JONL

The FRANZ LISP Manual

by

John K. Foderaro

A document in
four movements

JOHL

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 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 maintain the software to use it. John Foderaro wrote the compiler, added a few functions, and wrote this modest manual.*

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 Algebraic Manipulation, Artificial Intelligence, and programming languages at the University of California at Berkeley. Its roots are in the PDP-11 Lisp system which originally came from Harvard. As it grew it adopted features of Maclisp and Lisp Machine Lisp which enables our work to be shared with colleagues at the Laboratory for Computer Science at M.I.T. It is written almost entirely in the programming language C. A small part is written in the assembler language for the current host machine, a VAX 11/780, and part is written in Lisp. Because FRANZ LISP is written in C, it is 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.

1.2. 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 only to those maintaining FRANZ LISP at their computer site. 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 33b, October 1980). 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 and compiler. 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 section 1.4 -- it is very important that these conventions are understood.

1.3. Data Types FRANZ LISP has eleven 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.

1.3.0. lispval This is the name we use to describe any lisp object. The function *type* will never return 'lispval'.

1.3.1. 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 §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. This is also used when one tests for equality of symbols using the function *equal*.

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	list or nil
function binding	getd	putd def	array, binary, list or nil
print name	get_pname		string
hash link			

1.3.2. 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

1.3.3. 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 (either "*subroutine*", "*function*", "*integer-function*", or "*real-function*") then this function is a foreign subroutine or function (see §8.4 for more details on this). Although the type of the *entry* field of a binary type object is either string or fixnum, the object pointed to is actually a sequence of machine instructions.
Objects of type binary are created by *mfunction*.

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

1.3.4. 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.

1.3.5. flonum A flonum is a double precision real number in the range $\pm 2.9\times10^{-37}$ to $\pm 1.7\times10^{38}$. There are approximately sixteen decimal digits of precision.

1.3.6. bignum A bignum is an integer of potentially unbounded size. When integer arithmetic exceeds the limits 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. If the user chooses to rewrite the bignum package he should take this into account.

†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.

The functions used for the extraction and modification of parts of bignums may change from what is shown in the table sometime in the future.

Subpart name	Get value	Set value	Type
i	car	rplaca	unboxed integer
CDR	cdr	rplacd	bignum or the symbol nil

1.3.7. 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 string object is surrounded by double quotes.

1.3.8. 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*.

1.3.9. array Arrays are rather complicated types and are fully described in §9. An array consists of a block of contiguous data, a function to reference that data and auxiliary fields for use by the referencing function. Since an array's referencing 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

1.3.10. 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.

1.3.11. 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.

1.4. 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 and an underscore. The letter 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
a	array
v	value
p	port (or nil)
h	hunk

In the first line of a function description those arguments preceded by a quote mark are evaluated (usually before the function is called). The quoting convention is used 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.

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

CHAPTER 2

Data Structure Access

The following functions allow one to create and manipulate the various types of lisp data structures. Refer to §1.3 for details of the data structures known to FRANZ LISP.

(*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: The **array* function creates a Maclisp compatible array. Arrays are fully described in §9. 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 §8.4).

(aexplode 's_arg)

RETURNS: a list of single character symbols which *print* would use to print out g_arg, that is the list returned will contain quoting characters if *print* would have used them to print s_arg.

NOTE: this is restricted to symbols and is mainly for use by explode.

(aexplodec 's_arg)

RETURNS: a list of symbols whose pnames are the characters in s_arg's pname.

(aexploden 's_arg)

RETURNS: a list of fixnums which represent the characters of s_arg's pname.

```
-> (setq x |quote this \| ok?|)
|quote this \| ok?|
-> (aexplode 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

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

(alphalessp 's_arg1 's_arg2)

RETURNS: t iff the print name of s_arg1 is alphabetically less than the print name of s_arg2.

(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 description of nconc for a cheaper way of doing the append.

(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)
; better is append1
-> (append1 '(a b c d) 'e)
(a b c d e)
->
```

(array s_name s_type x_dim1 ... x_dim*i*)

NOTE: this is the same as *array except the arguments are not evaluated.

(arraycall s_type 'as_array 'x_ind1 ...)

RETURNS: the element selected by the indicies 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.

(arrayp 'g_arg)

RETURNS: t iff g_arg is of type array.

(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* used the data, length and delta fields of a_name to determine which object to return.

```
; 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))
```

(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.

(assoc 'g_arg1 'l_arg2)

RETURNS: the first top level element of l_arg2 whose *car* is *equal* to g_arg1.

NOTE: the test is make with the lisp function equal. Usually l_arg2 has an *a-list* structure and g_arg1 acts as key.

(assq 'g_arg1 'l_arg2)

RETURNS: the first top level element of l_arg2 whose *car* is equal to g_arg1 using the lisp function *eq*.

NOTE: This is faster than assoc since eq is faster than equal but lisp objects which print alike are not always *eq*. See the description of eq for more details.

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

(atom 'g_arg)

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

NOTE: (atom '()) returns t.

(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.

(bcdp 'g_arg)

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

NOTE: the name of this function is a throwback to the PDP-11 Lisp system.

(bigp 'g_arg)

RETURNS: t iff g_arg is a bignum.

(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. Currently one may dissect hunks and bignums with c..r as well although this is subject to change.

(concat ['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

(cons 'g_arg1 'g_arg2)

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

(copy 'g_arg)

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

(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.

(cpy1 'xvt_arg)

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

(cxr 'x_ind 'h_hunk)

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

(defprop ls_name g_val g_ind)

RETURNS: g_val.

SIDE EFFECT: The property list of ls_name is updated by adding g_val as the value of indicator g_ind.

NOTE: this is similar to putprop except that the arguments to defprop are not evaluated. ls_name may be a disembodied property list, see *get.*

(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 occurances 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])

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

NOTE: *delq* is 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!
```

(dtpr 'g_arg)

RETURNS: t iff g_arg is a list cell.

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

(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.]

(equal '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
```

(explode 'g_arg)

RETURNS: a list of single character symbols which *print* would use to print g_arg.

(explodec 'g_val)

RETURNS: the list of characters which print would use to print g_val except that special characters in symbols are not escaped (just as if patom were used to print them).

(exploden 'g_val)

RETURNS: a list of fixnums which print would use to print g_val except that special characters in symbols are not escaped (just as if patom were used to print them).

(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.

(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.

(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 looking at every other element for g_ind.

```
-> (putprop 'xlate 'a 'alpha)
a
-> (putprop 'xlate 'b 'beta)
b
-> (plist 'xlate)
(alpha a beta b)
-> (get 'xlate 'alpha)
a
-> (get '(nil fateman rjf sklower kls foderaro jkf) 'sklower)
kls
```

(get_pname 's_arg)

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

(getaccess 'a_array)

RETURNS: the access function for the array a_array.

NOTE: this function will most likely disappear in future releases.

(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.

(getaux 'a_array)

RETURNS: the auxiliary field for the array a_array.

NOTE: this function will most likely disappear in future releases.

(getchar '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.

(getcharn 's_arg 'x_index)

RETURNS: the fixnum representation of 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.

(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.

(getdelta 'a_array)

RETURNS: the delta field for a_array.

NOTE: this function will most likely disappear in future releases.

(getentry 'y_funchd)

RETURNS: the entry field of the binary y_funchd.

NOTE: this function will most likely disappear in future releases.

(getlength 'a_array)

RETURNS: the length field of the array a_array.

NOTE: this function will most likely disappear in future releases.

(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}

(hunksize 'h_arg)

RETURNS: the size of the hunk h_arg.

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

(implode 'l_arg)

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

RETURNS: The symbol whose print name is the result of concatenating the print names of the symbols in the list. Any fixnums are converted to the equivalent ascii character.

(intern 's_arg)

RETURNS: s_arg

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

(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)

(length 'l_arg)

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

(list ['g_arg1 ...])

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

(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.

(maknam 'l_arg)

RETURNS: what implode would return except the resulting symbol is uninterned.

(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.

(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 § 1.3.9).

(member '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: the test for equality is made with *equal.*

(memq 'g_arg1 'l_arg2)

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

NOTE: the test for equality is made with *eq.*

(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
```

(ncons 'g_arg)

RETURNS: a new list cell with g_arg as car and nil as cdr.

(not 'g_arg)

RETURNS: t iff g_arg is nil.

(nreverse 'l_arg)

RETURNS: the reverse of l_arg.

NOTE: The reverse is done in place, that is the list structure is modified. No new list cells are allocated.

(nthelem 'n_arg1 'l_arg2)

RETURNS: The n_arg1'*st* element of the list l_arg2.

NOTE: If n_arg1 is non-positive or greater than the length of the list, nil is returned.

(null 'g_arg)

RETURNS: t iff g_arg is nil.

EQUIVALENT TO: not.

(plist 's_name)

RETURNS: the property list of s_name.

(ptr 'g_arg)

RETURNS: a value cell initialize to point to g_arg.

(putaccess 'a_array 's_func)

RETURNS: s_func.

SIDE EFFECT: replaces the access field of a_array with s_func.

NOTE: this function will most likely disappear in future releases.

(putaux 'a_array 'g_aux)

RETURNS: s_aux.

SIDE EFFECT: replaces the auxillary field of a_array with g_aux.

NOTE: this function will most likely disappear in future releases.

(putdelta 'a_array 'x_delta)

RETURNS: x_delta.

SIDE EFFECT: replaces the delta field of a_array with x_delta.

NOTE: this function will most likely disappear in future releases.

(putdisc 'y_func 's_discipline)

RETURNS: s_discipline

SIDE EFFECT: the discipline field of y_func is set to s_discipline.

(putlength 'a_array 'x_length)

RETURNS: x_length

SIDE EFFECT: replaces the length field of a_array with x_length.

NOTE: this function will most likely disappear in future releases.

(putprop '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: this is similar to *defprop* except the arguments are evaluated to *putprop.* ls_name may be a disembodied property list, see *get.*

(quote g_arg)

RETURNS: g_arg.

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

(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!

(remob 's_symbol)

RETURNS: s_symbol

SIDE EFFECT: s_symbol is removed from the oblist.

(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*.

(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, one is to change the value element of x's symbol structure to point to a word of memory with a different value. Another 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, that is 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))*.

(reverse 'l_arg)

RETURNS: the reverse of the list l_arg.

NOTE: The *reverse* is performed by allocating new list cells to duplicate the top level of l_arg. This can be expensive if l_arg is large. The function nreverse will reverse the list without allocating new list cells.

(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.

(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.

(*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: This is the same as *rplacx* except you may replace uninitialized hunk entries. This is only to be used by functions such as *hunk* and *makhunk* which create hunks of sizes which are not powers of two.

(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.

(set 's_arg1 'g_arg2)

RETURNS: g_arg2.

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

(setplist 's_atm 'l_plist**)**

RETURNS: l_plist.

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

(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*.

(stringp 'g_arg**)**

RETURNS: t iff g_arg is a string.

(symbolp 'g_arg**)**

RETURNS: t iff g_arg is a symbol.

(type 'g_arg**)**

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

(typep 'g_arg**)**

EQUIVALENT TO: type.

(uconcat ['s_arg1 ...]**)**

RETURNS: a symbol whose pname is the result of concatenating the print names (pnames) of the s_arg*i*.

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

(valuep 'g_arg**)**

RETURNS: t iff g_arg is a value cell

CHAPTER 3

Arithmetic Functions

3.1. This chapter describes FRANZ LISP's functions for doing arithmetic. Often the same function is know by many names, such as *add* which is also *plus*, *sum*, and +. This is due to our desire to be compatible with other Lisps. The FRANZ LISP user is advised to avoid using functions with names such as + and * unless their arguments are fixnums. The lisp compiler takes advantage of the fact that their arguments are fixnums.

An attempt to divide by zero will cause a floating exception signal from the UNIX operating system. The user can catch and process this interrupt if he wishes (see the description of the *signal* function).

(abs 'n_arg)

RETURNS: the absolute value of n_arg.

(absval 'n_arg)

EQUIVALENT TO: abs.

(add ['n_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)

RETURNS: n_arg plus 1.

(acos 'fx_arg)

RETURNS: the arc cosine of fx_arg in the range 0 to πi.

(asin 'fx_arg)

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

(atan 'fx_arg1 'fx_arg2)

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

(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
key	8	9	10	11	12	13	14	15
result	¬ (x + y)	¬(x ⊕ y)	¬ x	¬ x + y	¬ y	x + ¬ y	¬ x + ¬ y	-1

(cos 'fx_angle)

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

(diff ['n_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.

(difference ['n_arg1 ...])

EQUIVALENT TO: diff.

(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.

(exp 'fx_arg)

RETURNS: *e* raised to the fx_arg power.

(expt 'n_base 'n_power)

RETURNS: n_base raised to the i_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.

(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.

(fixp 'g_arg)

RETURNS: t iff g_arg is a fixnum or bignum.

(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.

(floatp 'g_arg)

RETURNS: t iff g_arg is a flonum.

(greaterp ['n_arg1 ...])

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

NOTE: 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.

(haipart bx_number x_bits)

RETURNS: the x_bits high bits of |bx_number| if x_bits is positive, otherwise it returns the |x_bits| low bits of |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 |bx_number| + 1.

(lessp ['n_arg1 ...])

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

NOTE: 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.

(log 'fx_arg)

RETURNS: the natural logarithm of fx_arg.

(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.

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

RETURNS: the maximum value in the list of arguments.

(min 'n_arg1 ... **)**

RETURNS: the minimum value in the list of arguments.

(minus 'n_arg**)**

RETURNS: zero minus n_arg.

(minusp 'g_arg**)**

RETURNS: t iff g_arg is a negative number.

(mod 'i_dividend 'i_divisor**)**

RETURNS: the remainder when i_dividend is divided by i_divisor.

(numberp 'g_arg**)**

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

(numbp 'g_arg**)**

EQUIVALENT TO: numberp.

(onep 'g_arg**)**

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

(plus ['n_arg ...]**)**

EQUIVALENT TO: to add.

(plusp 'n_arg**)**

RETURNS: t iff n_arg is greater than zero.

(product ['n_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 ...])

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.

(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.

(remainder 'i_dividend 'i_divisor)

EQUIVALENT TO: mod.

(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.

(sin 'fx_angle)

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

(sqrt 'fx_arg)

RETURNS: the square root of fx_arg.

(sub1 'n_arg)

RETURNS: n_arg minus 1.

(sum ['n_arg1 ...])

EQUIVALENT TO: add and plus.

(times ['n_arg1 ...])

EQUIVALENT TO: product.

(zerop 'g_arg)

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

3.2. These functions are restricted to fixnum arguments in Maclisp. The lisp compiler will assume the arguments are fixnums and open code most of these functions.

(1+ 'n_arg)

EQUIVALENT TO: add1.

(1− 'n_arg)

EQUIVALENT TO: sub1.

(+ 'n_arg)

EQUIVALENT TO: add.

(* 'n_arg)

EQUIVALENT TO: times.

(− 'n_arg)

EQUIVALENT TO: difference.

(/ 'n_arg1 'n_arg2)

EQUIVALENT TO: quotient

(< 'n_arg1 'n_arg2)

EQUIVALENT TO: lessp.

(= 'g_arg1 'g_arg2)

EQUIVALENT TO: equal.

(> 'n_arg1 'n_arg2)

EQUIVALENT TO: greaterp.

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
```

(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 nil, 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.*

(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 *throw* is done during the evaluation of g_exp we will catch the value thrown iff 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 §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 nil 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.

(declare [g_arg ...])

RETURNS: nil

NOTE: this is a no-op to the evaluator. It has special meaning to the compiler.

(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.

(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.

```
; 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
```

(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 nil 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 nil then the car is evaluated and if it is non nil 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.

(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.

(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 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).*

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 nil value, then the error message associated with the error is printed before control returns to the errset.

(eval 'g_val**)**

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 g_val is of type value, then its value 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 nil 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 "subroutine", "function", "integer-function" or "real-function". If g_func is an array then this form is evaluated specially, see §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 §8.4 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.

(eval-when l_times g_exp1 ... g_expn)

WHERE: l_times is a list containing any combination of compile, eval and load.

RETURNS: nil if the symbol eval is not member of l_times, else returns the value of g_expn.

SIDE EFFECT: If eval is a member of l_times, then the forms g_exp*i* are evaluated.

NOTE: this is used mainly to control when the compiler evaluates forms.

(exec s_arg1 ...)

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

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

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

(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)*.

(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 't_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.

(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 sub**lists 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.

(mfunction entry 's_disc)

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

NOTE: 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 nil 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 nil 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 are evaluated from left to right. This is a prog body (obviously) and this means than any symbols seen are not evaluated, instead they are treated as labels. This also means that returns and go's are allowed.

(prog2 g_exp1 g_exp2 [g_exp3 ...])

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

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

(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 _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.

(putd 's_name 'u_func)

RETURNS: 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.

(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.

CHAPTER 5

Input/Output

The following functions are used to read and write to and from external devices 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 not both simultaneously. There are only a limited number of ports (20) and they will not be reclaimed unless you *close* them. All ports are reclaimed by a *resetio* call but this drastic step won't be necessary if the program closes what it uses. If you don't supply a port argument to a function which requires one or if you supply a bad port argument (such as nil) then FRANZ LISP will use the default port according to this scheme. If you are reading then the default port is the value of the symbol *piport* and if you are writing it 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 it if the symbol ^*w* is non nil (although it will still go to *ptport* if *ptport* is a valid port).

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

RETURNS: t

SIDE EFFECT: This is use 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 nil, 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 -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 that should be "-lF77 -lIf77". 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.

(close 'p_port**)**

RETURNS:

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.

(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.

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

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. After st_file is loaded, the standard fortran libraries will be searched to resolve external references.

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

(flatc 'g_form ['x_max])

RETURNS: the number of characters required to print g_form using *patom*. If x_max is given, and the *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.

(fseek 'p_port 'x_offset 'x_flag)

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

SIDE EFFECT: this function position 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 (the bytes between the end of the file and the new position will be filled with zeroes).

(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.

(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 nil, or else equal to the current readtable. See chapter XX for a description of readtables and their uses.

(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 after the buffer (of 512 characters) is filled or when *terpr* is called.

(outfile 's_filename)

RETURNS: a port or nil

SIDE EFFECT: this opens a port to write s_filename. The file opened is truncated by the *outfile* if it existed beforehand. If there are no free ports, outfile returns nil.

(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 then 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)

WHERE: xfs_arg is a fixnum, flonum or symbol.

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. The option *(F filename)* causes output to go to the file filename. The option *(P portname)* causes output to go to the port portname which should have been opened previously.

(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.

(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.

(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.

(resetio)

RETURNS: nil

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

(setsyntax 's_symbol 'sx_code ['ls_func])

RETURNS: t

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

(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 flush the buffer.

(terpri ['p_port])

EQUIVALENT TO: terpr.

(tyi ['p_port])

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

(tyipeek ['p_port])

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

NOTE: This does not actually read the character, it just peeks at it.

(tyo 'x_char ['p_port])

RETURNS: x_char.

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

(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 which one uses to interact with FRANZ LISP running in the UNIX environment.

(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. It allocates pages one at a time so that if an error occurs this means that all free storage has been allocated. 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 that 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.

(boundp 's_name)

RETURNS: nil if s_name is unbound, that is it has never be given a value. If x_name has the value g_val, then (nil . g_val) is returned.

(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.

(dumplisp s_name)

RETURNS: nil

SIDE EFFECT: the current lisp is dumped to the disk with the file name s_name. 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_expl ...)

SIDE EFFECT: l_time may contain any combination of the symbols *load*, *eval*, and *compile*. The effects of load and compile will is discussed in the section on the compiler. If eval is present however, this simply means that the expressions g_expl 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.

(gc)

RETURNS: nil

SIDE EFFECT: this causes a garbage collection.

NOTE: 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.

(include s_filename)

RETURNS: nil

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

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

(includef 's_filename)

RETURNS: nil

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

(maknum 'g_arg)

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

(opval 's_arg ['g_newval])

RETURNS: the value associated with s_arg before the call.

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

NOTE: *opval* keeps track of storage allocation. If s_arg is one of the data types then *opval* 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 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.

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. this 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, the second is the amount of time used by the garbage collector so far.

NOTE: 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 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)*.

(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.

(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 not show all of them.

(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 nil then 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.

(sstatus automatic-reset g_val)

RETURNS: g_val

SIDE EFFECT: If g_val is non nil then when an error occurs which no one wants to handle, a *reset* will be done instead of entering a primitive internal break loop.

(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.

(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 default 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.

(sstatus feature g_val)

RETURNS: g_val

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

(sstatus ignoreeof g_val)

RETURNS: g_val

SIDE EFFECT: If g_val is non nil then if a end of file (CNTL D on UNIX) is typed to the 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, a EOF will always cause lisp to exit.

(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.

NOTE: For a discussion of transfer tables, see the Section on the compiler.

(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).

(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

(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 as described in the UNIX manual under LOCALTIME(3).

EXAMPLE: *(status localtime)* ==> (20 51 16 29 5 80 0 1 nil)

(status syntax s_char)

RETURNS: a fixnum which is the syntax code associated with the character s_char in the current readtable.

NOTE: You cannot set the syntax code with with *(sstatus syntax ch)*, you must use *setsyntax*.

(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 33b"

(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 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.

(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.

CHAPTER 7

The Reader

The FRANZ LISP reader is controlled by a *readtable*. A readtable is an array of fixnums, one fixnum for each of the 128 ascii characters. The fixnums tell the reader what the properties of the each character are. The initial readtable is described below. The user may create new readtables using the *makereadtable* function. The current readtable is the value of the lisp symbol *readtable* which, like any lisp symbol, may be lambda bound to allow the user to change the reader syntax very quickly. The values which may appear in the readtable are:

type	value (decimal)	meaning	default
vnum	0	digit	0-9
vsign	1	plus or minus	+ -
vchar	2	alphabetic character	A-Z a-z ^H ! $ % & * , / : ; < = > ? @ ^ _ ` { | } ~
vsca	66	single char atom	none
vlpara	195	left paren	(
vrpara	196	right paren	)
vperd	197	period	.
vlbrkt	198	left bracket	[
vrbrkt	199	right bracket	]
veof	200	end of file	rubout
vsq	201	single quote	'
vdq	138	double quote, all characters between matching double quotes are escaped (i.e. treated as **vchar**)	|
vsd	137	string delimiter, all characters between matching delimiters are *concat*ed into an object of type string	"
verr	203	illegal character	null ^A-^I ^N-^Z ^\-^_
vsep	204	separator	^I-^M esc space
vspl	205	splicing macro character	none
vmac	206	macro character	none
vesc	143	escape character	\

The names in the type column are not known to Franz, we are just using them to tag the various classes. You must use the value in the second column. The default column shows the syntax values of characters in the raw lisp, i.e., the lisp which contains only machine language functions. The lisp which you get when you give the lisp command to the shell is an augmented version of the raw lisp, with additional lisp coded functions and changes in the readtable. The syntax changes in the lisp may differ from installation to installation but will probably include making one character be a comment character. In the lisp at Berkeley, semicolon is the

comment character. This was done by declaring it to be a splicing macro character which invokes the function *zapline* when seen.

To read the syntax of a character, you may use (**status syntax** s_char).

To change the syntax bits of a character, use the *setsyntax* function. There are two forms, one when you just want to change the syntax bits, and the other when you want to define a character macro. The first form is:

(**setsyntax** *'s_c 'x_value*)

Here *s_c* may be the character itself or it may be the fixnum representation of that character. *x_value* is one of the values given above in the second column. You should be careful when you change the syntax of a character as the change lasts until you explicitly change it back or until you begin with a new lisp. Also, some syntax changes are silly and will probably cause system errors (e.g. changing the syntax of an alphabetic character to be a **vnum**). The only syntax values you will probably ever use are: **vdq** and **vesc**. You should not change the syntax to **vspl** or **vmac** using the above form, instead it will be done automatically when you use the form below.

To declare a character macro use:

(**setsyntax** *'s_c 's_type 's_fcn*)

Where *s_c* is again either the character itself or its fixnum equivalent, type is *splicing* or *macro*, and *s_fcn* is either the name of a function expecting no arguments or is a lambda expression. The result of the setsyntax function is twofold: the readtable value for that character is changed to **vspl** or **vmac**, and the function is put on the property list of the character under the indicator "macro". The difference between a splicing macro and a macro is this: the value returned by a splicing macro is *nconc*ed to what has been read so far (i.e. (**nconc** sofar justreturned)), while the value returned by a macro is added to what has been read, (i.e (**nconc** sofar (**list** justread)). Thus if a splicing macro returns nil, then it isn't seen since (**nconc** any nil) ==> any. In particular, splicing macros are useful for conditional loading of lisp expressions.

FRANZ LISP treats left and right square brackets in a special way when building lists. A left bracket is just like a left parenthesis, and a right bracket matches a left bracket or all open left parentheses, whichever comes first.

When building atoms, a character with the syntax code **vesc** will cause the next character to be read in and treated as a **vchar**. To escape an entire string of characters, you surround them with matching characters having the **vdq** syntax code. To escape the **vdq** character within the string of characters you use any character of class **vesc**. The standard UNIX escape character, backslash ('\'), is in this class by default.

CHAPTER 8

Functions and Macros

8.1. valid function objects

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

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
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"†	8.4
foreign function	binary with discipline of "function"†	8.4
foreign integer function	binary with discipline of "integer-function"†	8.4
foreign real function	binary with discipline of "real-function"†	8.4
array	array object	9

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

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

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 won't 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 the number of arguments is lambda-bound to the single formal parameter of the lexpr function. The lexpr then accesses the arguments using the *arg* function.

When a function is compiled *special* declaration 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 features 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 key names.† 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 in the jkfmacs.l package. It allows you to write

```
(If (equal numb 0) then (print 'zero) (terpr)
 elseif (equal numb 1) then (print 'one) (terpr)
 else (print |I give up|))
```

which expands to

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

8.3.1. macro forms A macro is a function 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))))
```

†A record definition macro package especially suited for FRANZ LISP is in the planning stages at Berkeley. At this time the Maclisp *struct* package can be used.

```
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))*. Now *eval* evaluates that, and the value is *a* which 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, one 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 only three characters with associated character macros. One is semicolon for comments. The other two are backquote and comma which are used by the backquote character macro. 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
```

8.4. foreign subroutines and functions FRANZ LISP has the ability to dynamically load object files produced by other compilers and then call functions defined in those files. These functions are called *foreign* functions. There are four types of foreign functions and they are characterized by the type of result they return:

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.

A foreign function is accessed through a binary object just like a compiled lisp function. The difference is that the discipline field for a binary object of a foreign function is a string whose first character is s for a subroutine, f for a function, i for an integer-function and r for a real-function. Two functions are provided for the setting up of foreign functions. *Cfasl* loads an object file into the lisp system and sets up one foreign function binary object. If there is more than one function in an object file, *getaddress* can be used to set up further foreign function objects.

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

symbol arguments are passed unchanged to the foreign function. Fixnum and flonum arguments are copied into a temporary location and a pointer to the value is passed (this is because Fortran uses call by reference and it is dangerous to modify the contents of a fixnum or flonum which something else might point to). If an array object is an argument 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).

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 is 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 33b
```
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 used 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 33b
-> (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

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 lispvalues. 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 §1.3.9 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 up 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 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 When an array is created a raw array object is allocated with *marray* and the user is responsible for filling in the parts. 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 function 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 are know 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 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 be stored instead of pointers to the numbers as are 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 vector of (less than 128) general lisp objects you would be wise to look into using hunks.

†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.

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 interrupt handler is called and is given as an argument a description of the error which just occured. 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, cause a value to be returned from the error handler 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 occured 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 not 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 and error handler which is called in a manner similar to the 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. For a further discussion of *break-err-handler*, see section xx. The other standard error

handler is *debug-err-handler.* This handler is designed to be connected to **ER%all.** It is useful if your program uses *errset* and you want to look at the error 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 ^C.[†]. 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 ^C is typed, the lisp system sets a flag that an interrupt has been requested. This flag is checked at safe places within the interpreter and in the *qlinker* function. If the lisp system doesn't respond to the first ^C, another ^C 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 ^C 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). There are two user callable functions in the trace package: *trace* and *untrace*. The trace package will be loaded automatically when you first use the *trace* function. Both *trace*and *untrace* are nlambdas (their arguments are not evaluated). The form of a call to *trace* is

(trace *arg1 arg2 ...***)**

where the *argi* 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 ^D or (tracereturn) and execution will continue. Note that ^D 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.

(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.

---

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

(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.

The function *trace* returns a list of functions is was able to trace. The function *untrace* untraces those functions its is argument list. If the argument list is empty then all functions being traced are untraced. *Untrace* returns a list of functions untraced.

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.

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. macroexpansion 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 *fas*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. 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 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 file name. This is the only switch with an argument, which must follow the switch. For example
% liszt foo -o xxx.o
will compile foo and into xxx.o instead of the default foo.o.

q Run in quiet mode. The names of functions being compiled and various "Note"'s are not printed.

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. 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.

12.7. 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.

APPENDIX A

Index to FRANZ LISP Functions

APPENDIX B

Special Symbols

The values of these symbols have a predefined meaning. Some values are counters while others are simply flags whose value the user can change to affect the operation of lisp system. In all cases, only the value cell of the symbol is important, the function cell is not. The value of some of the symbols (like **ER%misc**) are functions - what this means is that the value cell of those symbols either contains a lambda expression, a binary object, or symbol with a function binding.

The values of the special symbols are:

$gccount$ — The number of garbage collections which have occurred.

$gcprint — If bound to a non nil value, then after each garbage collection and subsequent storage allocation a summary of storage allocation will be printed.

ER%all — The function which is the error handler for all errors (see §10)

ER%brk — The function which is the handler for the error signal generated by the evaluation of the *break* function (see §10).

ER%err — The function which is the handler for the error signal generated by the evaluation of the *err* function (see §10).

ER%misc — The function which is the handler of the error signal generated by one of the unclassified errors (see §10). Most errors are unclassified at this point.

ER%tpl — The function which is the handler to be called when an error has occurred which has not been handled (see §10).

ER%undef — The function which is the handler for the error signal generated when a call to an undefined function is made.

^w — When bound to a non nil value this will prevent output to the standard output port (poport) from reaching the standard output (usually a terminal). Note that ^w is a two character symbol and should not be confused with ^W which is how we would denote control-w. The value of ^w is checked when the standard output buffer is flushed which occurs after a *terpr*, *drain* or when the buffer overflows. This is most useful in conjunction with ptport described below. System error handlers rebind ^w to nil when they are invoked to assure that error messages are not lost. (This was introduced for Maclisp compatibility).

defmacro-for-compiling — The has an effect during compilation. If non-nil it causes macros defined by defmacro to be compiled and included in the object file.

environment — The UNIX environment in assoc list form.

errlist — When a *reset* is done, the value of errlist is saved away and control is thrown to the top level. *Eval* is then mapped over the saved away value of this list.

errport — This port is initially bound to the standard error file.

ibase — This is the input radix used by the lisp reader. It may be either eight or ten. Numbers followed by a decimal point are assumed to be decimal regardless of what ibase is.

linel — The line length used by the pretty printer, pp. This should be used by *print* but it is not at this time.

nil — This symbol represents the null list and thus can be written (). Its value is always nil. Any attempt to change the value will result in an error.

piport — Initially bound to the standard input (usually the keyboard). A read with no arguments reads from piport.

poport — Initially bound to the standard output (usually the terminal console). A print with no second argument writes to poport. See also: ^w and ptport.

ptport — Initially bound to nil. If bound to a port, then all output sent to the standard output will also be sent to this port as long as this port is not also the standard output (as this would cause a loop). Note that ptport will not get a copy of whatever is sent to poport if poport is not bound to the standard output.

readtable — The value of this is the current readtable. It is an array but you should NOT try to change the value of the elements of the array using the array functions. This is because the readtable is an array of bytes and the smallest unit the array functions work with is a full word (4 bytes). You can use *setsyntax* to change the values and *(status syntax ...)* to read the values.

t — This symbol always has the value t. It is possible to change the value of this symbol for short periods of time but you are strongly advised against it.

top-level — In a lisp system without /usr/lib/lisp/toplevel.l loaded, after a *reset* is done, the lisp system will *funcall* the value of top-level if it is non nil. This provides a way for the user to introduce his own top level interpreter. When /usr/lib/lisp/toplevel.l is loaded, it sets top-level to franz-top-level and changes the *reset* function so that once franz-top-level starts, it cannot be replaced by changing top-level. Franz-top-level does provide a way of changing the top level however, and that is through user-top-level.

user-top-level — If this is bound then after a *reset*, the value of this variable will be *funcall*ed.

APPENDIX C

Short Subjects.

The Garbage Collector

The garbage collector is invoked automatically whenever a collectable data type runs out. All data types are collectable except strings and atoms are not. After a garbage collection finishes, the collector will call the function *gcafter* which should be a lambda of one argument. The argument passed to *gcafter* is the name of the data type which ran out and caused the garbage collection. It is *gcafter*'s responsibility to allocate more pages of free space. The default *gcafter* makes its decision based on the percentage of space still in use after the garbage collection. If there is a large percentage of space still in use, *gcafter* allocates a larger amount of free space than if only a small percentage of space is still in use. The default *gcafter* will also print a summary of the space in use if the variable *$gcprint* is non nil. The summary always includes the state of the list and fixnum space and will include another type if it caused the garbage collection. The type which caused the garbage collection is preceded by an asterisk.

Debugging

There are two built-in functions to help you debug your programs: *baktrace* and *showstack*. When an error occurs (or when you type the interrupt character), you will be left at a break level with the state of the computation frozen in the stack. At this point, calling the function *showstack* will cause the contents of the lisp evaluation stack to be printed in reverse chronological order (most recent first). When the programs you are running are interpreted or traced, the output of *showstack* can be very verbose. The function *baktrace* prints a summary of what *showstack* prints. That is, if showstack would print a list, *baktrace* would only print the first element of the list. If you are running compiled code with the *(status translink)* non nil, then fast links are being made. In this case, there is not enough information on the stack for *showstack* and *baktrace*. Thus, if you are debugging compiled code you should probably do *(sstatus translink nil)*.

If the contents of the stack don't tell you enough about your problem, the next thing you may want to try is to run your program with certain functions traced. You can direct the trace package to stop program execution when it enters a function, allowing you to examine the contents of variables or call other functions. The trace package is documented in §11.

It is also possible to single step the evaluator and to look at stack frames within lisp. The programs which take advantage of these things are in /usr/lib/lisp and are called step and fixit. They are maintained by the people at Carnegie-Mellon (currently Lars Ericson is in charge of the code there). There are documentation files for these programs in /usr/lib/lisp as well. We run compiled lisp almost exclusively at Berkeley. Sites which run a mostly interpreted code should examine these files.

The Interpreter's Top Level

The top level interpreter for Franz, named *franz-top-level* is defined in /usr/lib/lisp/toplevel.l It is given control when the lisp system starts up because the variable top-level is bound to the symbol *franz-top-level.* The first action *franz-top-level* takes is to print out the name of the current version of the lisp system. Then it loads the file .lisprc from the HOME directory of the person invoking the lisp system if that file exists. The .lisprc file allows you to set up your own defaults, read in files, set up autoloading or anything else you might want to do to personalize the lisp system. Next, the top level goes into a prompt-read-eval-print loop. Each time around the loop, before printing the prompt it checks if the variable user-top-level is bound. If so, then the value of user-top-level will be *funcall*ed. This provides a convenient way for a user to introduce his own top level (Liszt, the lisp compiler, is an example of a program which uses this). If the user types a ^D (which is the end of file character), and the standard input is not from a keyboard, the lisp system will exit. If the standard input is a keyboard and if the value of *(status ignoreeof)* is nil, the lisp system will also exit. Otherwise the end of file will be ignored. When a *reset* is done the current value of *errlist* is saved away and control is thrown back up to the top level where *eval* is mapped over the saved value of *errlist.*