list is returned. The global variable *changes* contains the atoms which are marked changed but not yet associated with any file. The *changes* function attempts to associate these names with files, and any that are not found are considered to belong to no file. The *changes* property is the means by which changed functions are associated with files. When a file is read in or written out its *changes* property is removed.

(dc s_word s_id [g_descriptor1 ...] <text> <esc>**)**

RETURNS: *dc* defines comments. It is exceptional in that its behavior is very context dependent. When *dc* is executed from *dskin* it simply records the fact that the comment exists. It is expected that in interactive mode comments will be found via *getdef* - this allows large comments which do not take up space in your core image. When *dc* is executed from the terminal it expects you to type a comment. *dskout* will write out the comments that you define and also copy the comments on the old version of the file, so that the new version will keep the old comments even though they were never actually brought into core. The optional id is a mechanism for distinguishing among several comments associated with the same word. It defaults to nil. However if you define two comments with the same id, the second is considered to be a replacement for the first. The behavior of *dc* is determined by the value of the global variable *def-comment*. *def-comment* contains the name of a function that is run. Its arguments are the word, id and attribute list. *def-comment* is initially *dc-define*. Other functions rebind it to *dc-help*, *dc-userhelp*, and the value of *dskin-comment.* The comment property of an atom is a list of entries, each representing one comment. Atomic entries are assumed to be identifiers of comments on a file but not in core. In-core comments are represented by a list of the id, the attribute list and the comment text. The comment text is an uninterned atom. Comments may be deleted or reordered by editing the comment property.

(dskin l_filenames**)**

SIDE EFFECT: READ-EVAL-PRINTs the contents of the given files. This is the function to use to read files created by *dskout. dskin* also declares the files that it reads (if a *file-fns* list is defined and the file is otherwise declarable by *file*), so that changes to it can be recorded.

(dskout s_file1 ...**)**

SIDE EFFECT: For each file specified, *dskout* assumes the list named filenameFNS (i.e., the file name, excluding extension, concatenated with *fns*) contains a list of function names, etc., to be loaded Any previous version of the file will be renamed to have extension ".back".

(dskouts s_file1 ...**)**

SIDE EFFECT: applies *dskout* to and prints the name of each s_filei (with no additional arguments, assuming filenameFNS to be a list to be loaded) for which s_file*i* is either not in *filelst* (meaning it is a new file not previously declared by *file* or given as an argument to *dskin*, *dskouts*, or *dskouts*) or is in *filelst* and has some recorded changes to definitions of atoms in filenameFNS, as recorded by *mark!changed* and noted by changes. If *file*i is not specified, *filelst* will be used. This is the most common way of using dskouts. Typing *(dskouts)* will save every file reported by *(changes)* to have changed definitions.

(dv s_atom g_value**)**

EQUIVALENT TO: *(setq atom 'value)*. *dv* calls *mark!changed.*

(file 's_file**)**

SIDE EFFECT: declares its argument to be a file to be used for reporting and saving changes to functions by adding the file name to a list of files, *filelst. file* is called for each file argument of *dskin, dskout,* and *dskouts.*

(file-fns 's_file**)**

RETURNS: the name of the fileFNS list for its file argument s_file.

(getdef 's_file ['s_i1 ...]**)**

SIDE EFFECT: selectively executes definitions for atoms s_i1 ... from the specified file. Any of the words to be defined which end with "@" will be treated as patterns in which the @ matchs any suffix (just like the editor). *getdef* is driven by *getdeftable* (and thus may be programmed). It looks for lines in the file that start with a word in the table. The first character must be a "(" or "[" followed by the word, followed by a space, return or something else that will not be considered as part of the identifier by *read*, e.g., "(" is unacceptable. When one is found the next word is read. If it matches one of the identifiers in the call to *getdef* then the table entry is executed. The table entry is a function of the expression starting in this line. Output from *dskout* is in acceptable format for *getdef. getdef*

RETURNS: a list of the words which match the ones it looked for, for which it found (but, depending on the table, perhaps did not execute) in the file.

NOTE: *getdeftable* is the table that drives *getdef.* It is in the form of an association list. Each element is a dotted pair consisting of the name of a function for which *getdef* searches and a function of one argument to be executed when it is found.

(mark!changed 's_f**)**

SIDE EFFECT: records the fact that the definition of s_f has been changed. It is automatically called by *def, defun, de, df, defprop, dm, dv,* and the editor when a definition is altered.

CHAPTER 14

The LISP Stepper

14.1. 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).

```
(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

14.2. Advanced Features

14.2.1. 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 preceded 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 only calls to eval use: *(step e)*

14.2.2. Stepping With Breakpoints.

For the moment, step is turned off inside of error breaks, but not by the break function. Upon exiting the error, step is reenabled. However, executing *(step nil)* inside a error loop will turn off stepping globally, i.e. within the error loop, and after return has be made from the loop.

14.3. Overhead of Stepping.

If stepping mode has been turned off by *(step nil)*, the execution overhead of having the stepping packing in your LISP is identically nil. 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 of the car of the current form into the list *(foo1 foo2 ...)* is required at each call to eval.

14.4. Evalhook and Funcallhook

There are hooks in the FRANZ LISP interpreter to permit a user written function to gain control of the evaluation process. These hooks are used by the Step package just described. There are two hooks and they have been strategically placed in the two key functions in the interpreter: *eval* (which all interpreted code goes through) and *funcall* (which all compiled code goes through if *(sstatus translink nil)* has been done). 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 enter.

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 given 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 fhooks 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 ...
```

CHAPTER 15

The FIXIT Debugger

15.1. Introduction FIXIT is a debugging environment for FRANZ LISP users doing program development. This documentation and FIXIT were written 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 the program 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 extremely rapidly, and with a minimum of frustration.

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 compiled code to compiled code linkage tables are unlinked, i.e do *(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 /usr/lisp/doc/fixit.help.  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 Rich 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
```

15.2. 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.

15.3. 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.

15.4. 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. The Editors

It is quite possible to use VI, Emacs or other standard editors to edit your lisp programs, and many people do just that. However there is a lisp structure editor which is particularly good for the editing of lisp programs, and operates in a rather different fashion, namely within a lisp environment. application. It is handy to know how to use it for fixing problems without exiting from the lisp system (e.g. from the debugger so you can continue to execute rather than having to start over.) The editor is not quite like the top-level and debugger, in that it expects you to type editor commands to it. It will not evaluate whatever you happen to type. (There is an editor command to evaluate things, though.)

The editor is available (assuming your system is set up correctly with a lisp library) by typing (load 'cmufncs) and (load 'cmuedit).

The most frequent use of the editor is to change function definitions by starting the editor with one of the commands described in section 16.14. (see *editf*), values (*editv*), properties (*editp*), and expressions (*edite*). The beginner is advised to start with the following (very basic) commands: *ok, undo, p, #*, under which are explained two different basic commands which start with numbers, and f.

This documentation, and the editor, were imported from PDP-10 CMULisp by Don Cohen. PDP-10 CMULisp is based on UCILisp, and the editor itself was derived from an early version of Interlisp. Lars Ericson, the author of this section, has provided this very concise summary. Tutorial examples and implementation details may be found in the Interlisp Reference Manual, where a similar editor is described.

16.2. 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 . If a p command would cause the editor to type ... before typing the current expression, (the current expression is a tail of the next higher expression) then has no effect; else, up makes the old current expression the first element in the new current expression.

!0 . Goes back to the next higher left parenthesis.

ˆ . Makes the top level expression be the current expression.

nx . Makes the current expression be the next expression.

(nx n) equivalent to n 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.

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 $) - generalized nth command. nth 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.

:: . (pattern :: . $) e.g., (cond :: return). finds a cond that contains a return, at any depth.

(below com x) . The below command is useful for locating a substructure by specifying something it contains. (below cond) will cause the cond clause containing the current expression to become the new current expression. Suppose you are editing a list of lists, and want to find a sublist that contains a foo (at any depth). Then simply executes f foo (below).

(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.

16.3. Pattern Matching Commands

Many editor commands that search take patterns. A pattern *pat* matches with x if:

PATTERN SPECIFICATION SUMMARY

- *pat* is *eq* to x.

- *pat* is &.

- *pat* is a number and equal to x.

- if (car *pat*) is the atom *any*, (cdr *pat*) is a list of patterns, and *pat* matches x if and only if one of the patterns on (cdr *pat*) matches x.

- if *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.) - otherwise if x is a list, *pat* matches x if (car *pat*) matches (car x), and (cdr *pat*) matches (cdr x).

- when searching, the pattern matching routine is called only to match with elements in the structure, unless the pattern begins with :::, in which case cdr of the pattern is matched against tails in the structure. (in this case, the tail does not have to be a proper tail, e.g. (::: a --) will match with the element (a b c) as well as with cdr of (x a b c), since (a b c) is a tail of (a b c).)

16.3.1. Commands That Search

SEARCH COMMAND SUMMARY

f pattern . f informs the editor that the next command is to be interpreted as a pattern. If no pattern is given on the same line as the f then the last pattern is used. f pattern means find the next instance of pattern.

(f pattern n). Finds the next instance of pattern.

(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 'stay here'.

(f pattern n) n>0. Finds the nth place that pattern matches. If the current expression is (foo1 foo2 foo3), (f f00@ 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.

(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.

(fs pattern1 ... patternn) . equivalent to f pattern1 followed by f pattern2 ... followed by f pattern n, so that if f pattern m fails, edit chain is left at place pattern m-1 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 . backwards find. If the current expression is *(prog nil (setq x (setq y (list z))) (cond ((setq w --) --)) --)* f list followed by bf setq will leave the current expression as (setq y (list z)), as will f cond followed by bf setq

(bf pattern t). backwards find. Search always includes current expression, i.e., starts at end of current expression and works backward, then ascends and backs up, etc.

16.3.1.1. 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 two exceptions. first, 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 3rd 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.

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 ˆ 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.

$. 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.

LOCATION COMMAND SUMMARY

(lc . $) . Provides a way of explicitly invoking the location operation. (lc cond 2 3) will perform search.

(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.

16.3.2. The Edit Chain The edit-chain is a list of which the first element is the the one you are now editing ("current expression"), the next element is what would become the current expression if you were to do 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. for example:

__ . Similar to _ 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 ˆ, _, __, !nx, all commands that involve a search, e.g., f, lc, ::, below, et al and and 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.

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 current expression as though printlev were given a depth of n. (p cond 3) will work.

? . 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. Structure Modification Commands

All structure modification commands are undoable. See *undo*.

STRUCTURE MODIFICATION COMMAND SUMMARY

 # [editor commands] (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.

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.

16.6. Extraction and Embedding Commands

EXTRACTION AND EMBEDDING COMMAND SUMMARY

(xtr . $) . replaces the original current expression with the expression that is current after performing (lcl . $).

(mbd x) . 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. (mbd x) : x atomic, same as (mbd (x *)).

(extract $1 from $2) . extract 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.) 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) . embed replaces the current expression with a new expression which contains it as a subexpression. ($ is the segment between embed and in.) example: (embed print in setq x), (embed 3 2 in *return*), (embed cond 3 1 in (or * (null x))).

16.7. Move and Copy Commands

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.

16.8. Parentheses Moving Commands

The commands presented in this section permit modification of the list structure itself, as opposed to modifying components thereof. their effect can be described as inserting or removing a single left or right parenthesis, or pair of left and right parentheses.

PARENTHESES MOVING COMMAND SUMMARY

(bi n m) . both in. 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) . both out. 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) . left in. 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) . left out. 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) . right in. move 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) . right out. move 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)).

(r x y) replaces all instances of x by y in the current expression, e.g., (r caadr cadar). x can be the s-expression (or atom) to be substituted for, or can be a pattern which specifies that s-expression (or atom).

(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.

16.8.1. Using to and thru

to, thru, extract, embed, delete, replace, and move can be made to operate on several contiguous elements, i.e., a segment of a list, by using the to or thru command in their respective location specifications. thru and to are intended to be used in conjunction with 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)*.

16.9. Undoing Commands

each command that causes structure modification automatically adds an entry to the front of undolst containing the information required to restore all pointers that were changed by the command. The undo command undoes the last, i.e., most recent such command.

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 test together with !undo provide a 'tentative' mode for editing, i.e., the user can perform a number of changes, and then undo all of them with a single !undo command.

undolst [value]. each editor command that causes structure modification automatically adds an entry to the front of undolst containing the information required to restore all pointers that were changed by the command.

?? prints the entries on undolst. The entries are listed most recent entry first.

16.10. Commands that Evaluate

EVALUATION COMMAND SUMMARY

e . only when typed in, (i.e., (insert d before e) will treat e as a pattern) causes the editor to call the lisp interpreter giving it the next input as argument.

(e x) evaluates x, and prints the result. (e x t) 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.

(coms x1 ... xn) . each xi is evaluated and its value executed as a command. The i command is not very convenient for computing an entire edit command for execution, since it computes the command name and its arguments separately. also, the i command cannot be used to compute an atomic command. The coms and comsq commands provide more general ways of computing commands. (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. (nil as a command is a nop.)

(comsq com1 ... comn) . executes com1 ... comn. comsq is 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 (quote comsq) x)) where x computed the list of commands, e.g., (coms (cons (quote comsq) (get foo (quote commands))))

16.11. Commands that Test

TESTING COMMAND SUMMARY

(if x) generates an error unless the value of (eval x) is non-nil, i.e., if (eval x) causes an error or (eval x)=nil, if will cause an error. (if x coms1 coms2) if (eval x) is non-nil, execute coms1; if (eval x) causes an error or is equal to nil, execute coms2. (if x coms1) if (eval x) is non-nil, execute coms1; otherwise generate an error.

(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.

16.12. 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.

(m c . coms) for c an atom, m defines c as an atomic command. (if a macro is redefined, its new definition replaces its old.) executing c is then the same as executing the list of commands coms. 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) bind 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 [value]. this variable contains the users editing macros . if you want to save your macros then you should save usermacros. you should probably also save editcomsl.

editcomsl [value]. editcomsl is the list of "list commands" recognized by the editor. (these are the ones of the form (command arg1 arg2 ...).)

16.13. Miscellaneous Editor Commands

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. 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. stop provides a way of distinguishing between a successful and unsuccessful (from the user's standpoint) editing session.

tl . 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) . 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 after performing (lc . $). (s foo) will set foo to the current expression, (s foo -1 1) will set foo to the first element in the last element of the current expression.

16.14. Editor Functions

(editf s_x1 ...**)**

SIDE EFFECT: edits a function. s_x1 is the name of the function, any additional arguments are an optional list of commands.

RETURNS: s_x1.

NOTE: if s_x1 is not an editable function, editf generates an fn not editable error.

(edite l_expr l_coms s_atm)**)**
edits an expression. its value is the last element of (editl (list l_expr) l_coms s_atm nil nil).

(editracefn s_com**)**
is available to help the user debug complex edit macros, or subroutine calls to the editor. editracefn is 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. editracefn is initially equal to nil, and undefined.

(editv s_var [g_com1 ...]**)**

SIDE EFFECT: similar to editf, for editing values. editv sets the variable to the value returned.

RETURNS: the name of the variable whose value was edited.

(editp s_x**)**

SIDE EFFECT: similar to editf for editing property lists. used if x is nil.

RETURNS: the atom whose property list was edited.

(editl coms atm marklst mess**)**

SIDE EFFECT: editl is the editor. its first argument is the edit chain, and its value is an edit chain, namely the value of l at the time editl is exited. (l is a special variable, and so can be examined or set by edit commands. ^ is equivalent to (e (setq l(last l)) t).) coms is an optional list of commands. for interactive editing, coms is nil. in this case, editl types edit and then waits for input from the teletype. (if mess is not nil editl types it instead of edit. for example, the tty: command is essentially (setq l (editl l nil nil nil (quote tty:))).) exit occurs only via an ok, stop, or save command. If coms is not nil, no message is typed, and each member of coms is treated as a command and executed. If an error occurs in the execution of one of the commands, no error message is printed , the rest of the commands are ignored, and editl exits with an error, i.e., the effect is the same as though a stop command had been executed. If all commands execute successfully, editl returns the current value of l. marklst is the list of marks. on calls from editf, atm is the name of the function being edited; on calls from editv, the name of the variable, and calls from editp, the atom of which some property of its property list is being edited. The property list of atm is used by the save command for saving the state of the edit. save will not save anything if atm=nil i.e., when editing arbitrary expressions via edite or editl directly.

(editfns s_x [g_coms1 ...]**)**
fsubr function, used to perform the same editing operations on several functions. editfns maps down the list of functions, prints the name of each function, and calls the editor (via editf) on that function.

EXAMPLE: editfns foofns (r fie fum)) will change every fie to fum in each of the functions on 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. The value of editfns is nil.

(edit4e pat y**)**

SIDE EFFECT: is the pattern match routine.

RETURNS: t if pat matches y. see edit-match for definition of 'match'.

NOTE: before each search operation in the editor begins, the entire pattern is scanned for atoms or strings that end in at-signs. These are replaced by patterns of the form (cons (quote /@) (explodec atom)). from the standpoint of edit4e, pattern type 5, atoms or strings ending in at-signs, is really "if car[pat] is the atom @ (at-sign), pat will match with any literal atom or string whose initial character codes (up to the @) are the same as those in cdr[pat]." if the user wishes to call edit4e directly, he must therefore convert any patterns which contain atoms or strings ending in at-signs to the form recognized by edit4e. this can be done via the function editfpat.

(editfpat pat flg**)**
makes a copy of pat with all patterns of type 5 (see edit-match) converted to the form expected by edit4e. flg should be passed as nil (flg=t is for internal use by the editor).

(editfindp x pat flg**)**

NOTE: Allows a program to use the edit find command as a pure predicate from outside the editor. x is an expression, pat a pattern. The value of editfindp is t if the command f pat would succeed, nil otherwise. editfindp calls editfpat to convert pat to the form expected by edit4e, unless flg=t. if the program is applying editfindp to several different expressions using the same pattern, it will be more efficient to call editfpat once, and then call editfindp with the converted pattern and flg=t.

(## g_com1 ...**)**

RETURNS: what the current expression would be after executing the edit commands com1 ... starting from the present edit chain. generates 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.

CHAPTER 17

Hash Tables

17.1. Overview

A hash table is an object that can efficiently map a given object to another. Each hash table is a collection of entries, each of which associates a unique *key* with a *value*. There are elemental 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.

Adding a key to a hash table modifies the hash table, and so it is a descructive 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. If a key is "eq" to another object, then a match is assumed. Likewise with "equal".

17.2. Functions

(makeht 'x_size ['s_test]**)**

RETURNS: A hash table of x_size hash buckets. If present, s_test is used as the test to compare keys in the hash table, the default being **eq**. *Equal* might be used to create a hash table where the keys are to be lists (or any lisp object).

NOTE: At this time, hash tables are implemented on top of vectors.

(hash-table-p 'H_arg**)**

RETURNS: t if H_arg is a hash table.

NOTE: Since hash tables are really vectors, the lisp type of a hash table is a vector, so that given a vector, this function will return t.

(gethash 'g_key 'H_htable ['g_default]**)**

RETURNS: the value associated the key g_key in hash table H_htable. If there is not an entry given by the key and g_default is specified, then g_default is returned, otherwise, a symbol that is unbound is returned. This is so that **nil** can be a associated with a key.

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

(remhash 'g_key 'H_htable**)**

RETURNS: t if there was an entry for g_key in the hash table H_htable, nil otherwise. In the case of a match, the entry and associated object are removed from the hash table.

(maphash 'u_func 'H_htable**)**

RETURNS: nil.

NOTE: The function u_func is applied to every element in the hash table H_htable. The function is called with two arguments: the key and value of an element. The mapped function should not add or delete object from the table because the results would be unpredicable.

(clrhash 'H_htable**)**

RETURNS: the hash table cleared of all entries.

(hash-table-count 'H_htable**)**

RETURNS: the number of entries in H_htable. Given a new hash table with no entries, this function returns zero.

```
; make a vanilla hash table using "eq" to compare items...
–> (setq black-box (makeht 20))
hash-table[26]
–> (hash-table-p black-box)
t
–> (hash-table-count black-box)
0
–> (setf (gethash 'key black-box) '(the value associated with the key))
key
–> (gethash 'key black-box)
(the value associated with the key)
–> (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 " val N)) black-box)
key composer value franz
key key value (the value associated with the key)
nil
–> (clrhash black-box)
hash-table[26]
–> (hash-table-count black-box)
0
–> (maphash '(lambda (key val) (msg "key " key " value " val N)) black-box)
nil

; here is an example using "equal" as the comparator
–> (setq ht (makeht 10 'equal))
hash-table[16]
–> (setf (gethash '(this is a key) ht) '(and this is the value))
(this is a key)
–> (gethash '(this is a key) ht)
(and this is the value)
; the reader makes a new list each time you type it...
–> (setq x '(this is a key))
(this is a key)
–> (setq y '(this is a key))
(this is a key)
; so these two lists are really different lists that compare "equal"
; not "eq"
–> (eq x y)
nil
; but 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)
```