21. The I/O System

Zetalisp provides a powerful and flexible system for performing input and output to peripheral devices. To allow device independent I/O (that is, to allow programs to be written in a general way so that the program's input and output may be connected with any device), the Zetalisp I/O system provides the concept of an "I/O stream". Streams, their usage, and their implementation are described in this chapter. This chapter also describes the Lisp "I/O" operations **read** and **print** and the printed representation they use for Lisp objects.

21.1 The Character Set

Zetalisp represents characters as fixnums. The Lisp Machine's mapping between these numbers and the characters is listed here. The mapping is similar to ASCII, but somewhat modified to allow the use of the so-called SAIL extended graphics, while avoiding certain ambiguities present in ITS. For a long time ITS treated the **Backspace**, **Control-H**, and λ keys on the keyboard identically as character code 10 octal; this problem is avoided from the start in the Lisp Machine's mapping.

It is worth pointing out that although the Zetalisp character set is different from the PDP-10 character set, when files are transferred between Lisp Machines and PDP-10's the characters are automatically converted. Details of the mapping are explained below.

Fundamental characters are eight bits wide. Only those less than 200 octal (with the 200 bit off) are printing graphics; when output to a device they are assumed to print a character and move the "cursor" one character position to the right. (All software provides for variable-width fonts, so the term "character position" shouldn't be taken too literally.)

Characters in the range of 200 to 236 inclusive are used for special characters. Character 200 is a "null character", which does not correspond to any key on the keyboard. The null character is not used for anything much; **fasload** uses it internally. Characters 201 through 236 correspond to the special function keys on the keyboard such as **Return** and **Call**. The remaining characters are reserved for future expansion.

It should never be necessary for a user or a source program to know these numerical values. Indeed, they are likely to be changed in the future. There are symbolic names for all characters; see below.

Most of the special characters do not normally appear in files (although it is not forbidden for files to contain them). These characters exist mainly to be used as "commands" from the keyboard.

A few special characters, however, are "format effectors" which are just as legitimate as printing characters in text files. The names and meanings of these characters are:

Return	The "newline" character, which separates lines of text. We do not use the PDP-10 convention which separates lines by a pair of characters, a "carriage return" and a "linefeed".

Page	The "page separator" character, which separates pages of text.
Tab	The "tabulation" character, which spaces to the right until the next "tab stop". Tab stops are normally every 8 character positions.

The space character is considered to be a printing character whose printed image happens to be blank, rather than a format effector.

In some contexts, a fixnum can hold both a character code and a font number for that character. The following byte specifiers are defined:

%%ch-char *Variable*

The value of %%ch-char is a byte specifier for the field of a fixnum character that holds the character code.

%%ch-font *Variable*

The value of %%ch-font is a byte specifier for the field of a fixnum character that holds the font number.

Characters read in from the keyboard include a character code and control bits. A character cannot contain both a font number and control bits, since these data are both stored in the same bits. The following byte specifiers are provided:

%%kbd-char *Variable*

The value of %%kbd-char is a byte specifier for the field of a keyboard character that holds the normal eight-bit character code.

%%kbd-control *Variable*

The value of %%kbd-control is a byte specifier for the field of a keyboard character that is 1 if either Control key was held down.

%%kbd-meta *Variable*

The value of %%kbd-meta is a byte specifier for the field of a keyboard character that is 1 if either Meta key was held down.

%%kbd-super *Variable*

The value of %%kbd-super is a byte specifier for the field of a keyboard character that is 1 if either Super key was held down.

%%kbd-hyper *Variable*

The value of %%kbd-hyper is a byte specifier for the field of a keyboard character that is 1 if either Hyper key was held down.

This bit is also set if Control and/or Meta is typed in combination with Shift and a letter. Shift is much easier than Hyper to reach with the left hand.

%%kbd-control-meta *Variable*

The value of %%kbd-control-meta is a byte specifier for the four-bit field of a keyboard character that contains the above control bits. The least-significant bit is Control. The most significant bit is Hyper.

The following fields are used by some programs that encode signals from the mouse in a the format of a character. Refer to the window system documentation for an explanation of how these characters are generated.

%%kbd-mouse *Variable*

The value of %%kbd-mouse is a byte specifier for the bit in a keyboard character that indicates that the character is not really a character, but a signal from the mouse.

%%kbd-mouse-button *Variable*

The value of %%kbd-mouse-button is a byte specifier for the field in a mouse signal that says which button was clicked. The value is 0, 1, or 2 for the left, middle, or right button, respectively.

%%kbd-mouse-n-clicks *Variable*

The value of %%kbd-mouse-n-clicks is a byte specifier for the field in a mouse signal that says how many times the button was clicked. The value is one less than the number of times the button was clicked.

When any of the control bits (Control, Meta, Super, or Hyper) is set in conjunction with a letter, the letter will always be upper-case. The character codes that consist of a lower-case letter and non-zero control bits are "holes" in the character set which are never used for anything. Note that when Shift is typed in conjuction with Control and/or Meta and a letter, it means Hyper rather than Shift.

Since the control bits are not part of the fundamental 8-bit character codes, there is no way to express keyboard input in terms of simple character codes. However, there is a convention accepted by the relevant programs for encoding keyboard input into a string of characters: if a character has its Control bit on, prefix it with an alpha. If a character has its Meta bit on, prefix it with a beta. If a character has both its Control and Meta bits on, prefix it with an epsilon. If a character has its Super bit on, prefix it with a pi. If a character has its Hyper bit on, prefix it with a lambda. To get an alpha, beta, epsilon, pi, lambda, or equivalence into the string, quote it by prefixing it with an equivalence.

When characters are written to a file server computer that normally uses the ASCII character set to store text, Lisp Machine characters are mapped into an encoding that is reasonably close to an ASCII transliteration of the text. When a file is written, the characters are converted into this encoding; the inverse transformation is done when a file is read back. No information is lost. Note that the length of a file, in characters, will not be the same measured in original Lisp Machine characters as it will measured in the encoded ASCII characters. In the currently implemented ASCII file servers, the following encoding is used. All printing characters and any characters not mentioned explicitly here are represented as themselves. Codes 010 (lambda), 011 (gamma), 012 (delta), 014 (plus-minus), 015 (circle-plus), 177 (integral), 200 through 207 inclusive, 213 (Delete), and 216 and anything higher, are preceded by a 177; that is, 177 is used as a "quoting character" for these codes. Codes 210 (Overstrike), 211 (Tab), 212 (Line), and

214 (Page), are converted to their ASCII cognates, namely 010 (backspace), 011 (horizontal tab), 012 (line feed), and 014 (form feed) respectively. Code 215 (Return) is converted into 015 (carriage return) followed by 012 (line feed). Code 377 is ignored completely, and so cannot be stored in files.

000	center-dot (·)	040	space	100	@	140	`
001	down arrow (↓)	041	!	101	A	141	a
002	alpha (α)	042	"	102	B	142	b
003	beta (β)	043	#	103	C	143	c
004	and-sign (∧)	044	$	104	D	144	d
005	not-sign (¬)	045	%	105	E	145	e
006	epsilon (ε)	046	&	106	F	146	f
007	pi (π)	047	'	107	G	147	g
010	lambda (λ)	050	(	110	H	150	h
011	gamma (γ)	051	)	111	I	151	i
012	delta (δ)	052	*	112	J	152	j
013	uparrow (↑)	053	+	113	K	153	k
014	plus-minus (±)	054	,	114	L	154	l
015	circle-plus (⊕)	055	-	115	M	155	m
016	infinity (∞)	056	.	116	N	156	n
017	partial delta (∂)	057	/	117	O	157	o
020	left horseshoe (⊂)	060	0	120	P	160	p
021	right horseshoe (⊃)	061	1	121	Q	161	q
022	up horseshoe (∩)	062	2	122	R	162	r
023	down horseshoe (∪)	063	3	123	S	163	s
024	universal quantifier (∀)	064	4	124	T	164	t
025	existential quantifier (∃)	065	5	125	U	165	u
026	circle-X (⊗)	066	6	126	V	166	v
027	double-arrow (↔)	067	7	127	W	167	w
030	left arrow (←)	070	8	130	X	170	x
031	right arrow (→)	071	9	131	Y	171	y
032	not-equals (≠)	072	:	132	Z	172	z
033	diamond (altmode) (◊)	073	;	133	[	173	{
034	less-or-equal (≤)	074	<	134	\	174	|
035	greater-or-equal (≥)	075	=	135	]	175	}
036	equivalence (≡)	076	>	136	^	176	~
037	or (∨)	077	?	137	_	177	∫
200	Null character	210	Overstrike	220	Stop-output	230	Roman-iv
201	Break	211	Tab	221	Abort	231	Hand-up
202	Clear	212	Line	222	Resume	232	Hand-down
203	Call	213	Delete	223	Status	233	Hand-left
204	Terminal escape	214	Page	224	End	234	Hand-right
205	Macro/backnext	215	Return	225	Roman-i	235	System
206	Help	216	Quote	226	Roman-ii	236	Network
207	Rubout	217	Hold-output	227	Roman-iii		
237-377	reserved for the future						

The Lisp Machine Character Set

21.2 Printed Representation

People cannot deal directly with Lisp objects, because the objects live inside the machine. In order to let us get at and talk about Lisp objects, Lisp provides a representation of objects in the form of printed text; this is called the *printed representation*. This is what you have been seeing in the examples throughout this manual. Functions such as **print**, **prin1**, and **princ** take a Lisp object and send the characters of its printed representation to a stream. These functions (and the internal functions they call) are known as the *printer*. The **read** function takes characters from a stream, interprets them as a printed representation of a Lisp object, builds a corresponding object, and returns it; it and its subfunctions are known as the *reader*. (Streams are explained in section 21.5, page 391.)

This section describes in detail what the printed representation is for any Lisp object, and just what **read** does. For the rest of the chapter, the phrase "printed representation" will usually be abbreviated as "p.r."

21.2.1 What the Printer Produces

The printed representation of an object depends on its type. In this section, we will consider each type of object and explain how it is printed.

Printing is done either with or without *slashification*. The non-slashified version is nicer looking in general, but if you give it to **read** it won't do the right thing. The slashified version is carefully set up so that **read** will be able to read it in. The primary effects of slashification are that special characters used with other than their normal meanings (e.g. a parenthesis appearing in the name of a symbol) are preceded by slashes or cause the name of the symbol to be enclosed in vertical bars, and that symbols that are not from the current package get printed out with their package prefixes (a package prefix looks like a symbol followed by a colon).

For a fixnum or a bignum: if the number is negative, the printed representation begins with a minus sign ("-"). Then the value of the variable **base** is examined. If **base** is a positive fixnum, the number is printed out in that base (**base** defaults to 8). If it is a symbol with a **si:princ-function** property, the value of the property will be applied to two arguments: **minus** of the number to be printed, and the stream to which to print it (this is a hook to allow output in Roman numerals and the like). Otherwise the value of **base** is invalid and an error is signalled. Finally, if **base** equals 10. and the variable ***nopoint** is **nil**, a decimal point is printed out. Slashification does not affect the printing of numbers.

base *Variable*

The value of **base** is a number that is the radix in which fixnums are printed, or a symbol with a **si:princ-function** property. The initial value of **base** is 8.

***nopoint** *Variable*

If the value of ***nopoint** is **nil**, a trailing decimal point is printed when a fixnum is printed out in base 10. This allows the numbers to be read back in correctly even if **ibase** is not 10. at the time of reading. If ***nopoint** is non-**nil**, the trailing decimal points are suppressed. The initial value of ***nopoint** is **nil**.

For a flonum: the printer first decides whether to use ordinary notation or exponential notation. If the magnitude of the number is too large or too small, such that the ordinary notation would require an unreasonable number of leading or trailing zeroes, then exponential notation will be used. The number is printed as an optional leading minus sign, one or more digits, a decimal point, one or more digits, and an optional trailing exponent, consisting of the letter **e**, an optional minus sign, and the power of ten. The number of digits printed is the "correct" number; no information present in the flonum is lost and no extra trailing digits are printed that do not represent information in the flonum. Feeding the p.r. of a flonum back to the reader is always supposed to produce an equal flonum. Flonums are always printed in decimal; they are not affected by slashification or by **base** and ***nopoint**.

For a small flonum: the printed representation is very similar to that of a flonum, except that exponential notation is always used and the exponent is delimited by **s** rather than **e**.

For a rationalnum: the printed representation consists of the numerator, a backslash, and the denominator. For example, the value of **(rationalize .5)** is printed as **1\2**.

For a complexnum: the printed representation consists of the real part, the character **+**, the imaginary part, and the character **i**. At the present, they cannot be read back in, and perhaps the printed representation will be changed.

For a symbol: if slashification is off, the p.r. is simply the successive characters of the print-name of the symbol. If slashification is on, two changes must be made. First, the symbol might require a package prefix in order that **read** work correctly, assuming that the package into which **read** will read the symbol is the one in which it is being printed. See the section on packages (chapter 24, page 506) for an explanation of the package name prefix. Secondly, if the p.r. would not read in as a symbol at all (that is, if the print-name looks like a number, or contains special characters), then the p.r. must have some quoting for those characters, either by the use of slashes (**/**) before each special character, or by the use of vertical bars (**|**) around the whole name. The decision whether quoting is required is done using the readtable (see section 21.2.6, page 379), so it is always accurate provided that **readtable** has the same value when the output is read back in as when it was printed.

For a string: if slashification is off, the p.r. is simply the successive characters of the string. If slashification is on, the string is printed between double quotes and any characters inside the string that need to be preceded by slashes will be. Normally these are only double-quote and slash. Compatibly with Maclisp, carriage return is *not* ignored inside strings and vertical bars.

For an instance or an entity: if the object has a method for the **:print-self** message, that message is sent with three arguments: the stream to print to, the current *depth* of list structure (see below), and whether slashification is enabled. The object should print a suitable p.r. on the stream. See chapter 20, page 321 for documentation on instances. Most such objects print like "any other data type" below, except with additional information such as a name. Some objects print only their name when slashification is not in effect (when **princ**'ed). Some objects, including pathnames, use a printed representation that begins with #⊂, ends with ⊃, and contains sufficient information for the reader to reconstruct an equivalent object. See page 377.

For an array that is a named structure: if the array has a named structure symbol with a **named-structure-invoke** property that is the name of a function, then that function is called on five arguments: the symbol **:print-self**, the object itself, the stream to print to, the current *depth* of list structure (see below), and whether slashification is enabled. A suitable printed representation should be sent to the stream. This allows a user to define his own p.r. for his named structures; more information can be found in the named structure section (see page 312). Typically the printed representation used will start with either **#<** if it is not supposed to be readable, or with **#⊂** (see page 377) if it is supposed to be readable.

If the named structure symbol does not have a **named-structure-invoke** property, the printed-representation is like the p.r. for random data-types: a number sign and a less than sign, the named structure symbol, the numerical address of the array, and a greater than sign.

Other arrays: the p.r. starts with a number sign and a less-than sign. Then the "**art-**" symbol for the array type is printed. Next the dimensions of the array are printed, separated by hyphens. This is followed by a space, the machine address of the array, and a greater-than sign.

Conses: The p.r. for conses tends to favor *lists*. It starts with an open-parenthesis. Then the *car* of the cons is printed and the *cdr* of the cons is examined. If it is **nil**, a close-parenthesis is printed. If it is anything else but a cons, space dot space followed by that object is printed. If it is a cons, we print a space and start all over (from the point *after* we printed the open-parenthesis) using this new cons. Thus, a list is printed as an open-parenthesis, the p.r.'s of its elements separated by spaces, and a close-parenthesis.

This is how the usual printed representations such as **(a b (foo bar) c)** are produced.

The following additional feature is provided for the p.r. of conses: as a list is printed, **print** maintains the length of the list so far and the depth of recursion of printing lists. If the length exceeds the value of the variable **prinlength**, **print** will terminate the printed representation of the list with an ellipsis (three periods) and a close-parenthesis. If the depth of recursion exceeds the value of the variable **prinlevel**, then the list will be printed as "**". These two features allow a kind of abbreviated printing that is more concise and suppresses detail. Of course, neither the ellipsis nor the "**" can be interpreted by **read**, since the relevant information is lost.

prinlevel *Variable*

prinlevel can be set to the maximum number of nested lists that can be printed before the printer will give up and just print a "**". If it is **nil**, which it is initially, any number of nested lists can be printed. Otherwise, the value of **prinlevel** must be a fixnum.

prinlength *Variable*

prinlength can be set to the maximum number of elements of a list that will be printed before the printer will give up and print a "...". If it is **nil**, which it is initially, any length list may be printed. Otherwise, the value of **prinlength** must be a fixnum.

For any other data type: The printed representation starts with **#<** and ends with **>**. This sort of printed representation cannot be read back in. The **#<** is followed by the "**dtp-**" symbol for this datatype, a space, and the octal machine address of the object. The object's name, if one can be determined, often appears before the address. If this style of printed representation is being used for a named structure or instance, other interesting information may appear as well. Finally a greater-than sign (>) is printed.

Including the machine address in the p.r. makes it possible to tell two objects of this kind apart without explicitly calling eq on them. This can be very useful during debugging. It is important to know that if garbage collection is turned on, objects will occasionally be moved, and therefore their octal machine addresses will be changed. It is best to shut off garbage collection temporarily when depending on these numbers.

Printed representations that start with "#<" can never be read back. This can be a problem if, for example, you are printing a structure into a file with the intent of reading it in later. The following feature allows you to make sure that what you are printing may indeed be read with the reader.

si:print-readably *Variable*

When si:print-readably is bound to t, the printer will signal an error if there is an attempt to print an object that cannot be interpreted by read. When the printer sends a :print-self or a :print message, it assumes that this error checking is done for it. Thus it is possible for these messages *not* to signal an error, if they see fit.

si:printing-random-object (*object stream . keywords*) &body *body* *Macro*

The vast majority of objects that define :print-self messages have much in common. This macro is provided for convenience so that users do not have to write out that repetitious code. It is also the preferred interface to si:print-readably. With no keywords, si:printing-random-object checks the value of si:print-readably and signals an error if it is not nil. It then prints a number sign and a less-than sign, evaluates the forms in *body*, then prints a space, the octal machine address of the object and a greater-than sign. A typical use of this macro might look like:

```
(si:printing-random-object (ship stream)
  (princ (typep ship) stream)
  (tyo #\space stream)
  (prin1 (ship-name ship) stream))
```

This might print #<ship "ralph" 23655126>.

The following keywords may be used to modify the behaviour of si:printing-random-object:

:no-pointer	This suppresses printing of the octal address of the object.
:typep	This prints the result of (typep *object*) after the less-than sign. In the example above, this option could have been used instead of the first two forms in the body.

sys:print-not-readable (error) *Condition*

This condition is signaled by si:print-readably when the object cannot be printed readably.

The condition instance supports the operation :object, which returns the object that was being printed.

If you want to control the printed representation of some object, usually the right way to do it is to make the object an array that is a named structure (see page 312), or an instance of a flavor (see chapter 20, page 321). However, occasionally it is desirable to get control over all printing of objects, in order to change, in some way, how they are printed. If you need to do this, the best way to proceed is to customize the behavior of **si:print-object** (see page 431), which is the main internal function of the printer. All of the printing functions, such as **print** and **princ**, as well as **format**, go through this function. The way to customize it is by using the "advice" facility (see section 27.10, page 592).

21.2.2 What The Reader Accepts

The purpose of the reader is to accept characters, interpret them as the p.r. of a Lisp object, and return a corresponding Lisp object. The reader cannot accept everything that the printer produces; for example, the p.r.'s of arrays (other than strings), compiled code objects, closures, stack groups, etc., cannot be read in. However, it has many features that are not seen in the printer at all, such as more flexibility, comments, and convenient abbreviations for frequently-used unwieldy constructs.

This section shows what kind of p.r.'s the reader understands, and explains the readtable, reader macros, and various features provided by **read**.

In general, the reader operates by recognizing tokens in the input stream. Tokens can be self-delimiting or can be separated by delimiters such as whitespace. A token is the p.r. of an atomic object such as a symbol or number, or a special character such as a parenthesis. The reader reads one or more tokens until the complete p.r. of an object has been seen, then constructs and returns that object.

The reader understands the p.r.'s of fixnums in a way more general than is employed by the printer. Here is a complete description of the format for fixnums.

Let a *simple fixnum* be a string of digits, optionally preceded by a plus sign or a minus sign, and optionally followed by a trailing decimal point. A simple fixnum will be interpreted by **read** as a fixnum. If the trailing decimal point is present, the digits will be interpreted in decimal radix; otherwise, they will be considered as a number whose radix is the value of the variable **ibase**.

ibase *Variable*

The value of **ibase** is a number that is the radix in which fixnums are read. The initial value of **ibase** is 8.

read will also understand a simple fixnum, followed by an underscore ("_") or a circumflex ("^"), followed by another simple fixnum. The two simple fixnums will be interpreted in the usual way; the character in between indicates an operation that will then performed on the two fixnums. The underscore indicates a binary "left shift"; that is, the fixnum to its left is doubled the number of times indicated by the fixnum to its right. The circumflex multiplies the fixnum to its left by **ibase** the number of times indicated by the fixnum to its right. (The second simple fixnum is not allowed to have a leading minus sign.) Examples: **645_6** means **64500** (in octal) and **645^3** means **645000**.

Here are some examples of valid representations of fixnums to be given to **read**:

```
4
23456.
-546
+45^+6
2_11
```

The syntax for bignums is identical to the syntax for fixnums. A number is a bignum rather than a fixnum if and only if it is too large to be represented as a fixnum. Here are some examples of valid representations of bignums:

```
72361356126536125376512375126535123712635
-123456789.
105_1000
105_1000.
```

The syntax for a flonum is an optional plus or minus sign, optionally some digits, a decimal point, and one or more digits. Such a flonum or a simple fixnum, followed by an "e" (or "E") and a simple fixnum, is also a flonum; the fixnum after the "e" is the exponent of 10 by which the number is to be scaled. (The exponent is not allowed to have a trailing decimal point.) If the exponent is introduced by "s" (or "S") rather than "e", the number is a small-flonum. Here are some examples of printed-representations that read as flonums:

```
0.0
1.5
14.0
0.01
.707
-.3
+3.14159
6.03e23
1E-9
1.e3
```

Here are some examples of printed-representations that read as small-flonums:

```
0s0
1.5s9
-42S3
1.s5
```

The syntax for a rationalnum is an integer, a backslash, and another integer. Here are examples:

```
1\2
100000000000000\3
```

A string of letters, numbers, and "extended alphabetic" characters is recognized by the reader as a symbol, provided it cannot be interpreted as a number. Alphabetic case is ignored in symbols; lower-case letters are translated to upper-case. When the reader sees the p.r. of a symbol, it *interns* it on a *package* (see chapter 24, page 506, for an explanation of interning and the package system). Symbols may start with digits; you could even have one named **-345T**; **read** will accept this as a symbol without complaint. If you want to put strange characters (such

as lower-case letters, parentheses, or reader macro characters) inside the name of a symbol, put a slash before each strange character. If you want to have a symbol whose print-name looks like a number, put a slash before some character in the name. You can also enclose the name of a symbol in vertical bars, which quote all characters inside them except vertical bars and slashes, which must be quoted with slash.
Examples of symbols:

```
foo
bar/(baz/)
34w23
|Frob Sale|
```

The reader will also recognize strings, which should be surrounded by double-quotes. If you want to put a double-quote or a slash inside a string, precede it by a slash.
Examples of strings:

```
"This is a typical string."
"That is known as a /"cons cell/" in Lisp."
```

When **read** sees an open-parenthesis, it knows that the p.r. of a cons is coming, and calls itself recursively to get the elements of the cons or the list that follows. Any of the following are valid:

```
(foo . bar)
(foo bar baz)
(foo . (bar . (baz . nil)))
(foo bar . quux)
```

The first is a cons, whose car and cdr are both symbols. The second is a list, and the third is exactly the same as the second (although **print** would never produce it). The fourth is a "dotted list"; the cdr of the last cons cell (the second one) is not **nil**, but **quux**.

Whenever the reader sees any of the above, it creates new cons cells; it never returns existing list structure. This contrasts with the case for symbols, as very often **read** returns symbols that it found interned in the package rather than creating new symbols itself. Symbols are the only thing that work this way.

The dot that separates the two elements of a dotted-pair p.r. for a cons is only recognized if it is surrounded by delimiters (typically spaces). Thus dot may be freely used within print-names of symbols and within numbers. This is not compatible with Maclisp; in Maclisp **(a.b)** reads as a cons of symbols **a** and **b**, whereas in Zetalisp it reads as a list of a symbol **a.b**.

If the circle-X ("⊗") character is encountered, it is an octal escape, which may be useful for including weird characters in the input. The next three characters are read and interpreted as an octal number, and the character whose code is that number replaces the circle-X and the digits in the input stream. This character is always taken to be an alphabetic character, just as if it had been preceded by a slash.

21.2.3 Macro Characters

Certain characters are defined to be macro characters. When the reader sees one of these, it calls a function associated with the character. This function reads whatever syntax it likes and returns the object represented by that syntax. Macro characters are always token delimiters; however, they are not recognized when quoted by slash or vertical bar, nor when inside a string. Macro characters are a syntax-extension mechanism available to the user. Lisp comes with several predefined macro characters:

Quote (') is an abbreviation to make it easier to put constants in programs. '*foo* reads the same as (quote *foo*).

Semicolon (;) is used to enter comments. The semicolon and everything up through the next carriage return are ignored. Thus a comment can be put at the end of any line without affecting the reader.

Backquote (`) makes it easier to write programs to construct lists and trees by using a template. See section 17.2.2, page 251, for details.

Comma (,) is part of the syntax of backquote and is invalid if used other than inside the body of a backquote. See section 17.2.2, page 251, for details.

Sharp sign (#) introduces a number of other syntax extensions. See the following section. Unlike the preceding characters, sharp sign is not a delimiter. A sharp sign in the middle of a symbol is an ordinary character.

The function set-syntax-macro-char (see page 380) can be used to define your own macro characters.

21.2.4 Sharp-sign Constructs

The reader's syntax includes several abbreviations introduced by sharp sign (#). These take the general form of a sharp sign, a second character which identifies the syntax, and following arguments. Certain abbreviations allow a decimal number or certain special "modifier" characters between the sharp sign and the second character. Here are the currently-defined sharp sign constructs; more are likely to be added in the future.

#/ #/*x* reads in as the number which is the character code for the character *x*. For example, #/a is equivalent to 141 but clearer in its intent. This is the recommended way to include character constants in your code. Note that the slash causes this construct to be parsed correctly by the editors, EMACS and ZWEI.

As in strings, upper and lower-case letters are distinguished after #/. Any character works after #/, even those that are normally special to read, such as parentheses. Even non-printing characters may be used, although for them #\ is preferred.

A semi-obsolete method of specifying control bits in a character is to insert the characters α, β, ε, π and ʃλ between the # and the /. Those stand for control, meta, control-meta, super and hyper, respectively. This syntax should be converted to the new

#\control-meta-x syntax described below.

#\ #*name* reads in as the number that is the character code for the non-printing character symbolized by *name*. A large number of character names are recognized; these are documented below (section 21.2.5, page 378). For example, #\return reads in as a fixnum, being the character code for the Return character in the Lisp Machine character set. In general, the names that are written on the keyboard keys are accepted. In addition, all the nonalphanumeric characters have names. The abbreviations cr for return and sp for space are accepted, since these characters are used so frequently. The page separator character is called page, although form and clear-screen are also accepted since the keyboard has one of those legends on the page key. The rules for reading *name* are the same as those for symbols; thus upper and lower-case letters are not distinguished, and the name must be terminated by a delimiter such as a space, a carriage return, or a parenthesis.

When the system types out the name of a special character, it uses the same table as the #\ reader; therefore any character name typed out is acceptable as input.

#\ can also be used to read in the names of characters that have control and meta bits set. The syntax looks like #\control-meta-b to get a "B" character with the control and meta bits set. You can use any of the prefix bit names control, meta, hyper, and super. They may be in any order, and case is not significant. Prefix bit names can be abbreviated as single letters, and control may be spelled ctrl as it is on the keyboards. The last hyphen may be followed by a single character or by any of the special character names normally recognized by #\. A single character is treated the same way the reader normally treats characters in symbols; if you want to use a lower-case character or a special character such as a parenthesis, you must precede it by a slash character. Examples: #\Hyper-Super-A, \meta-hyper-roman-i, #\CTRL-META-/(.

greek (or front) and top are also allowed as names in the #\ construct. Thus, #\top-g is equivalent to #/↑ or #\uparrow. #\top-g should be used if you are specifying the keyboard commands of a program and the mnemonic significance belongs to the "G" rather than to the actual character code.

#^ #^*x* is exactly like #\control-*x* if the input is being read by Zetalisp; it generates ASCII Control-*x*. In Maclisp *x* is converted to upper case and then exclusive-or'ed with 100 (octal). Thus #^*x* always generates the character returned by tyi if the user holds down the control key and types *x*. (In Maclisp #\control-*x* sets the bit set by the Control key when the TTY is open in fixnum mode.)

#' #'*foo* is an abbreviation for (function *foo*). *foo* is the p.r. of any object. This abbreviation can be remembered by analogy with the ' macro-character, since the function and quote special forms are somewhat analogous.

#, #,*foo* evaluates *foo* (the p.r. of a Lisp form) at read time, unless the compiler is doing the reading, in which case it is arranged that *foo* will be evaluated when the QFASL file is loaded. This is a way, for example, to include in your code complex list-structure constants that cannot be written with quote. Note that the reader does not put quote around the result of the evaluation. You must do this yourself if you want it, typically by using the ' macro-character. An example of a case where you do not want quote around it is when this object is an element of a constant list.

#. #.*foo* evaluates *foo* (the p.r. of a lisp form) at read time, regardless of who is doing the reading.

#' #' is a construct for repeating an expression with some subexpressions varying. It is an abbreviation for writing several similar expressions or for the use of **mapc**. Each subexpression that is to be varied is written as a comma followed by a list of the things to substitute. The expression is expanded at read time into a **progn** containing the individual versions.

```
    #'(send stream ',(:clear-input :clear-output))
```
expands into
```
    (progn (send stream ':clear-input)
           (send stream ':clear-output))
```

Multiple repetitions can be done in parallel by using commas in several subexpressions:
```
    #'(renamef ,("foo" "bar") ,("ofoo" "obar"))
```
expands into
```
    (progn (renamef "foo" "ofoo")
           (renamef "bar" "obar"))
```

If you want to do multiple independent repetitions, you must use nested #' constructs. Individual commas inside the inner #' apply to that #'; they vary at maximum speed. To specify a subexpression that varies in the outer #', use two commas.
```
    #'#'(print (* ,(5 7) ,,(11. 13.)))
```
expands into
```
    (progn (progn (print (* 5 11.)) (print (* 7 11.)))
           (progn (print (* 5 13.)) (print (* 7 13.))))
```

#O #O *number* reads *number* in octal regardless of the setting of **ibase**. Actually, any expression can be prefixed by #O; it will be read with **ibase** bound to 8.

#X #X *number* reads *number* in radix 16. (hexadecimal) regardless of the setting of *ibase*. As with #O, any expression can be prefixed by #X.

The letters A through F are used as the digits beyond 9, but if a number contains one of these, it must begin with a sign in order to be properly recognized as a number. Thus, #X+FF is the same as 377, but #XFF is the symbol **ff**.

#R #*radix*R *number* reads *number* in radix *radix* regardless of the setting of **ibase**. As with #O, any expression can be prefixed by #*radix*R; it will be read with **ibase** bound to *radix*. *radix* must consist of digits only, and it is read in decimal.

For example, #3R102 is another way of writing 11. and #11R32 is another way of writing 35. Bases larger than ten use the letters starting with A as the additional digits.

#Q #Q *foo* reads as *foo* if the input is being read by Zetalisp, otherwise it reads as nothing (whitespace).

#M #M *foo* reads as *foo* if the input is being read into Maclisp, otherwise it reads as nothing (whitespace).

#N #N *foo* reads as *foo* if the input is being read into NIL or compiled to run in NIL, otherwise it reads as nothing (white space).

#+ This abbreviation provides a read-time conditionalization facility similar to, but more general than, that provided by #M, #N, and #Q. It is used as #+*feature form*. If *feature* is a symbol, then this is read as *form* if (status feature *feature*) is t. If (status feature *feature*) is nil, then this is read as whitespace. Alternately, *feature* may be a boolean expression composed of and, or, and not operators and symbols representing items that may appear on the (status features) list. (or lispm amber) represents evaluation of the predicate (or (status feature lispm) (status feature amber)) in the read-time environment.

For example, #+lispm *form* makes *form* exist if being read by Zetalisp, and is thus equivalent to #Q *form*. Similarly, #+maclisp *form* is equivalent to #M *form*. #+(or lispm nil) *form* will make *form* exist on either Zetalisp or in NIL. Note that items may be added to the (status features) list by means of (sstatus feature *feature*), thus allowing the user to selectively interpret or compile pieces of code by parameterizing this list. See page 650.

Here is a list of features with standard meanings:

lispm	This feature is present on any Lisp machine (no matter what version of hardware or software).
maclisp	This feature is present in Maclisp.
nil	This feature is present in NIL (New Implementation of Lisp).
mit	This feature is present in the MIT Lisp machine system, which is what this manual is about.
symbolics	This feature is present in the Symbolics version of the Lisp machine system. With luck, you should have no reason to be using that.
chaos	This feature is present in Lisp machine systems that use the Chaosnet protocol for their local network (regardless of details of hardware interfacing to the network).
ether	This feature is present in Lisp machines that use the Xerox-PARC Ethernet protocol for their local network communication.

#- #-*feature form* is equivalent to #+(not *feature*) *form*.

#< This is not legal reader syntax. It is used in the p.r. of objects that cannot be read back in. Attempting to read a #< will cause an error.

#⊂

This is used in the p.r. of miscellaneous objects (usually named structures or instances) that can be read back in. #⊂ should be followed by a typename and any other data needed to construct an object, terminated with a ⊃. For example, a pathname might print as

```
        #⊂FS:ITS-PATHNAME "AI: RMS; TEST 5"⊃
```

The typename is a keyword that read uses to figure out how to read in the rest of the printed representation and construct the object. It is read in in package user (but it can contain a package prefix). The resulting symbol should either have a si:read-instance property or be the name of a flavor that handles the :read-instance operation.

In the first case, the property is applied as a function to the typename symbol itself and the input stream. In the second, the handler for that operation is applied to the operation name (as always), the typename symbol, and the input stream (three arguments, but the first is implicit and not mentioned in the defmethod). self will be nil and instance variables should not be referred to.

In either case, the handler function should read the remaining data from the stream, and construct and return the datum it describes. It should return with the ⊃ character waiting to be read from the input stream (:untyi it if necessary). read will get an error after it is returned to if a ⊃ character is not next.

The typename can be any symbol with an appropriate property or flavor, not necessarily related to the type of object that is created; but for clarity, it is good if it is the same as the typep of the object printed. Since the type symbol is passed to the handler, one flavor's handler can be inherited by many other flavors and can examine the type symbol read in to decide what flavor to construct.

The function set-syntax-#-macro-char (see page 380) can be used to define your own sharp sign abbreviations.

21.2.5 Special Character Names

The following are the recognized special character names, in alphabetical order except with synonyms together. These names can be used after a "#\" to get the character code for that character. Most of these characters type out as this name enclosed in a lozenge. First we list the special function keys.

abort	break	call	clear-input, clear
delete	end	hand-down	hand-left
hand-right	hand-up	help	hold-output
line, lf	macro, back-next	network	
overstrike, backspace, bs		page, form, clear-screen	
quote	resume	return, cr	
roman-i	roman-ii	roman-iii	roman-iv
rubout	space, sp	status	stop-output
system	tab	terminal, esc	

These are printing characters that also have special names because they may be hard to type on a PDP-10.

altmode	circle-plus	delta	gamma
integral	lambda	plus-minus	uparrow
center-dot	down-arrow	alpha	beta
and-sign	not-sign	epsilon	pi
lambda	gamma	delta	up-arrow
plus-minus	circle-plus	infinity	partial-delta
left-horseshoe	right-horseshoe	up-horseshoe	down-horseshoe
universal-quantifier		existential-quantifier	
circle-x	double-arrow	left-arrow	right-arrow
not-equal	altmode	less-or-equal	greater-or-equal
equivalence	or-sign		

The following are special characters sometimes used to represent single and double mouse clicks. The buttons can be called either l, m, r or 1, 2, 3 depending on stylistic preference. These characters all contain the %%kbd-mouse bit.

mouse-l-1 = mouse-1-1	mouse-l-2 = mouse-1-2
mouse-m-1 = mouse-2-1	mouse-m-2 = mouse-2-2
mouse-r-1 = mouse-3-1	mouse-r-2 = mouse-3-2

21.2.6 The Readtable

There is a data structure called the *readtable* which is used to control the reader. It contains information about the syntax of each character. Initially it is set up to give the standard Lisp meanings to all the characters, but the user can change the meanings of characters to alter and customize the syntax of characters. It is also possible to have several readtables describing different syntaxes and to switch from one to another by binding the symbol readtable.

readtable *Variable*

The value of readtable is the current readtable. This starts out as the initial standard readtable. You can bind this variable to change temporarily the readtable being used.

si:initial-readtable *Variable*

The value of si:initial-readtable is the initial standard readtable. You should not ever change the contents of this readtable; only examine it by using it as the *from-readtable* argument to copy-readtable or set-syntax-from-char.

The user can program the reader by changing the readtable in any of three ways. The syntax of a character can be set to one of several predefined possibilities. A character can be made into a *macro character*, whose interpretation is controlled by a user-supplied function which is called when the character is read. The user can create a completely new readtable, using the readtable compiler (SYS: IO; RTC LISP) to define new kinds of syntax and to assign syntax classes to characters. Use of the readtable compiler is not documented here.

copy-readtable &optional *from-readtable to-readtable*

from-readtable, which defaults to the current readtable, is copied. If *to-readtable* is unsupplied or nil, a fresh copy is made. Otherwise *to-readtable* is clobbered with the copy. Use copy-readtable to get a private readtable before using the following functions to change the syntax of characters in it. The value of readtable at the start of a Lisp

Machine session is the initial standard readtable, which usually should not be modified.

set-syntax-from-char *to-char from-char* &optional *to-readtable from-readtable*
Makes the syntax of *to-char* in *to-readtable* be the same as the syntax of *from-char* in *from-readtable*. *to-readtable* defaults to the current readtable, and *from-readtable* defaults to the initial standard readtable.

set-character-translation *from-char to-char* &optional *readtable*
Changes *readtable* so that *from-char* will be translated to *to-char* upon read-in, when *readtable* is the current readtable. This is normally used only for translating lower case letters to upper case. Character translations are turned off by slash, string quotes, and vertical bars. *readtable* defaults to the current readtable.

set-syntax-macro-char *char function* &optional *readtable*
Changes *readtable* so that *char* is a macro character. When *char* is read, *function* is called. *readtable* defaults to the current readtable.

function is called with two arguments, *list-so-far* and the input stream. When a list is being read, *list-so-far* is that list (nil if this is the first element). At the "top level" of read, *list-so-far* is the symbol :toplevel. After a dotted-pair dot, *list-so-far* is the symbol :after-dot. *function* may read any number of characters from the input stream and process them however it likes.

function should return three values, called *thing*, *type*, and *splice-p*. *thing* is the object read. If *splice-p* is nil, *thing* is the result. If *splice-p* is non-nil, then when reading a list *thing* replaces the list being read—often it will be *list-so-far* with something else nconc'ed onto the end. At top-level and after a dot, if *splice-p* is non-nil the *thing* is ignored and the macro-character does not contribute anything to the result of read. *type* is a historical artifact and is not really used; nil is a safe value. Most macro character functions return just one value and let the other two default to nil.

function should not have any side-effects other than on the stream and *list-so-far*. Because of the way the rubout-handler works, *function* can be called several times during the reading of a single expression in which the macro character only appears once.

char is given the same syntax that single-quote, backquote, and comma have in the initial readtable (it is called :macro syntax).

set-syntax-#-macro-char *char function* &optional *readtable*
Causes *function* to be called when *# char* is read. *readtable* defaults to the current readtable. The function's arguments and return values are the same as for normal macro characters, documented above. When *function* is called, the special variable si:xr-sharp-argument contains nil or a number that is the number or special bits between the *#* and *char*.

set-syntax-from-description *char description* &optional *readtable*

Sets the syntax of *char* in *readtable* to be that described by the symbol *description*. The following descriptions are defined in the standard readtable:

si:alphabetic	An ordinary character such as "A".
si:break	A token separator such as "(". (Obviously left parenthesis has other properties besides being a break.
si:whitespace	A token separator that can be ignored, such as " ".
si:single	A self-delimiting single-character symbol. The initial readtable does not contain any of these.
si:slash	The character quoter. In the initial readtable this is "/".
si:verticalbar	The symbol print-name quoter. In the initial readtable this is "|".
si:doublequote	The string quoter. In the initial readtable this is '"'.
si:macro	A macro character. Don't use this; use **set-syntax-macro-char**.
si:circlecross	The octal escape for special characters. In the initial readtable this is "⊗".

These symbols will probably be moved to the standard keyword package at some point. *readtable* defaults to the current readtable.

setsyntax *character arg2 arg3*

This exists only for Maclisp compatibility. The above functions are preferred in new programs. The syntax of *character* is altered in the current readtable, according to *arg2* and *arg3*. *character* can be a fixnum, a symbol, or a string, i.e. anything acceptable to the **character** function. *arg2* is usually a keyword; it can be in any package since this is a Maclisp compatibility function. The following values are allowed for *arg2*:

:macro	The character becomes a macro character. *arg3* is the name of a function to be invoked when this character is read. The function takes no arguments, may **tyi** or **read** from **standard-input** (i.e. may call **tyi** or **read** without specifying a stream), and returns an object which is taken as the result of the read.
:splicing	Like **:macro** but the object returned by the macro function is a list that is nconced into the list being read. If the character is read anywhere except inside a list (at top level or after a dotted-pair dot), then it may return (), which means it is ignored, or (*obj*), which means that *obj* is read.
:single	The character becomes a self-delimiting single-character symbol. If *arg3* is a fixnum, the character is translated to that character.
nil	The syntax of the character is not changed, but if *arg3* is a fixnum, the character is translated to that character.
a symbol	The syntax of the character is changed to be the same as that of the character *arg2* in the standard initial readtable. *arg2* is converted to a character by taking the first character of its print name. Also if *arg3* is a fixnum, the character is translated to that character.

setsyntax-sharp-macro *character type function* &optional *readtable*

This exists only for Maclisp compatibility. set-syntax-#-macro-char is preferred. If *function* is nil, #*character* is turned off, otherwise it becomes a macro that calls *function*. *type* can be :macro, :peek-macro, :splicing, or :peek-splicing. The splicing part controls whether *function* returns a single object or a list of objects. Specifying peek causes *character* to remain in the input stream when *function* is called; this is useful if *character* is something like a left parenthesis. *function* gets one argument, which is nil or the number between the # and the *character*.

21.3 Input Functions

Most of these functions take optional arguments called *stream* and *eof-option*. *stream* is the stream from which the input is to be read; if unsupplied it defaults to the value of standard-input. The special pseudo-streams nil and t are also accepted, mainly for Maclisp compatibility. nil means the value of standard-input (i.e. the default) and t means the value of terminal-io (i.e. the interactive terminal). This is all more-or-less compatible with Maclisp, except that instead of the variable standard-input Maclisp has several variables and complicated rules. For detailed documentation of streams, refer to section 21.5, page 391.

eof-option controls what happens if input is from a file (or any other input source that has a definite end) and the end of the file is reached. If no *eof-option* argument is supplied, an error will be signalled. If there is an *eof-option*, it is the value to be returned. Note that an *eof-option* of nil means to return nil if the end of the file is reached; it is *not* equivalent to supplying no *eof-option*.

Functions such as read, which read an "object" rather than a single character, will always signal an error, regardless of *eof-option*, if the file ends in the middle of an object. For example, if a file does not contain enough right parentheses to balance the left parentheses in it, read will complain. If a file ends in a symbol or a number immediately followed by end-of-file, read will read the symbol or number successfully and when called again will see the end-of-file and obey *eof-option*. If a file contains ignorable text at the end, such as blank lines and comments, read will not consider it to end in the middle of an object and will obey *eof-option*.

These end-of-file conventions are not completely compatible with Maclisp. Maclisp's deviations from this are generally considered to be bugs rather than features.

The functions below that take *stream* and *eof-option* arguments can also be called with the stream and eof-option in the other order. This functionality is only for compatibility with old Maclisp programs, and should never be used in new programs. The functions attempt to figure out which way they were called by seeing whether each argument is a plausible stream. Unfortunately, there is an ambiguity with symbols: a symbol might be a stream and it might be an eof-option. If there are two arguments, one being a symbol and the other being something that is a valid stream, or only one argument, which is a symbol, then these functions will interpret the symbol as an eof-option instead of as a stream. To force them to interpret a symbol as a stream, give the symbol an si:io-stream-p property whose value is t.

Note that all of these functions will echo their input if used on an interactive stream (one which supports the :rubout-handler operation; see below.) The functions that input more than one character at a time (read, readline) allow the input to be edited using rubout. tyipeek echoes all of the characters that were skipped over if tyi would have echoed them; the character not removed from the stream is not echoed either.

read &optional *stream eof-option*

read reads in the printed representation of a Lisp object from *stream*, builds a corresponding Lisp object, and returns the object. The details have been explained above. (This function can take its arguments in the other order, for Maclisp compatibility only; see the note above.)

read-preserve-delimiters *Variable*

Certain printed representations given to read, notably those of symbols and numbers, require a delimiting character after them. (Lists do not, because the matching close-parenthesis serves to mark the end of the list.) Normally read will throw away the delimiting character if it is "whitespace", but will preserve it (with a :untyi stream operation) if the character is syntactically meaningful, since it may be the start of the next expression.

If read-preserve-delimiters is bound to t around a call to read, no delimiting characters will be thrown away, even if they are whitespace. This may be useful for certain reader macros or special syntaxes.

tyi &optional *stream eof-option*

tyi inputs one character from *stream* and returns it. The character is echoed if *stream* is interactive, except that Rubout is not echoed. The Control, Meta, etc. shifts echo as "C-", "M-", etc.

The :tyi stream operation is preferred over the tyi function for some purposes. Note that it does not echo. See page 391.

(This function can take its arguments in the other order, for Maclisp compatibility only; see the note above.)

read-for-top-level &optional *stream eof-option*

This is a slightly different version of read. It differs from read only in that it ignores close-parentheses seen at top level, and it returns the symbol si:eof if the stream reaches end-of-file if you have not supplied an *eof-option* (instead of signalling an error as read would). This version of read is used in the system's "read-eval-print" loops.

(This function can take its arguments in the other order, for uniformity with read only; see the note above.)

read-check-indentation &optional *stream eof-option*

This is like read, but validates the input based on indentation. It assumes that the input data is formatted to follow the usual convention for source files, that an open-parenthesis in column zero indicates a top-level list (with certain specific exceptions). An open-parenthesis in column zero encountered in the middle of a list is more likely to result from close-parentheses missing before it than from a mistake in indentation.

If read-check-indentation finds an open-parenthesis following a return character in the middle of a list, it invents enough close-parentheses to close off all pending lists, and returns. The offending open-parenthesis is :untyi'd so it can begin the next list, as it probably should. End of file in the middle of a list is handled likewise.

read-check-indentation notifies the caller of the incorrect formatting by signaling the condition sys:missing-closeparen. This is how the compiler is able to record a warning about the missing parentheses. If a condition handler proceeds, read goes ahead and invents close-parentheses.

There are a few special forms that are customarily used around function definitions—for example, eval-when, local-declare, and comment. Since it is desirable to begin the function definitions in column zero anyway, read-check-indentation allows a list to begin in column zero within one of these special forms. A non-nil si:may-surround-defun property identifies the symbols for which this is allowed.

read-check-indentation *Variable*

This variable is non-nil during a read in which indentation is being checked.

readline &optional *stream eof-option options*

readline reads in a line of text, terminated by a return. It returns the line as a character string, *without* the return character. This function is usually used to get a line of input from the user. If rubout processing is happening, then *options* is passed as the list of options to the rubout handler. One option that is particularly useful is the :do-not-echo option (see page 430), which you can use to suppress the echoing of the return character that terminates the line. (This function can take its arguments in the other order, for Maclisp compatibility only; see the note above.)

readline-trim &optional *stream eof-option options*

This is like readline except that leading and trailing spaces and tabs are discarded from the value before it is returned.

readch &optional *stream eof-option*

This function is provided only for Maclisp compatibility, since in the Zetalisp characters are always represented as fixnums. readch is just like tyi, except that instead of returning a fixnum character, it returns a symbol whose print name is the character read in. The symbol is interned in the current package. This is just like a Maclisp "character object". (This function can take its arguments in the other order, for Maclisp compatibility only; see the note above.)

tyipeek &optional *peek-type stream eof-option*

This function is provided mainly for Maclisp compatibility; the :tyipeek stream operation is usually clearer (see page 392).

What tyipeek does depends on the *peek-type*, which defaults to nil. With a *peek-type* of nil, tyipeek returns the next character to be read from *stream*, without actually removing it from the input stream. The next time input is done from *stream* the character will still be there; in general, (= (tyipeek) (tyi)) is t.

If *peek-type* is a fixnum less than 1000 octal, then tyipeek reads characters from *stream* until it gets one equal to *peek-type*. That character is not removed from the input stream.

If *peek-type* is t, then tyipeek skips over input characters until the start of the printed representation of a Lisp object is reached. As above, the last character (the one that starts an object) is not removed from the input stream.

The form of tyipeek supported by Maclisp in which *peek-type* is a fixnum not less than 1000 octal is not supported, since the readtable formats of the Maclisp reader and the Zetalisp reader are quite different.

Characters passed over by tyipeek are echoed if *stream* is interactive.

prompt-and-read *type-of-parsing format-string* &rest *format-args*

prompt-and-read reads some sort of object from query-io, parsing it according to *type-of-parsing*, and prompting by calling format using *format-string* and *format-args*.

type-of-parsing is either a keyword or a list starting with a keyword and continuing with a list of options and values, whose meanings depend on the keyword used.

The keywords defined are

:eval-sexp — This keyword directs prompt-and-read to accept a Lisp expression. It is evaluated, and the value is returned by prompt-and-read.

If the Lisp expression is not a constant or quoted, the user is asked to confirm the value it evaluated to.

:eval-sexp-or-end — This keyword directs prompt-and-read to accept a Lisp expression or just the character End. If End is typed, prompt-and-read returns nil as its first value and #\end as its second value. Otherwise, things proceed as for :eval-sexp.

:read — This keyword directs prompt-and-read to read an object and return it, with no evaluation.

:number — This keyword directs prompt-and-read to read and return a number. It will insist on getting a number, forcing the user to rub out anything else.

:string — This keyword directs prompt-and-read to read a line and return its contents as a string, using readline.

:string-or-nil — This keyword directs prompt-and-read to read a line and return its contents as a string, using readline-trim. In addition, if the result would be empty, nil is returned instead of the empty string.

:pathname — This keyword directs prompt-and-read to read a line and parse it as a pathname, merging it with the defaults. You can specify the defaults to use by passing a list of the form

```
(:pathname :defaults defaults-alist-or-pathname)
```

as the *type-of-parsing* argument.

:fquery This keyword directs prompt-and-read to query the user for a fixed set of alternatives, using fquery. *type-of-parsing* should always be a list, whose car is :fquery and whose cdr is a list to be passed as the list of options (fquery's first argument).
Example:

```
(prompt-and-read '(:fquery
                           . ,format:y-or-p-options)
                         "Eat it? ")
```

is equivalent to

```
(y-or-n-p "Eat it? ")
```

This keyword is most useful as a way to get to fquery when going through an interface defined to call prompt-and-read.

The following functions are related functions which do not operate on streams. Most of the text at the beginning of this section does not apply to them.

read-from-string *string* &optional *eof-option* (*start* *0*) *end*

The characters of *string* are given successively to the reader, and the Lisp object built by the reader is returned. Macro characters and so on will all take effect. If *string* has a fill-pointer it controls how much can be read.

eof-option is what to return if the end of the string is reached, as with other reading functions. *start* is the index in the string of the first character to be read. *end* is the index at which to stop reading; that point is treated as end of file.

read-from-string returns two values; the first is the object read and the second is the index of the first character in the string not read. If the entire string was read, this will be either the length of the string or 1 more than the length of the string.

Example:

```
(read-from-string "(a b c)") => (a b c) and 7
```

readlist *char-list*

This function is provided mainly for Maclisp compatibility. *char-list* is a list of characters. The characters may be represented by anything that the function character accepts: fixnums, strings, or symbols. The characters are given successively to the reader, and the Lisp object built by the reader is returned. Macro characters and so on will all take effect.

If there are more characters in *char-list* beyond those needed to define an object, the extra characters are ignored. If there are not enough characters, an "eof in middle of object" error is signalled.

See also the with-input-from-string special form (page 150).

sys:read-error (error) *Condition*

This condition name classifies all errors detected by the reader per se.

sys:end-of-file (error) *Condition*

All errors signaled to report end of file possess this condition name.

The :stream operation on the condition instance returns the stream on which end of file was reached.

sys:read-end-of-file (sys:read-error sys:end-of-file error) *Condition*

Whenever read signals an error for end of file, the condition object possesses this condition name.

sys:read-list-end-of-file (sys:read-end-of-file sys:read-error sys:end-of-file error) *Condition*

This condition is signaled when read detects end of file in the middle of a list.

In addition to the :stream operation provided because sys:end-of-file is one of the proceed types, the condition instance supports the :list operation, which returns the list read so far.

Proceed type :no-action is provided. If it is used, the reader invents a close-parenthesis to close off the list. Within read-check-indentation, the reader signals the error only once, no matter how many levels of list are unterminated.

sys:read-string-end-of-file (sys:read-end-of-file sys:read-error sys:end-of-file error) *Condition*

This is signaled when read detects end of file in the middle of a string delimited by doublequotes.

The :string operation on the condition instance returns the string read so far.

sys:read-symbol-end-of-file (sys:read-end-of-file sys:read-error sys:end-of-file error) *Condition*

This is signaled when read detects end of file in the middle of a symbol delimited by vertical bars.

The :string operation on the condition instance returns the print name read so far.

sys:missing-closeparen (condition) *Condition*

This condition, which is not an error, is signaled when read-check-indentation finds an open-parenthesis in column zero within a list.

Proceed type :no-action is provided. On proceeding, the reader invents enough close-parentheses to close off all the lists that are pending.

21.4 Output Functions

These functions all take an optional argument called *stream*, which is where to send the output. If unsupplied *stream* defaults to the value of standard-output. If *stream* is nil, the value of standard-output (i.e. the default) is used. If it is t, the value of terminal-io is used (i.e. the interactive terminal). If *stream* is a list of streams, then the output is performed to all of the streams (this is not implemented yet, and an error is signalled in this case). This is all more-or-less compatible with Maclisp, except that instead of the variable standard-output Maclisp has several variables and complicated rules. For detailed documentation of streams, refer to section 21.5, page 391.

prin1 *x* &optional *stream*
: prin1 outputs the printed representation of *x* to *stream*, with slashification (see page 367). *x* is returned.

prin1-then-space *x* &optional *stream*
: prin1-then-space is like prin1 except that output is followed by a space.

print *x* &optional *stream*
: print is just like prin1 except that output is preceded by a carriage return and followed by a space. *x* is returned.

princ *x* &optional *stream*
: princ is just like prin1 except that the output is not slashified. *x* is returned.

tyo *char* &optional *stream*
: tyo outputs the character *char* to *stream*.

terpri &optional *stream*
: terpri outputs a carriage return character to stream.

The format function (see page 411) is very useful for producing nicely formatted text. It can do anything any of the above functions can do, and it makes it easy to produce good looking messages and such. format can generate a string or output to a stream.

The grindef function (see page 426) is useful for formatting Lisp programs.

See also the with-output-to-string special form (page 151).

stream-copy-until-eof *from-stream to-stream* &optional *leader-size*
: stream-copy-until-eof inputs characters from *from-stream* and outputs them to *to-stream*, until it reaches the end-of-file on the *from-stream*. For example, if x is bound to a stream for a file opened for input, then (stream-copy-until-eof x terminal-io) will print the file on the console.

: If *from-stream* supports the :line-in operation and *to-stream* supports the :line-out operation, then stream-copy-until-eof will use those operations instead of :tyi and :tyo, for greater efficiency. *leader-size* will be passed as the argument to the :line-in operation.

beep &optional *beep-type* (*stream* terminal-io)

This function is intended to attract the user's attention by causing an audible beep, or flashing the screen, or something similar. If the stream supports the :beep operation, then this function sends it a :beep message, passing *type* along as an argument. Otherwise it just causes an audible beep on the terminal. *type* is a keyword selecting among several different beeping noises. The allowed types have not yet been defined; *type* is currently ignored and should always be nil. (The :beep operation is described on page 396.)

cursorpos &rest *args*

This function exists primarily for Maclisp compatibility. Usually it is preferable to send the appropriate messages (see the window system documentation).

cursorpos normally operates on the standard-output stream; however, if the last argument is a stream or t (meaning terminal-io) then cursorpos uses that stream and ignores it when doing the operations described below. Note that cursorpos only works on streams that are capable of these operations, for instance windows. A stream is taken to be any argument that is not a number and not a symbol, or that is a symbol other than nil with a name more than one character long.

(cursorpos) => (*line* . *column*), the current cursor position.

(cursorpos *line column*) moves the cursor to that position. It returns t if it succeeds and nil if it doesn't.

(cursorpos *op*) performs a special operation coded by *op*, and returns t if it succeeds and nil if it doesn't. *op* is tested by string comparison, it is not a keyword symbol and may be in any package.

F	Moves one space to the right.
B	Moves one space to the left.
D	Moves one line down.
U	Moves one line up.
T	Homes up (moves to the top left corner). Note that t as the last argument to cursorpos is interpreted as a stream, so a stream *must* be specified if the T operation is used.
Z	Home down (moves to the bottom left corner).
A	Advances to a fresh line. See the :fresh-line stream operation.
C	Clears the window.
E	Clear from the cursor to the end of the window.
L	Clear from the cursor to the end of the line.
K	Clear the character position at the cursor.
X	B then K.

exploden *x*

exploden returns a list of characters (as fixnums) that are the characters that would be typed out by (princ *x*) (i.e. the unslashified printed representation of *x*).
Example:

```
(exploden '(+ /12 3)) => (50 53 40 61 62 40 63 51)
```

explodec *x*

explodec returns a list of characters represented by symbols that are the characters that would be typed out by (princ *x*) (i.e. the unslashified printed representation of *x*).
Example:

```
        (explodec '(+ /12 3)) => ( /( + /  /1 /2 /  /3 /) )
```

(Note that there are slashified spaces in the above list.)

explode *x*

explode returns a list of characters represented by symbols that are the characters that would be typed out by (prin1 *x*) (i.e. the slashified printed representation of *x*).
Example:

```
        (explode '(+ /12 3)) => ( /( + /  // /1 /2 /  /3 /) )
```

(Note that there are slashified spaces in the above list.)

flatsize *x*

flatsize returns the number of characters in the slashified printed representation of *x*.

flatc *x*

flatc returns the number of characters in the unslashified printed representation of *x*.

21.5 I/O Streams

Many programs accept input characters and produce output characters. The method for performing input and output to one device is very different from the method for some other device. We would like our programs to be able to use any device available, but without each program having to know about each device.

Zetalisp makes this possible with *i/o streams*. A stream is a source and/or sink of characters or bytes. A set of *operations* is available with every stream; operations include things like "output a character" and "input a character". The way to perform an operation to a stream is the same for all streams, although what happens inside the stream is very different depending on what kind of a stream it is. So all a program has to know is how to deal with streams using the standard, generic operations. A programmer creating a new kind of stream only needs to implement the appropriate standard operations.

A stream is a message-receiving object. This means that it is something that you can apply to arguments. The first argument is a keyword symbol which is the name of the operation you wish to perform. The rest of the arguments depend on what operation you are doing. Message-passing and generic operations are explained in the flavor chapter (chapter 20, page 321).

Some streams can only do input, some can only do output, and some can do both. Some operations are only supported by some streams. Also, there are some operations that the stream may not support by itself, but will work anyway, albeit slowly, because the "stream default handler" can handle them. If you have a stream, there is an operation called **:which-operations** that will return a list of the names of all of the operations that are supported "natively" by the stream. *All* streams support **:which-operations**, and so it may not be in the list itself.

All input streams support all the standard input operations, and all output streams support all the standard output operations. All bidirectional streams support both.

21.5.1 Standard Input Stream Operations

:tyi &optional *eof* *Operation on streams*

The stream will input one character and return it. For example, if the next character to be read in by the stream is a "C", then the form

```
        (funcall s ':tyi)
```

will return the value of #/C (that is, 103 octal). Note that the **:tyi** operation will not "echo" the character in any fashion; it just does the input. The **tyi** function (see page 383) will do echoing when reading from the terminal.

The optional *eof* argument to the **:tyi** message tells the stream what to do if it gets to the end of the file. If the argument is not provided or is **nil**, the stream will return **nil** at the end of file. Otherwise it will signal an error, and print out the argument as the error message. Note that this is *not* the same as the eof-option argument to **read**, **tyi**, and related functions.

The **:tyi** operation on a binary input stream will return a non-negative number, not necessarily to be interpreted as a character.

:tyipeek &optional *eof* *Operation on streams*

Peeks at the next character or byte from the stream without discarding it. The next :tyi or :tyipeek operation will get the same character.

eof is the same as in the :tyi operation: if nil, end of file returns nil; otherwise, end of file is an error and *eof* is used as the error message.

:untyi *char* *Operation on streams*

Unread the character or byte *char*; that is to say, put it back into the input stream so that the next :tyi operation will read it again. For example,

```
        (funcall s ':untyi 120)
        (funcall s ':tyi) ==> 120
```

This operation is used by read, and any stream that supports :tyi must support :untyi as well.

You are only allowed to :untyi one character before doing a :tyi, and the character you :untyi must be the last character read from the stream. That is, :untyi can only be used to "back up" one character, not to stuff arbitrary data into the stream. You also can't :untyi after you have peeked ahead with :tyipeek since that does one :untyi itself. Some streams implement :untyi by saving the character, while others implement it by backing up the pointer to a buffer.

:string-in *eof-option string* &optional (*start 0*) *end* *Operation on streams*

Reads characters from the stream and stores them into the array *string*. Many streams can implement this far more efficiently that repeated :tyi's. *start* and *end*, if supplied, delimit the portion of *string* to be stored into. *eof-option* if non-nil is an error message and an error is signalled if end-of-file is reached on the stream before the string has been filled. If *eof-option* is nil, any number of characters before end-of-file is acceptable, even no characters.

If *string* has an array-leader, the fill pointer is adjusted to *start* plus the number of characters stored into *string*.

Two values are returned: the index of the next position in *string* to be filled, and a flag that is non-nil if end-of-file was reached before *string* was filled. Most callers will not need to look at either of these values.

string may be any kind of array, not necessarily a string; this is useful when reading from a binary input stream.

:line-in &optional *leader* *Operation on streams*

The stream should input one line from the input source, and return it as a string with the carriage return character stripped off. Contrary to what you might assume from its name, this operation is not much like the readline function.

Many streams have a string that is used as a buffer for lines. If this string itself were returned, there would be problems caused if the caller of the stream attempted to save the string away somewhere, because the contents of the string would change when the next line was read in. In order to solve this problem, the string must be copied. On the other hand, some streams don't reuse the string, and it would be wasteful to copy it on every :line-in operation. This problem is solved by using the *leader* argument to :line-in. If *leader* is nil (the default), the stream will not bother to copy the string and the caller should not rely on the contents of that string after the next operation on the stream. If *leader* is t, the stream will make a copy. If *leader* is a fixnum then the stream will make a copy with an array leader *leader* elements long. (This is used by the editor, which represents lines of buffers as strings with additional information in their array-leaders, to eliminate an extra copy operation.)

If the stream reaches the end-of-file while reading in characters, it will return the characters it has read in as a string and return a second value of t. The caller of the stream should therefore arrange to receive the second value, and check it to see whether the string returned was a whole line or just the trailing characters after the last carriage return in the input source.

This message should be implemented by all input streams whose data are characters.

:read-until-eof *Operation on streams*
Discard all data from the stream until it is at end of file, or do anything else with the same result.

:close &optional *ignore* *Operation on streams*
Release resources associated with the string, when it is not going to be used any more. On some kinds of streams, this may do nothing. On chaosnet streams, it closes the chaosnet connection, and on file streams, it closes the input file on the file server.

The argument is accepted for compatibility with :close on output streams.

21.5.2 Standard Output Stream Operations

:tyo *char* *Operation on streams*
The stream will output the character *char*. For example, if s is bound to a stream, then the form

```
        (funcall s ':tyo #/B)
```

will output a B to the stream. For binary output streams, the argument is a non-negative number rather than specifically a character.

:fresh-line *Operation on streams*
This tells the stream that it should position itself at the beginning of a new line. If the stream is already at the beginning of a fresh line it should do nothing; otherwise it should output a carriage return. If the stream cannot tell whether it is at the beginning of a line, it should always output a carriage return.

:string-out *string* &optional *start end* *Operation on streams*
The characters of the string are successively output to the stream. This operation is provided for two reasons; first, it saves the writing of a loop which is used very often, and second, many streams can perform this operation much more efficiently than the equivalent sequence of :tyo operations.

If *start* and *end* are not supplied, the whole string is output. Otherwise a substring is output; *start* is the index of the first character to be output (defaulting to 0), and *end* is one greater than the index of the last character to be output (defaulting to the length of the string). Callers need not pass these arguments, but all streams that handle :string-out must check for them and interpret them appropriately.

:line-out *string* &optional *start end* *Operation on streams*

The characters of the string, followed by a carriage return character, are output to the stream. *start* and *end* optionally specify a substring, as with :string-out. If the stream doesn't support :line-out itself, the default handler will turn it into a bunch of :tyos.

This message should be implemented by all output streams whose data are characters.

:close &optional *mode* *Operation on streams*

The stream is "closed" and no further output operations should be performed on it. However, it is all right to :close a closed stream.

This operation does nothing on streams for which it is not meaningful.

The *mode* argument is normally not supplied. If it is :abort, we are abnormally exiting from the use of this stream. If the stream is outputting to a file, and has not been closed already, the stream's newly-created file will be deleted; it will be as if it was never opened in the first place. Any previously existing file with the same name will remain undisturbed.

:eof *Operation on streams*

Indicates the end of data on an output stream. This is different from :close because some devices allow multiple data files to be transmitted without closing. :close implies :eof when the stream is an output stream and the close mode is not :abort.

This operation does nothing on streams for which it is not meaningful.

21.5.3 Asking Streams What They Can Do

All streams are supposed to support certain operations which enable a program using the stream to ask which operations are available.

:which-operations *Operation on streams*

This returns a list of operations handled "natively" by the stream. Certain operations not in the list may work anyway, but slowly, so it is just as well that any programs that work with or without them will choose not to use them.

Also, :which-operations is by convention not included in the list.

:operation-handled-p *operation* *Operation on streams*

This returns t if *operation* is handled "natively" by the stream: if *operation* is a member of the :which-operations list, or is :which-operations.

:send-if-handles *operation* &rest *arguments* *Operation on streams*

This performs the operation *operation*, with the specified *arguments*, only if the stream can handle it. If *operation* is handled, this is the same as sending an *operation* message directly, but if *operation* is not handled, using :send-if-handles avoids any error.

If *operation* is handled, :send-if-handles returns whatever values the execution of the *operation* returns. If *operation* is not handled, :send-if-handles returns nil.

:direction *Operation on streams*

This operation returns :input, :output, or :bidirectional for a bidirectional stream.

There are a few kinds of streams, which cannot do either input or output, for which the :direction operation returns nil. For example, open with the :direction keyword specified as nil returns a stream-like object which cannot do input or output but can handle certain file inquiry operations such as :truename and :creation-date.

:characters *Operation on streams*

This operation returns t if the data input or output on the stream are characters, or nil if they are just numbers (as for a stream reading a non-text file).

21.5.4 Operations for Interactive Streams

The operations :listen, :tyi-no-hang, :rubout-handler and :beep are intended for interactive streams, which communicate with the user. :listen and :tyi-no-hang are supported in a trivial fashion by other streams, for compatibility.

:listen *Operation on streams*

On an interactive device, the :listen operation returns non-nil if there are any input characters immediately available, or nil if there is no immediately available input. On a non-interactive device, the operation always returns non-nil except at end-of-file.

The main purpose of :listen is to test whether the user has hit a key, perhaps trying to stop a program in progress.

:tyi-no-hang &optional *eof* *Operation on streams*

Just like :tyi except that it returns nil rather than waiting if it would be necessary to wait in order to get the character. This lets the caller check efficiently for input being available and get the input if there is any.

:tyi-no-hang is different from :listen because it reads a character.

Streams for which the question of whether input is available is not meaningful will treat this operation just like :tyi. So will chaosnet file streams. Although in fact reading character from a file stream may involve a delay, these delays are *supposed* to be insignificant, so we pretend they do not exist.

:rubout-handler *options function* &rest *args* *Operation on streams*

This is supported by interactive bidirectional streams, such as windows on the terminal, and is described in its own section below (see section 21.7, page 427).

:beep &optional *type* *Operation on streams*

This is supported by interactive streams. It attracts the attention of the user by making an audible beep and/or flashing the screen. *type* is a keyword selecting among several different beeping noises. The allowed types have not yet been defined; *type* is currently ignored and should always be nil.

21.5.5 Cursor Positioning Stream Operations

:read-cursorpos &optional (*units* ':pixel) *Operation on streams*

This operation is supported by all windows and some other streams.

It returns two values, the current *x* and *y* coordinates of the cursor. It takes one optional argument, which is a symbol indicating in what units *x* and *y* should be; the symbols :pixel and :character are understood. :pixel means that the coordinates are measured in display pixels (bits), while :character means that the coordinates are measured in characters horizontally and lines vertically.

This operation and :increment-cursorpos are used by the format "~T" request (see page 414), which is why "~T" doesn't work on all streams. Any stream that supports this operation should support :increment-cursorpos as well.

Some streams return a meaningful value for the horizontal position but always return zero for the vertical position. This is sufficient for "~T" to work.

:increment-cursorpos *x-increment y-increment* &optional (*units* ':pixel) *Operation on streams*

Moves the stream's cursor left or down according to the specified increments, as if by outputting an appropriate number of space or return characters. *x* and *y* are like the values of :read-cursorpos and *units* is the same as the *units* argument to :read-cursorpos.

Any stream which supports this operation should support :read-cursorpos as well, but it need not support :set-cursorpos.

Moving the cursor with :increment-cursorpos differs from moving it to the same place with :set-cursorpos in that this operation is thought of as doing output and :set-cursorpos is not. For example, moving a window's cursor down with :increment-cursorpos when it is near the bottom to begin with will wrap around, possibly doing a **MORE**. :set-cursorpos, by comparison, cannot move the cursor "down" if it is at the bottom of the window; it can move the cursor explicitly to the top of the window, but then no **MORE** will happen.

Some streams, such as those created by with-output-to-string, cannot implement arbitrary cursor motion, but do implement this operation.

:set-cursorpos *x y* &optional (*units* ':pixel) *Operation on streams*

This operation is supported by the same streams that support :read-cursorpos. It sets the position of the cursor. *x* and *y* are like the values of :read-cursorpos and *units* is the same as the *units* argument to :read-cursorpos.

:clear-screen *Operation on streams*

Erases the screen area on which this stream displays. Non-window streams don't support this operation.

There are many other special-purpose stream operations for graphics. They are not documented here, but in the window-system documentation. No claim that the above operations are the most useful subset should be implied.

21.5.6 Operations for Efficient Grinding

grindef runs much more efficiently on streams that implement the :untyo-mark and :untyo operations.

:untyo-mark *Operation on streams*

This is used by the grinder (see page 426) if the output stream supports it. It takes no arguments. The stream should return some object that indicates how far output has gotten up to in the stream.

:untyo *mark* *Operation on streams*

This is used by the grinder (see page 426) in conjunction with :untyo-mark. It takes one argument, which is something returned by the :untyo-mark operation of the stream. The stream should back up output to the point at which the object was returned.

21.5.7 Random Access File Operations

The following operations are implemented only by streams to random-access devices, principally files.

:read-pointer *Operation on streams*

Returns the current position within the file, in characters (bytes in fixnum mode). For text files on PDP-10 file servers, this is the number of Lisp Machine characters, not PDP-10 characters. The numbers are different because of character-set translation.

:set-pointer *new-pointer* *Operation on streams*

Sets the reading position within the file to *new-pointer* (bytes in fixnum mode). For text files on PDP-10 file servers, this will not do anything reasonable unless *new-pointer* is 0, because of character-set translation. Some file systems support this operation for input streams only.

:rewind *Operation on streams*

This operation is obsolete. It is the same as :set-pointer with argument zero.

21.5.8 Buffered Stream Operations

:clear-input *Operation on streams*

This operation discards any buffered input the stream may have. It does nothing on streams for which it is not meaningful.

:clear-output *Operation on streams*

This operation discards any buffered output the stream may have. It does nothing on streams for which it is not meaningful.

:force-output *Operation on streams*

This is for output streams to buffered asynchronous devices, such as the Chaosnet. :force-output causes any buffered output to be sent to the device. It does not wait for it to complete; use :finish for that. If a stream supports :force-output, then :tyo, :string-out, and :line-out may have no visible effect until a :force-output is done.

This operation does nothing on streams for which it is not meaningful.

:finish *Operation on streams*

This is for output streams to buffered asynchronous devices, such as the Chaosnet. :finish does a :force-output, then waits until the currently pending I/O operation has been completed.

This operation does nothing on streams for which it is not meaningful.

The following operations are implemented only by buffered input streams. They allow increased efficiency by making the stream's internal buffer available to the user.

:read-input-buffer &optional *eof* *Operation on streams*

Returns three values: a buffer array, the index in that array of the next input byte, and the index in that array just past the last available input byte. These values are similar to the *string*, *start*, *end* arguments taken by many functions and stream operations. If the end of the file has been reached and no input bytes are available, the stream returns nil or signals an error, based on the *eof* argument, just like the :tyi message. After reading as many bytes from the array as you care to, you must send the :advance-input-buffer message.

:get-input-buffer &optional *eof* *Operation on streams*

This is an obsolete operation similar to :read-input-buffer. The only difference is that the third value is the number of significant elements in the buffer-array, rather than a final index. If found in programs, it should be replaced with :read-input-buffer.

:advance-input-buffer &optional *new-pointer* *Operation on streams*

If *new-pointer* is non-nil, it is the index in the buffer array of the next byte to be read. If *new-pointer* is nil, the entire buffer has been used up.

21.5.9 Standard Streams

There are several variables whose values are streams used by many functions in the Lisp system. These variables and their uses are listed here. By convention, variables that are expected to hold a stream capable of input have names ending with -input, and similarly for output. Those expected to hold a bidirectional stream have names ending with -io.

standard-input *Variable*

In the normal Lisp top-level loop, input is read from standard-input (that is, whatever stream is the value of standard-input). Many input functions, including tyi and read, take a stream argument that defaults to standard-input.

standard-output *Variable*

In the normal Lisp top-level loop, output is sent to standard-output (that is, whatever stream is the value of standard-output). Many output functions, including tyo and print, take a stream argument that defaults to standard-output.

error-output *Variable*

The value of error-output is a stream to which noninteractive error or warning messages should be sent. Normally this is the same as standard-output, but standard-output might be bound to a file and error-output left going to the terminal.

[This seems not be used by all the things that ought to use it.]

debug-io *Variable*

The value of error-handler-io, is used for all input and output by the error handler. Normally this is the same as standard-output.

query-io *Variable*

The value of query-io is a stream that should be used when asking questions of the user. The question should be output to this stream, and the answer read from it. The reason for this is that when the normal input to a program may be coming from a file, questions such as "Do you really want to delete all of the files in your directory??" should be sent directly to the user, and the answer should come from the user, not from the data file. query-io is used by fquery and related functions; see page 621.

terminal-io *Variable*

The value of terminal-io is the stream that connects to the user's console. In an "interactive" program, it will be the window from which the program is being run; I/O on this stream will read from the keyboard and display on the TV. However, in a "background" program that does not normally talk to the user, terminal-io defaults to a stream that does not ever expect to be used. If it is used, perhaps by an error printout, it turns into a "background" window and requests the user's attention.

trace-output *Variable*

The value of trace-output is the stream on which the trace function prints its output.

standard-input, standard-output, error-output, debug-io, trace-output, and query-io are initially bound to synonym streams that pass all operations on to the stream that is the value of terminal-io. Thus any operations performed on those streams will go to the TV terminal.

Most user program should not change the value of terminal-io. A program which wants (for example) to divert output to a file should do so by binding the value of standard-output; that way queries on query-io, debugging on debug-io and error messages sent to error-output can still get to the user by going through terminal-io, which is usually what is desired.

21.5.10 Obtaining Streams to Use

One important class of streams is windows. Each window can be used as a stream. Output is displayed on the window and input comes from the keyboard. A window is created using tv:make-window. Simple programs use windows implicitly through terminal-io and the other standard stream variables.

Also important are *file streams*, which are produced by the function open (see page 432). These read or write the contents of a file.

Chaosnet streams are made from chaosnet connections. Data output to the stream goes out over the network; data coming in over the network is available as input from the stream. File streams that deal with chaosnet file servers are very similar to chaosnet streams, but chaosnet streams can be used for many purposes other than file access.

String streams read or write the contents of a string. They are made by with-output-to-string or with-input-from-string.

Editor buffer streams read or write the contents of an editor buffer.

The *null stream* may be passed to a program that asks for a stream as an argument. It returns immediate end of file if used for input and throws away any output. The null stream is the symbol si:null-stream. This is to say, you do not call that function to get a stream or use the symbol's value as the stream; *the symbol itself* is the object that is the stream.

The *cold-load stream* is able to do i/o to the keyboard and screen without using the window system. It is what is used to by the error handler if you type Terminal Call to handle a background error that the window system cannot deal with. It is called the cold-load stream because it is what is used during system bootstrapping, before the window system has been loaded.

si:null-stream *operation* &rest *arguments*

This function is the null stream. Like any stream, it supports various operations. Output operations are ignored and input operations report end of file immediately, with no data.

si:cold-load-stream *Variable*

The value of this variable is the one and only cold-load stream.

with-open-stream (*variable expression*) *body...* *Special Form*

body is executed with *variable* bound to the value of *expression*, which ought to be a stream. On exit, whether normal or by throwing, a :close message with argument :abort is sent to the stream.

This is a generalization of with-open-file, which is equivalent to using with-open-stream with a call to open as the *expression*.

with-open-stream-case (*variable expression*) *clauses...* *Special Form*

Like with-open-stream as far as opening and closing the stream are concerned, but instead of a simple body, it has clauses like those of a condition-case that say what to do if *expression* does or does not get an error. See with-open-file-case, page 432.

make-syn-stream *symbol-or-locative*

make-syn-stream creates and returns a "synonym stream" (syn for short). Any operations sent to this stream will be redirected to the stream that is the value of the argument (if it is a symbol) or the contents of it (if it is a locative).

A synonym stream made from a symbol is actually a symbol named *symbol*-syn-stream whose function definition is *symbol*, with a property that declares it to be a legitimate stream. The generated symbol is interned in the same package as *symbol*.

A synonym stream made from a locative is a closure.

make-broadcast-stream &rest *streams*

Returns a stream that only works in the output direction. Any output sent to this stream will be sent to all of the streams given. The :which-operations is the intersection of the :which-operations of all of the streams. The value(s) returned by a stream operation are the values returned by the last stream in *streams*.

zwei:interval-stream *interval-or-from-bp* &optional *to-bp in-order-p hack-fonts*

Returns a bidirectional stream that reads or writes all or part of an editor buffer. Note that editor buffer streams can also be obtained from open by using a pathname whose host is ED, ED-BUFFER or ED-FILE (see section 22.7.6, page 479).

The first three arguments specify the buffer or portion to be read or written. Either the first argument is an *interval* (a buffer is one kind of interval), and all the text of that interval is read or written, or the first two arguments are two buffer pointers delimiting the range to be read or written. The third argument is used only in the latter case; if non-nil, it tells the function to assume that the second buffer pointer comes later in the buffer than the first and not to take the time to verify the assumption.

The stream has only one pointer inside it, used for both input and output. As you do input, the pointer advances through the text. When you do output, it is inserted in the buffer at the place where the pointer has reached. The pointer starts at the beginning of the specified range.

hack-fonts tells what to do about fonts.

If *hack-fonts* is t, then the character ε is recognized as special when you output to the stream; sequences such as ε2 are interpreted as font-changes. They do not get inserted into the buffer; instead, they change the font in which following output will be inserted. On input, font change sequences are inserted to indicate faithfully what was in the buffer.

If *hack-fonts* is :tyo, then you are expected to read and write 16-bit characters containing font numbers.

If *hack-fonts* is nil, then all output is inserted in font zero and font information is discarded in the input you receive. This is the best mode to use if you are evaluating the contents of an editor buffer.

si:with-help-stream (*stream options...*) *body...* *Special Form*

Executes the *body* with the variable *stream* bound to a suitable stream for printing a large help message. If standard-output is a window, then *stream* is also a window; a temporary window which fills the screen. Otherwise, *stream* is just the same as standard-output.

The purpose of this is to spare the user the need to read a large help printout in a small window, or have his data overwritten by it permanently. This is the mechanism used if you type the Help key while in the rubout handler.

Important note: the body gets turned into a lambda-expression with the stream and the width and height variables bound appropriately; this means that variables from the surrounding program that are to be accessed by the body must be special.

options is a list of alternating keywords and values.

:label	The value (which is evaluated) is used as the label of the temporary window, if one is used.
:width	The value, which is not evaluated, is a symbol. While *body* is executed, this symbol is bound to the width, in characters, available for the message.
:height	The value is a symbol, like the value after :width, and it is bound to the height in lines of the area available for the help message.
:superior	The value, which is evaluated, specifies the original stream to use in deciding where to print the help message. This overrides the use of standard-output.

21.5.11 Implementing Streams

There are two ways to implement a stream: using **defun** and using flavors.

Using flavors is best when you can take advantage of the predefined stream mixins, including those which perform buffering, or when you wish to define several similar kinds of streams that can inherit methods from each other.

defun may have an advantage if you are dividing operations into broad groups and handling them by passing them off to one or more other streams. In this case, the automatic operation decoding provided by flavors may get in the way. A number of streams in the system are implemented using **defun** (or using **defselect**, which is nearly the same thing) for historical reasons. It isn't yet clear whether there is any reason not to convert most of them to use flavors.

If you use **defun**, you can use the *stream default handler* to implement some of the standard operations for you in a default manner. If you use flavors, there are predefined mixins to do this for you.

A few streams are individual objects, one of a kind. For example, there is only one null stream, and no need for more, since two null streams would behave identically. But most streams are elements of a general class. For example, there can be many file streams for different files, even though all behave the same way. There can also be multiple streams reading from different points in the same file.

If you implement a class of streams with **defun**, then the actual streams will be closures of the function you define, made with **closure**.

If you use flavors to implement the streams, having a class of similar streams comes naturally: each instance of the flavor is a stream, and the instance variables distinguish one stream of the class from another.

21.5.12 Implementing Streams with Flavors

To define a stream using flavors, define a flavor which incorporates the appropriate predefined stream flavor, and then redefine those operations which are peculiar to your own type of stream.

Flavors for defining unbuffered streams:

si:stream *Flavor*

This flavor provides default definitions for a few standard operations such as **:direction** and **:characters**. Usually you do not have to mention this explicitly; instead you use the higher level flavors below, which are built on this one.

si:input-stream *Flavor*

This flavor provides default definitions of all the mandatory input operations except **:tyi** and **:untyi**, in terms of those two. You can make a simple non-character input stream by defining a flavor incorporating this one and giving it methods for **:tyi** and **:untyi**.

si:output-stream *Flavor*

This flavor provides default definitions of all the mandatory output operations except :tyo, in terms of :tyo. All you need to do to define a simple unbuffered non-character output stream is to define a flavor incorporating this one and give it a method for the :tyo operation.

si:bidirectional-stream *Flavor*

This is a combination of si:input-stream and si:output-stream. It defines :direction to return :bidirectional. To define a simple unbuffered non-character bidirectional stream, build on this flavor and define :tyi, :untyi and :tyo.

The unbuffered streams implement operations such as :string-out and :string-in by repeated use of :tyo or :tyi.

For greater efficiency, if the stream's data is available in blocks, it is better to define a buffered stream. You start with the predefined buffered stream flavors, which define :tyi or :tyo themselves and manage the buffers for you. You must provide other operations that the system uses to obtain the next input buffer or to write or discard an output buffer.

Flavors for defining buffered streams:

si:buffered-input-stream *Flavor*

This flavor is the basis for a non-character buffered input stream. It defines :tyi as well as all the other standard input operations, but you must define the two operations :next-input-buffer and :discard-input-buffer, which the buffer management routines use.

:next-input-buffer *Operation on* si:buffered-input-stream

In a buffered input stream, this operation is used as a subroutine of the standard input operations, such as :tyi, to get the next bufferful of input data. It should return three values: an array containing the data, a starting index in the array, and an ending index. For example, in a chaosnet stream, this operation would get the next packet of input data and return pointers delimiting the actual data in the packet.

:discard-input-buffer *buffer-array* *Operation on* si:buffered-input-stream

In a buffered input stream, this operation is used as a subroutine of the standard input operations such as :tyi. It says that the buffer management routines have used or thrown away all the input in a buffer, and the buffer is no longer needed.

In a chaosnet stream, this operation would return the buffer, a packet, to the pool of free packets.

si:buffered-output-stream *Flavor*

This flavor is the basis for a non-character buffered output stream. It defines :tyo as well as all the other standard output operations, but you must define the operations :new-input-buffer, :send-input-buffer and :discard-output-buffer, which the buffer management routines use.

:new-output-buffer *Operation on* si:buffered-output-stream

In a buffered output stream, this operation is used as a subroutine of the standard output operations, such as :tyo, to get an empty buffer for storing more output data. How the buffer is obtained depends on the kind of stream, but in any case this operation should return an array (the buffer), a starting index, and an ending index. The two indices delimit the part of the array that is to be used as a buffer.

For example, a chaosnet stream would get a packet from the free pool and return indices delimiting the part of the packet array which can hold data bytes.

:send-output-buffer *buffer-array ending-index* *Operation on* si:buffered-output-stream

In a buffered output stream, this operation is used as a subroutine of the standard output operations, such as :tyo, to send the data in a buffer that has been completely or partially filled.

ending-index is the first index in the buffer that has not actually been stored. This may not be the same as the ending index that was returned by the :new-output-buffer operation that was used to obtain this buffer; if a :force-output is done, *ending-index* will indicate how much of the buffer was full at the time.

The method for this operation should process the buffer's data and, if necessary, return the buffer to a free pool.

:discard-output-buffer *buffer-array* *Operation on* si:buffered-output-stream

In a buffered output stream, this operation is used as a subroutine of the standard output operations, such as :clear-output, to free an output buffer and say that the data in it should be ignored.

It should simply return *buffer-array* to a free pool, if appropriate.

Some buffered output streams simply have one buffer array which they use over and over. For such streams, :new-output-buffer can simply return that particular array each time; :send-output-buffer and :discard-output-buffer to not have to do anything about returning the buffer to a free pool. In fact, :discard-output-buffer can probably do nothing.

si:buffered-stream *Flavor*

This is a combination of si:buffered-input-stream and si:buffered-output-stream, used to make a buffered bidirectional stream. The input and output buffering are completely independent of each other. You must define all five of the low level operations: :new-input-buffer, :send-input-buffer and :discard-output-buffer for output, and :next-input-buffer and :discard-input-buffer for input.

The data in most streams are characters. Character streams should support either :line-in or :line-out in addition to the other standard operations.

si:unbuffered-line-input-stream *Flavor*

This flavor is the basis for unbuffered character input streams. You need only define :tyi and :untyi.

si:line-output-stream-mixin *Flavor*

To make an unbuffered character output stream, mix this flavor into the one you define, together with si:output-stream. In addition, you must define :tyo, as for unbuffered non-character streams.

si:buffered-input-character-stream *Flavor*

This is used just like si:buffered-input-stream, but it also provides the :line-in operation and makes :characters return t.

si:buffered-output-character-stream *Flavor*

This is used just like si:buffered-output-stream, but it also provides the :line-out operation and makes :characters return t.

si:buffered-character-stream *Flavor*

This is used just like si:buffered-stream, but it also provides the :line-in and :line-out operations and makes :characters return t.

To make an unbuffered random-access stream, you need only define the :read-pointer and :set-pointer operations as appropriate. Since you provide the :tyi or :tyo handler yourself, the system cannot help you.

In a buffered random-access stream, the random access operations must interact with the buffer management. The system provides for this.

si:input-pointer-remembering-mixin *Flavor*

Incorporate this into a buffered input stream to support random access. This flavor defines the :read-pointer and :set-pointer operations. If you wish :set-pointer to work, you must provide a definition for the :set-buffer-pointer operation. You need not do so if you wish to support only :read-pointer.

:set-buffer-pointer *new-pointer* *Operation on* si:input-pointer-remembering-mixin

You must define this operation if you use si:input-pointer-remembering-mixin and want the :set-pointer operation to work.

This operation should arrange for the next :next-input-buffer operation to provide a bufferful of data that includes the specified character or byte position somewhere inside it.

The value returned should be the file pointer corresponding to the first character or byte of that next bufferful.

si:output-pointer-remembering-mixin *Flavor*

Incorporate this into a buffered output stream to support random access. This mixin defines the :read-pointer and :set-pointer operations. If you wish :set-pointer to work, you must provide definitions for the :set-buffer-pointer and :get-old-data operations. You need not do so if you wish to support only :read-pointer.

:set-buffer-pointer *new-pointer* *Operation on* si:output-pointer-remembering-mixin

This is the same as in si:input-pointer-remembering-mixin.

:get-old-data *buffer-array lower-output-limit* *Operation on* si:output-pointer-remembering-mixin

The buffer management routines perform this operation when you do a :set-pointer that outside the range of pointers that fit in the current output buffer. They first send the old buffer, then do :set-buffer-pointer as described above to say where in the file the next output buffer should come, then do :new-output-buffer to get the new buffer. Then the :get-old-data operation is performed.

It should fill current buffer (*buffer-array*) with the *old* contents of the file at the corresponding addresses, so that when the buffer is eventually written, any bytes skipped over by random access will retain their old values.

The instance variable si:stream-output-lower-limit is the starting index in the buffer of the part that is supposed to be used for output. si:stream-output-limit is the ending index. The instance variable si:output-pointer-base is the file pointer corresponding to the starting index in the buffer.

si:file-stream-mixin *Flavor*

Incorporate this mixin together with si:stream to make a *file probe stream*, which cannot do input or output but records the answers to an enquiry about a file. You should specify the init option :pathname when you instantiate the flavor.

You must provide definitions for the :plist and :truename operations; in terms of them, this mixin will define the operations :get, :creation-date, and :info.

si:file-input-stream-mixin *Flavor*

Incorporate this mixin into input streams that are used to read files. You should specify the file's pathname with the :pathname init option when you instantiate the flavor.

In addition to the services and requirements of si:file-stream-mixin, this mixin takes care of mentioning the file in the who-line. It also includes si:input-pointer-remembering-mixin so that the :read-pointer operation, at least, will be available.

si:file-output-stream-mixin *Flavor*

This is the analogue of si:file-input-stream-mixin for output streams.

21.5.13 Implementing Streams Without Flavors

You do not need to use flavors to implement a stream. Any object that can be used as a function, and decodes its first argument appropriately as an operation name, will serve as a stream. Although in practice using flavors is as easy as any other way, it is educational to see how to define streams "from scratch".

We could begin to define a simple output stream, which accepts characters and conses them onto a list, as follows:

```
(defvar the-list nil)

(defun list-output-stream (op &optional arg1 &rest rest)
    (selectq op
        (:tyo
         (setq the-list (cons arg1 the-list)))
        (:which-operations '(:tyo))))
```

This is an output stream, and so it supports the :tyo operation. All streams must support :which-operations.

The lambda-list for a stream defined with a defun must always have one required parameter (*op*), one optional parameter (*arg1*), and a rest parameter (*rest*).

This definition is not satisfactory, however. It handles :tyo properly, but it does not handle :string-out, :direction, :send-if-handles, and other standard operations.

The function stream-default-handler exists to spare us the trouble of defining all those operations from scratch in simple streams like this. By adding one additional clause, we let the default handler take care of all other operations, if it can.

```
(defun list-output-stream (op &optional arg1 &rest rest)
    (selectq op
        (:tyo
         (setq the-list (cons arg1 the-list)))
        (:which-operations '(:tyo))
        (otherwise
          (stream-default-handler #'list-output-stream
                                  op arg1 rest))))
```

If the operation is not one that the stream understands (e.g. :string-out), it calls stream-default-handler. Note how the rest argument is passed to it. This is why the argument list must look the way it does. stream-default-handler can be thought of as a restricted analogue of flavor inheritance.

If we want to have only one stream of this sort, the symbol list-output-stream can be used as the stream. The data output to it will appear in the global value of the-list. One more step is required, though:

```
    (defprop list-output-stream t si:io-stream-p)
```

This tells certain functions including read to treat the symbol list-output-stream as a stream rather than as an end-of-file option.

If we wish to be able to create any number of list output streams, each accumulating its own list, we must use closures:

```
(defvar the-stream)
(defvar the-list)

(defun list-output-stream (op &optional arg1 &rest rest)
    (selectq op
        (:tyo
         (push arg1 the-list)))
        (:withdrawal (prog1 the-list (setq the-list nil)))
        (:which-operations '(:tyo :withdrawal))
        (otherwise
          (stream-default-handler the-stream
                                  op arg1 rest))))

(defun make-list-output-stream ()
  (let ((the-stream the-list))
    (setq the-stream
          (closure '(the-stream the-list)
                   'list-output-stream))))
```

We have added a new operation :withdrawal that can be used to find out what data has been accumulated by a stream. This is necessary because we can no longer simply look at or set the global value of the-list; that is not the same as the value closed into the stream.

In addition, we have a new variable the-stream which allows the function list-output-stream to know which stream it is serving at any time. This variable is passed to stream-default-handler so that when it simulates :string-out by means of :tyo, it can do the :tyo's to the same stream that the :string-out was done to.

The same stream could be defined with defselect instead of defun. It actually makes only a small difference. The defun for list-output-stream could be replaced with this code:

```
(defselect (list-output-stream list-output-d-h)
  (:tyo (arg1)
    (push arg1 the-list))
  (:withdrawal ()
    (prog1 the-list (setq the-list nil))))

(defun list-output-d-h (op &optional arg1 &rest rest)
  (stream-default-handler the-stream op arg1 rest))
```

defselect takes care of decoding the operations, provides a definition for :which-operations, and allows you to write a separate lambda list for each operation.

By comparison, the same stream defined using flavors looks like this:

```
(defflavor list-output-stream ((the-list nil))
           (si:line-output-stream-mixin si:output-stream))

(defmethod (list-output-stream :tyo) (character)
  (push character the-list))

(defmethod (list-outut-stream :withdrawal) ()
  (prog1 the-list (setq the-list nil)))

(defun make-list-output-stream ()
  (make-instance 'list-output-stream))
```

Here is a simple input stream, which generates successive characters of a list.

```
(defvar the-list)       ;Put your input list here
(defvar the-stream)
(defvar untyied-char nil)

(defun list-input-stream (op &optional arg1 &rest rest)
  (selectq op
    (:tyi
     (cond ((not (null untyied-char))
             (prog1 untyied-char (setq untyied-char nil)))
            ((null the-list)
             (and arg1 (error arg1)))
            (t (pop the-list))))
    (:untyi
     (setq untyied-char arg1))
    (:which-operations '(:tyi :untyi))
    (otherwise
      (stream-default-handler the-stream
                              op arg1 rest))))

(defun make-list-input-stream (the-list)
  (let (the-stream untyied-char)
    (setq the-stream
          (closure '(the-list the-stream untyied-char)
                   'list-input-stream))))
```

The important things to note are that :untyi must be supported, and that the stream must check for having reached the end of the information and do the right thing with the argument to the :tyi operation.

stream-default-handler *stream op arg1 rest*

stream-default-handler tries to handle the *op* operation on *stream*, given arguments of *arg1* and the elements of *rest*. The exact action taken for each of the defined operations is explained with the documentation on that operation, above.

21.6 Formatted Output

There are two ways of doing general formatted output. One is the function **format**. The other is the **output** subsystem. **format** uses a control string written in a special format specifier language to control the output format. **format:output** provides Lisp functions to do output in particular formats.

For simple tasks in which only the most basic format specifiers are needed, **format** is easy to use and has the advantage of brevity. For more complicated tasks, the format specifier language becomes obscure and hard to read. Then **format:output** becomes advantageous because it works with ordinary Lisp control constructs.

For formatting Lisp code (as opposed to text and tables), there is the grinder (see page 426).

21.6.1 The Format Function

format *destination control-string* &rest *args*

format is used to produce formatted output. **format** outputs the characters of *control-string*, except that a tilde ("~") introduces a directive. The character after the tilde, possibly preceded by prefix parameters and modifiers, specifies what kind of formatting is desired. Most directives use one or more elements of *args* to create their output; the typical directive puts the next element of *args* into the output, formatted in some special way.

The output is sent to *destination*. If *destination* is **nil**, a string is created which contains the output; this string is returned as the value of the call to **format**. In all other cases **format** returns no interesting value (generally it returns **nil**). If *destination* is a stream, the output is sent to it. If *destination* is **t**, the output is sent to **standard-output**. If *destination* is a string with an array-leader, such as would be acceptable to **string-nconc** (see page 147), the output is added to the end of that string.

A directive consists of a tilde, optional prefix parameters separated by commas, optional colon (":") and atsign ("@") modifiers, and a single character indicating what kind of directive this is. The alphabetic case of the character is ignored. The prefix parameters are generally decimal numbers. Examples of control strings:

```
"~S"          ; This is an S directive with no parameters.
"~3,4:@s"     ; This is an S directive with two parameters, 3 and 4,
              ;   and both the colon and atsign flags.
"~,4S"        ; The first prefix parameter is omitted and takes
              ;   on its default value, while the second is 4.
```

format includes some extremely complicated and specialized features. It is not necessary to understand all or even most of its features to use **format** efficiently. The beginner should skip over anything in the following documentation that is not immediately useful or clear. The more sophisticated features are there for the convenience of programs with complicated formatting requirements.

Sometimes a prefix parameter is used to specify a character, for instance the padding character in a right- or left-justifying operation. In this case a single quote (" ' ") followed by the desired character may be used as a prefix parameter, so that you don't have to know the decimal numeric values of characters in the character set. For example, you can use
`"~5,'0d"` instead of `"~5,48d"`
to print a decimal number in five columns with leading zeros.

In place of a prefix parameter to a directive, you can put the letter V, which takes an argument from *args* as a parameter to the directive. Normally this should be a number but it doesn't really have to be. This feature allows variable column-widths and the like. Also, you can use the character # in place of a parameter; it represents the number of arguments remaining to be processed.

Here are some relatively simple examples to give you the general flavor of how **format** is used.

```
(format nil "foo") => "foo"
(setq x 5)
(format nil "The answer is ~D." x) => "The answer is 5."
(format nil "The answer is ~3D." x) => "The answer is   5."
(setq y "elephant")
(format nil "Look at the ~A!" y) => "Look at the elephant!"
(format nil "The character ~:@C is strange." 1003)
      => "The character Meta-β (Top-X) is strange."
(setq n 3)
(format nil "~D item~:P found." n) => "3 items found."
(format nil "~R dog~:[s are~; is~] here." n (= n 1))
      => "three dogs are here."
(format nil "~R dog~:*~[~1; is~:;s are~] here." n)
      => "three dogs are here."
(format nil "Here ~[~1;is~:;are~] ~:*~R pupp~:@P." n)
      => "Here are three puppies."
```

The directives will now be described. *arg* will be used to refer to the next argument from *args*.

~A *arg*, any Lisp object, is printed without slashification (as by **princ**). ~:A prints **()** if *arg* is **nil**; this is useful when printing something that is always supposed to be a list. ~*n*A inserts spaces on the right, if necessary, to make the column width at least *n*. The @ modifier causes the spaces to be inserted on the left rather than the right. ~*mincol,colinc,minpad,padchar*A is the full form of ~A, which allows elaborate control of the padding. The string is padded on the right with at least *minpad* copies of *padchar*; padding characters are then inserted *colinc* characters at a time until the total width is at least *mincol*. The defaults are 0 for *mincol* and *minpad*, 1 for *colinc*, and space for *padchar*.

~S This is just like ~A, but *arg* is printed *with* slashification (as by **prin1** rather than **princ**).

~D *arg*, a number, is printed as a decimal integer. Unlike **print**, ~D will never put a decimal point after the number. ~*n*D uses a column width of *n*; spaces are inserted on the left if the number requires less than *n* columns for its digits and sign. If the number doesn't fit in *n* columns, additional columns are used as needed. ~*n*,*m*D uses *m* as the pad character instead of space. If *arg* is not a number, it is printed in ~A format and decimal base. The @ modifier causes the number's sign to be printed always; the default is only to print it if the number is negative. The : modifier causes commas to be printed between groups of three digits; the third prefix parameter may be used to change the character used as the comma. Thus the most general form of ~D is ~*mincol*,*padchar*,*commachar*D.

~O This is just like ~D but prints in octal instead of decimal.

~F *arg* is printed in floating point. ~*n*F rounds *arg* to a precision of *n* digits. The minimum value of *n* is 2, since a decimal point is always printed. If the magnitude of *arg* is too large or too small, it is printed in exponential notation. If *arg* is not a number, it is printed in ~A format. Note that the prefix parameter *n* is not *mincol*; it is the number of digits of precision desired. Examples:

```
(format nil "~2F" 5)  =>  "5.0"
(format nil "~4F" 5)  =>  "5.0"
(format nil "~4F" 1.5) => "1.5"
(format nil "~4F" 3.14159265)  =>  "3.142"
(format nil "~3F" 1e10)  =>  "1.0e10"
```

~E *arg* is printed in exponential notation. This is identical to ~F, including the use of a prefix parameter to specify the number of digits, except that the number is always printed with a trailing exponent, even if it is within a reasonable range.

~$ ~*rdig*,*ldig*,*field*,*padchar*$ prints *arg*, a flonum, with exactly *rdig* digits after the decimal point. The default for *rdig* is 2, which is convenient for printing amounts of money. At least *ldig* digits will be printed preceding the decimal point; leading zeros will be printed if there would be fewer than *ldig*. The default for *ldig* is 1. The number is right justified in a field *field* columns long, padded out with *padchar*. The colon modifier means that the sign character is to be at the beginning of the field, before the padding, rather than just to the left of the number. The atsign modifier says that the sign character should always be output.

If *arg* is not a number, or is unreasonably large, it will be printed in ~*field*,,,*padchar*@A format; i.e. it will be **princ**'ed right-justified in the specified field width.

~C (**character** *arg*) is put in the output. *arg* is treated as a keyboard character (see page 363), thus it may contain extra control-bits. These are printed first by representing them with abbreviated prefixes: "C-" for **Control**, "M-" for **Meta**, "H-" for **Hyper**, and "S-" for **Super**.

With the colon flag (~:C), the names of the control bits are spelled out (e.g. "Control-Meta-F") and non-printing characters are represented by their names (e.g. "Return") rather than being output as themselves.

With both colon and atsign (~:@C), the colon-only format is printed, and then if the character requires the Top or Greek (Front) shift key(s) to type it, this fact is mentioned (e.g. "∀ (Top-U)"). This is the format used for telling the user about a key he is expected to type, for instance in prompt messages.

For all three of these formats, if the character is not a keyboard character but a mouse "character", it is printed as Mouse-, the name of the button, "-", and the number of clicks.

With just an atsign (~@C), the character is printed in such a way that the Lisp reader can understand it, using "#/" or "#\".

~% Outputs a carriage return. ~*n*% outputs *n* carriage returns. No argument is used. Simply putting a carriage return in the control string would work, but ~% is usually used because it makes the control string look nicer in the Lisp source program.

~& The :fresh-line operation is performed on the output stream. Unless the stream knows that it is already at the front of a line, this outputs a carriage return. ~*n*& does a :fresh-line operation and then outputs *n*-1 carriage returns.

~| Outputs a page separator character (#\page). ~*n*| does this *n* times. With a : modifier, if the output stream supports the :clear-screen operation this directive clears the screen, otherwise it outputs page separator character(s) as if no : modifier were present. | is vertical bar, not capital I.

~X Outputs a space. ~*n*X outputs *n* spaces.

~~ Outputs a tilde. ~*n*~ outputs *n* tildes.

~<CR> Tilde immediately followed by a carriage return ignores the carriage return and any whitespace at the beginning of the next line. With a :, the whitespace is left in place. With an @, the carriage return is left in place. This directive is typically used when a format control string is too long to fit nicely into one line of the program.

~* *arg* is ignored. ~*n** ignores the next *n* arguments. ~:* "ignores backwards"; that is, it backs up in the list of arguments so that the argument last processed will be processed again. ~*n*:* backs up *n* arguments. When within a ~{ construct (see below), the ignoring (in either direction) is relative to the list of arguments being processed by the iteration.

~P If *arg* is not 1, a lower-case "s" is printed. ("P" is for "plural".) ~:P does the same thing, after doing a ~:*; that is, it prints a lower-case s if the *last* argument was not 1. ~@P prints "y" if the argument is 1, or "ies" if it is not. ~:@P does the same thing, but backs up first.

~T Spaces over to a given column. ~*n*,*m*T will output sufficient spaces to move the cursor to column *n*. If the cursor is already past column *n*, it will output spaces to move it to column *n*+*mk*, for the smallest integer value *k* possible. *n* and *m* default to 1. Without the colon flag, *n* and *m* are in units of characters; with it, they are in units of pixels. Note: this operation works properly *only* on streams that support the

:read-cursorpos and :increment-cursorpos stream operations (see page 396). On other streams, any ~T operation will simply output two spaces. When **format** is creating a string, ~T will work, assuming that the first character in the string is at the left margin.

~R ~R prints *arg* as a cardinal English number, e.g. four. ~:R prints *arg* as an ordinal number, e.g. fourth. ~@R prints *arg* as a Roman numeral, e.g. IV. ~:@R prints *arg* as an old Roman numeral, e.g. IIII.

~*n*R prints *arg* in radix *n*. The flags and any remaining parameters are used as for the ~D directive. Indeed, ~D is the same as ~10R. The full form here is therefore ~*radix*,*mincol*,*padchar*,*commachar*R.

~*n* G "Goes to" the *n*th argument. ~0G goes back to the first argument in *args*. Directives after a ~*n*G will take sequential arguments after the one gone to. When within a ~{ construct, the "goto" is relative to the list of arguments being processed by the iteration. This is an "absolute goto"; for a "relative goto", see ~*.

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

This is a set of alternative control strings. The alternatives (called *clauses*) are separated by ~; and the construct is terminated by ~]. For example,

```
"~[Siamese ~;Manx ~;Persian ~;Tortoise-Shell ~
   ~;Tiger ~;Yu-Shiang ~]kitty"
```

The *arg*th alternative is selected; 0 selects the first. If a prefix parameter is given (i.e. ~*n*[), then the parameter is used instead of an argument (this is useful only if the parameter is "#"). If *arg* is out of range no alternative is selected. After the selected alternative has been processed, the control string continues after the ~].

~[*str0*~;*str1*~;...~;*strn*~:;*default*~] has a default case. If the *last* ~; used to separate clauses is instead ~:;, then the last clause is an "else" clause, which is performed if no other clause is selected. For example,

```
"~[Siamese ~;Manx ~;Persian ~;Tiger ~
   ~;Yu-Shiang ~:;Bad ~] kitty"
```

~[~*tag00*,*tag01*,...;*str0*~*tag10*,*tag11*,...;*str1*...~] allows the clauses to have explicit tags. The parameters to each ~; are numeric tags for the clause which follows it. That clause is processed which has a tag matching the argument. If ~*a1*,*a2*,*b1*,*b2*,...:; (note the colon) is used, then the following clause is tagged not by single values but by ranges of values *a1* through *a2* (inclusive), *b1* through *b2*, etc. ~:; with no parameters may be used at the end to denote a default clause. For example,

```
"~[~'+,'-,'*,'//;operator ~'A,'Z,'a,'z:;letter ~
   ~'0,'9:;digit ~:;other ~]"
```

~:[*false*~;*true*~] selects the *false* control string if *arg* is nil, and selects the *true* control string otherwise.

~@[*true*~] tests the argument. If it is not nil, then the argument is not used up, but is the next one to be processed, and the one clause is processed. If it is nil, then the argument is used up, and the clause is not processed. For example,

```
(setq prinlevel nil prinlength 5)
(format nil "~@[ PRINLEVEL=~D~]~@[ PRINLENGTH=~D~]"
           prinlevel prinlength)
   =>  " PRINLENGTH=5"
```

The combination of ~[and # is useful, for example, for dealing with English conventions for printing lists:

```
(setq foo "Items:~#[ none~; ~S~; ~S and ~
          ~S~:;~@{~#[~1; and~] ~S~^,~}~].")
(format nil foo)
        =>  "Items: none."
(format nil foo 'foo)
        =>  "Items: FOO."
(format nil foo 'foo 'bar)
        =>  "Items: FOO and BAR."
(format nil foo 'foo 'bar 'baz)
        =>  "Items: FOO, BAR, and BAZ."
(format nil foo 'foo 'bar 'baz 'quux)
        =>  "Items: FOO, BAR, BAZ, and QUUX."
```

~; Separates clauses in ~[and ~< constructions. It is undefined elsewhere.

~] Terminates a ~[. It is undefined elsewhere.

~{*str*~} This is an iteration construct. The argument should be a list, which is used as a set of arguments as if for a recursive call to **format**. The string *str* is used repeatedly as the control string. Each iteration can absorb as many elements of the list as it likes; if *str* uses up two arguments by itself, then two elements of the list will get used up each time around the loop. If before any iteration step the list is empty, then the iteration is terminated. Also, if a prefix parameter *n* is given, then there will be at most *n* repetitions of processing of *str*. Here are some simple examples:

```
(format nil "Here it is:~{ ~S~}." '(a b c))
     => "Here it is: A B C."
(format nil "Pairs of things:~{ <~S,~S>~}." '(a 1 b 2 c 3))
     => "Pairs of things: <A,1> <B,2> <C,3>."
```

~:{*str*~} is similar, but the argument should be a list of sublists. At each repetition step one sublist is used as the set of arguments for processing *str*; on the next repetition a new sublist is used, whether or not all of the last sublist had been processed. Example:

```
(format nil "Pairs of things:~:{ <~S,~S>~}."
            '((a 1) (b 2) (c 3)))
     => "Pairs of things: <A,1> <B,2> <C,3>."
```

~@{*str*~} is similar to ~{*str*~}, but instead of using one argument which is a list, all the remaining arguments are used as the list of arguments for the iteration. Example:

```
(format nil "Pairs of things:~@{ <~S,~S>~}."
            'a 1 'b 2 'c 3)
       => "Pairs of things: <A,1> <B,2> <C,3>."
```

~:@{*str*~} combines the features of ~:{*str*~} and ~@{*str*~}. All the remaining arguments are used, and each one must be a list. On each iteration the next argument is used as a list of arguments to *str*. Example:

```
(format nil "Pairs of things:~:@{ <~S,~S>~}."
            '(a 1) '(b 2) '(c 3))
       => "Pairs of things: <A,1> <B,2> <C,3>."
```

Terminating the repetition construct with ~:} instead of ~} forces *str* to be processed at least once even if the initial list of arguments is null (however, it will not override an explicit prefix parameter of zero).

If *str* is empty, then an argument is used as *str*. It must be a string, and precedes any arguments processed by the iteration. As an example, the following are equivalent:

```
(lexpr-funcall #'format stream string args)
(format stream "~1{~:}" string args)
```

This will use string as a formatting string. The ~1{ says it will be processed at most once, and the ~:} says it will be processed at least once. Therefore it is processed exactly once, using args as the arguments.

As another example, the format function itself uses format-error (a routine internal to the format package) to signal error messages, which in turn uses ferror, which uses format recursively. Now format-error takes a string and arguments, just like format, but also prints some additional information: if the control string in ctl-string actually is a string (it might be a list—see below), then it prints the string and a little arrow showing where in the processing of the control string the error occurred. The variable ctl-index points one character after the place of the error.

```
(defun format-error (string &rest args)
  (if (stringp ctl-string)
      (ferror nil "~1{~:}~%~VT↓~%~3X/"~A/"~%"
              string args (+ ctl-index 3) ctl-string)
      (ferror nil "~1{~:}" string args)))
```

This first processes the given string and arguments using ~1{~:}, then tabs a variable amount for printing the down-arrow, then prints the control string between double-quotes. The effect is something like this:

```
(format t "The item is a ~[Foo~;Bar~;Loser~]." 'quux)
>>ERROR: The argument to the FORMAT "~[" command
         must be a number
                   ↓
   "The item is a ~[Foo~;Bar~;Loser~]."
...
```

~} Terminates a ~{. It is undefined elsewhere.

~< *~mincol,colinc,minpad,padchar<text~>* justifies *text* within a field at least *mincol* wide. *text* may be divided up into segments with ~;—the spacing is evenly divided between the text segments. With no modifiers, the leftmost text segment is left justified in the field, and the rightmost text segment right justified; if there is only one, as a special case, it is right justified. The : modifier causes spacing to be introduced before the first text segment; the @ modifier causes spacing to be added after the last. *Minpad*, default 0, is the minimum number of *padchar* (default space) padding characters to be output between each segment. If the total width needed to satisfy these constraints is greater than *mincol*, then *mincol* is adjusted upwards in *colinc* increments. *colinc* defaults to 1. *mincol* defaults to 0. For example,

```
(format nil "~10<foo~;bar~>")           =>  "foo    bar"
(format nil "~10:<foo~;bar~>")          =>  "  foo  bar"
(format nil "~10:@<foo~;bar~>")         =>  "  foo bar "
(format nil "~10<foobar~>")             =>  "    foobar"
(format nil "~10:<foobar~>")            =>  "    foobar"
(format nil "~10@<foobar~>")            =>  "foobar    "
(format nil "~10:@<foobar~>")           =>  "  foobar  "
(format nil "$~10,,,'*<~3f~>" 2.59023) =>  "$******2.59"
```

Note that *text* may include format directives. The last example illustrates how the ~< directive can be combined with the ~f directive to provide more advanced control over the formatting of numbers.

Here are some examples of the use of ~^ within a ~< construct. ~^ is explained in detail below, however the general idea is that it eliminates the segment in which it appears and all following segments if there are no more arguments.

```
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo)
        =>  "            FOO"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar)
        =>  "FOO         BAR"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar 'baz)
        =>  "FOO   BAR   BAZ"
```

The idea is that if a segment contains a ~^, and **format** runs out of arguments, it just stops there instead of getting an error, and it as well as the rest of the segments are ignored.

If the first clause of a ~< is terminated with ~:; instead of ~;, then it is used in a special way. All of the clauses are processed (subject to ~^, of course), but the first one is omitted in performing the spacing and padding. When the padded result has been determined, then if it will fit on the current line of output, it is output, and the text for the first clause is discarded. If, however, the padded text will not fit on the current line, then the text segment for the first clause is output before the padded text. The first clause ought to contain a carriage return (~%). The first clause is always processed, and so any arguments it refers to will be used; the decision is whether to use the resulting segment of text, not whether to process the first clause. If the ~:; has a prefix parameter *n*, then the padded text must fit on the current line with *n* character positions to spare to avoid outputting the first clause's text. For example, the control string

```
"~%;; ~{~<~%;; ~1:; ~S~>~^,~}.~%"
```

can be used to print a list of items separated by commas, without breaking items over line boundaries, and beginning each line with ";; ". The prefix parameter 1 in ~1:; accounts for the width of the comma which will follow the justified item if it is not the last element in the list, or the period if it is. If ~:; has a second prefix parameter, then it is used as the width of the line, thus overriding the natural line width of the output stream. To make the preceding example use a line width of 50, one would write

```
"~%;; ~{~<~%;; ~1,50:; ~S~>~^,~}.~%"
```

If the second argument is not specified, then **format** sees whether the stream handles the **:size-in-characters** message. If it does, then **format** sends that message and uses the first returned value as the line length in characters. If it doesn't, **format** uses **95.** as the line length.

Rather than using this complicated syntax, one can often call the function **format:print-list** (see page 420).

~> Terminates a ~<. It is undefined elsewhere.

~^ This is an escape construct. If there are no more arguments remaining to be processed, then the immediately enclosing ~{ or ~< construct is terminated. If there is no such enclosing construct, then the entire formatting operation is terminated. In the ~< case, the formatting *is* performed, but no more segments are processed before doing the justification. The ~^ should appear only at the *beginning* of a ~< clause, because it aborts the entire clause. ~^ may appear anywhere in a ~{ construct.

If a prefix parameter is given, then termination occurs if the parameter is zero. (Hence ~^ is the same as ~#^.) If two parameters are given, termination occurs if they are equal. If three are given, termination occurs if the second is between the other two in ascending order. Of course, this is useless if all the prefix parameters are constants; at least one of them should be a # or a V parameter.

If ~^ is used within a ~:{ construct, then it merely terminates the current iteration step (because in the standard case it tests for remaining arguments of the current step only); the next iteration step commences immediately. To terminate the entire iteration process, use ~:^.

~Q An escape to arbitrary user-supplied code. *arg* is called as a function; its arguments are the prefix parameters to ~Q, if any. *args* can be passed to the function by using the V prefix parameter. The function may output to **standard-output** and may look at the variables **format:colon-flag** and **format:atsign-flag**, which are **t** or **nil** to reflect the : and @ modifiers on the ~Q. For example,

```
(format t "~VQ" foo bar)
```

is a fancy way to say

```
(funcall bar foo)
```

and discard the value. Note the reversal of order; the V is processed before the Q.

~\ This begins a directive whose name is longer than one character. The name is terminated by another \ character. The following directives have names longer than one character and make use of the ~\ mechanism as part of their operation.

~\date\ This expects an argument that is a universal time (see page 628), and prints it as a date and time using **time:print-universal-date**.
Example:
```
        (format t "It is now ~\date\" (time:get-universal-time))
```
prints
```
        It is now Saturday the fourth of December, 1982; 4:00:32 am
```

~\time\ This expects an argument that is a universal time (see page 628), and prints it in a brief format using **time:print-universal-time**.
Example:
```
        (format t "It is now ~\time\" (time:get-universal-time))
```
prints
```
        It is now 12/04/82 04:01:38
```

~\datime\
This prints the current time and date. It does not use an argument. It is equivalent to using the ~\time\ directive with **(time:get-universal-time)** as argument.

~\time-interval\
This prints a time interval measured in seconds using the function **time:print-interval-or-never**.
Example:
```
        (format t "It took ~\time-interval\." 3601.)
```
prints
```
        It took 1 hour 1 second.
```

You can define your own directives. How to do this is not documented here; read the code. Names of user-defined directives longer than one character may be used if they are enclosed in backslashes (e.g. ~4,3\GRAPH\).

(Note: **format** also allows *control-string* to be a list. If the list is a list of one element, which is a string, the string is simply printed. This is for the use of the **format:outfmt** function below. The old feature wherein a more complex interpretation of this list was possible is now considered obsolete; use **format:output** if you like using lists.)

A condition instance can also be used as the *control-string*. Then the **:report** operation is used to print the condition instance; any other arguments are ignored. This way, you can pass a condition instance directly to any function that normally expects a format string and arguments.

format:print-list *destination element-format list* &optional *separator start-line tilde-brace-options*

This function provides a simpler interface for the specific purpose of printing comma-separated lists with no list element split across two lines; see the description of the ~:; directive (page 418) to see the more complex way to do this within **format**. *destination* tells where to send the output; it can be **t**, **nil**, a **string-nconc**-able string, or a stream, as with **format**. *element-format* is a **format** control-string that tells how to print each element of *list*; it is used as the body of a "~{...~}" construct. *separator*, which defaults to ", " (comma, space) is a string which goes after each element except the last. **format** control commands are not recommended in *separator*. *start-line*, which defaults to three spaces, is a **format** control-string that is used as a prefix at the beginning of each line of output, except the first. **format** control commands are allowed in *separator*, but they should not swallow arguments from *list*. *tilde-brace-options* is a string inserted before the opening "{"; it defaults to the null string, but allows you to insert colon and/or atsign. The line-width of the stream is computed the same way that the ~:; command computes it; it is not possible to override the natural line-width of the stream.

21.6.2 The Output Subsystem

The formatting functions associated with the **format:output** subsystem allow you to do formatted output using Lisp-style control structure. Instead of a directive in a **format** control string, there is one formatting function for each kind of formatted output.

The calling conventions of the formatting functions are all similar. The first argument is usually the datum to be output. The second argument is usually the minimum number of columns to use. The remaining arguments are options—alternating keywords and values.

Options which most functions accept include :padchar, followed by a character to use for padding; :minpad, followed by the minimum number of padding characters to output after the data; and :tab-period, followed by the distance between allowable places to stop padding. To make the meaning of :tab-period clearer, if the value of :tab-period is 5, the minimum size of the field is 10, and the value of :minpad is 2, then a datum that takes 9 characters will be padded out to 15 characters. The requirement to use at least two characters of padding means it can't fit into 10 characters, and the :tab-period of 5 means the next allowable stopping place is at 10+5 characters. The default values for :minpad and :tab-period, if they are not specified, are zero and one. The default value for :padchar is space.

The formatting functions always output to **standard-output** and do not require an argument to specify the stream. The macro **format:output** allows you to specify the stream or a string, just as **format** does, and also makes it convenient to concatenate constant and variable output.

format:output *stream string-or-form...* *Macro*

format:output makes it convenient to intersperse arbitrary output operations with printing of constant strings. **standard-output** is bound to *stream*, and each *string-or-form* is processed in succession from left to right. If it is a string, it is printed; otherwise it is a form, which is evaluated for effect. Presumably the forms will send output to **standard-output**.

If *stream* is written as **nil**, then the output is put into a string which is returned by **format:output**. If *stream* is written as **t**, then the output goes to the prevailing value of **standard-output**. Otherwise *stream* is a form, which must evaluate to a stream.

Here is an example:

```
(format:output t "FOO is " (prin1 foo) " now." (terpri))
```

Because **format:output** is a macro, what matters about *stream* is not whether it *evaluates* to **t** or **nil**, but whether it is actually written as **t** or **nil**.

format:outfmt *string-or-form...* *Macro*

Some system functions ask for a **format** control string and arguments, to be printed later. If you wish to generate the output using the formatted output functions, you can use **format:outfmt**, which produces a control argument that will eventually make **format** print the desired output (this is a list whose one element is a string containing the output). A call to **format:outfmt** can be used as the second argument to **ferror**, for example:

```
        (ferror nil (format:outfmt "Foo is " (format:onum foo)
                                   " which is too large"))
```

format:onum *number* &optional *radix minwidth* &rest *options*

format:onum outputs *number* in base *radix*, padding to at least *minwidth* columns and obeying the other padding options specified as described above.

radix can be a number, or it can be :roman, :english, or :ordinal. The default *radix* is 10. (decimal).

Two special keywords are allowed as options: :signed and :commas. :signed with value t means print a sign even if the number is positive. :commas with value t means print a comma every third digit in the customary way. These options are meaningful only with numeric radices.

format:ofloat *number* &optional *n-digits force-exponential-notation minwidth* &rest *options*

format:ofloat outputs *number* as a floating point number using *n-digits* digits. If *force-exponential-notation* is non-nil, then an exponent is always used. *minwidth* and *options* are used to control padding as usual.

format:ostring *string* &optional *minwidth* &rest *options*

format:ostring outputs *string*, padding to at least *minwidth* columns if *minwidth* is not nil, and obeying the other padding options specified as described above.

Normally the contents of the string are left-justified; any padding follows the data. The special option :right-justify causes the padding to come before the data. The amount of padding is not affected.

The argument need not really be a string. Any Lisp object is allowed, and it is output by princ.

format:oprint *object* &optional *minwidth* &rest *options*

format:oprint prints *object*, any Lisp object, padding to at least *minwidth* columns if *minwidth* is not nil, and obeying the padding options specified as described above.

Normally the data are left justified; any padding follows. The special option :right-justify causes the padding to come before the data. The amount of padding is not affected.

The printing of the object is done with prin1.

format:ochar *character* &optional *style top-explain minwidth* &rest *options*

format:ochar outputs *character* in one of three styles, selected by the *style* argument. *minwidth* and *options* control padding as usual.

:read or nil	The character is printed using #/ or #\ so that it could be read back in.
:editor	Output is in the style of the string "Meta-Rubout". If the character has a name, the name is used instead of the character.

:brief	Brief prefixes such as "C-" and "M-" are used, rather than "Control-" or "Meta-". Also, character names are used only if there are meta bits present.
:lozenged	The output is the same as that of the :editor style, but If the character is not a graphic character or if it has meta bits, and the stream supports the :display-lozenged-string operation, that operation is used instead of :string-out to print the text. On windows this operation puts the character name inside a lozenge.
:sail	*α*, , etc. are used to represent "Control" and "Meta", and shorter names for characters are also used when possible. See section 21.1, page 362.

top-explain is useful with the :editor, :brief and :sail styles. It says that any character that has to be typed using the Top or Greek keys should be followed by an explanation of how to type it. For example: "→ (Top-K)" or "*α* (Greek-a)".

format:tab *mincol* &rest *options*

format:tab outputs padding at least until column *mincol*. It is the only formatting function that bases its actions on the actual cursor position rather than the width of what is being output. The padding options :padchar, :minpad, and :tab-period are obeyed. Thus, at least the :minpad number of padding characters are output even if that goes past *mincol*, and once past *mincol*, padding can only stop at a multiple of :tab-period characters past *mincol*.

In addition, if the :terpri option is t, then if column *mincol* is passed, format:tab starts a new line and indents it to *mincol*.

The :unit option specifies the units of horizontal position. The default is to count in units of characters. If :unit is specified as :pixel, then the computation (and the argument *mincol* and the :minpad and :tab-period options) are in units of pixels.

format:pad (*minwidth option...*) *body...* *Macro*

format:pad is used for printing several items in a fixed amount of horizontal space, padding between them to use up any excess space. Each of the *body* forms prints one item. The padding goes between items. The entire format:pad always uses at least *minwidth* columns; any columns that the items don't need are distributed as padding between the items. If that isn't enough space, then more space is allocated in units controlled by the :tab-period option until there is enough space. If it's more than enough, the excess is used as padding.

If the :minpad option is specified, then at least that many pad characters must go between each pair of items.

Padding goes only between items. If you want to treat several actual pieces of output as one item, put a progn around them. If you want padding before the first item or after the last, as well as between the items, include a dummy item nil at the beginning or the end.

If there is only one item, it is right justified. One item followed by **nil** is left-justified. One item preceded and followed by **nil** is centered. Therefore, **format:pad** can be used to provide the usual padding options for a function that does not provide them itself.

format:plural *number singular* &optional *plural*

format:plural outputs either the singular or the plural form of a word depending on the value of *number*. The singular is used if and only if *number* is 1. *singular* specifies the singular form of the word. **string-pluralize** is used to compute the plural, unless *plural* is explicitly specified.

It is often useful for *number* to be a value returned by **format:onum**, which returns its argument. For example:

```
        (format:plural (format:onum n-frobs) " frob")
```

will print "1 frob" or "2 frobs".

format:breakline *linel print-if-terpri print-always...* *Macro*

format:breakline is used to go to the next line if there is not enough room for something to be output on the current line. The *print-always* forms print the text which is supposed to fit on the line. *linel* is the column before which the text must end. If it doesn't end before that column, then **format:breakline** moves to the next line and executes the *print-if-terpri* form before doing the *print-always* forms.

Constant strings are allowed as well as forms for *print-if-terpri* and *print-always*. A constant string is just printed.

To go to a new line unconditionally, simply call **terpri**.

Here is an example that prints the elements of a list, separated by commas, breaking lines between elements when necessary.

```
(defun pcl (list linel)
  (do ((l list (cdr l))) ((null l))
    (format:breakline linel "  "
      (princ (car l))
      (and (cdr l) (princ ", ")))))
```

21.6.3 Formatting Lisp Code

grindef *function-spec...* *Special Form*

Prints the definitions of one or more functions, with indentation to make the code readable. Certain other "pretty-printing" transformations are performed: The quote special form is represented with the ' character. Displacing macros are printed as the original code rather than the result of macro expansion. The code resulting from the backquote (`) reader macro is represented in terms of `.

The subforms to grindef are the function specs whose definitions are to be printed; the usual way grindef is used is with a form like (grindef foo) to print the definition of foo. When one of these subforms is a symbol, if the symbol has a value its value is prettily printed also. Definitions are printed as defun special forms, and values are printed as setq special forms.

If a function is compiled, grindef will say so and try to find its previous interpreted definition by looking on an associated property list (see uncompile (page 228). This will only work if the function's interpreted definition was once in force; if the definition of the function was simply loaded from a QFASL file, grindef will not find the interpreted definition and will not be able to do anything useful.

With no subforms, grindef assumes the same arguments as when it was last called.

grind-top-level *obj* &optional *width* (*stream* standard-output) (*untyo-p* nil) (*displaced* 'si:displaced) (*terpri-p* t) *notify-fun* *loc*

Pretty-prints *obj* on *stream*, putting up to *width* characters per line. This is the primitive interface to the pretty-printer. Note that it does not support variable-width fonts. If the *width* argument is supplied, it is how many characters wide the output is to be. If *width* is unsupplied or nil, grind-top-level will try to figure out the "natural width" of the stream, by sending a :size-in-characters message to the stream and using the first returned value. If the stream doesn't handle that message, a width of 95. characters is used instead.

The remaining optional arguments activate various strange features and usually should not be supplied. These options are for internal use by the system and are documented here for only completeness. If *untyo-p* is t, the :untyo and :untyo-mark operations will be used on *stream*, speeding up the algorithm somewhat. *displaced* controls the checking for displacing macros; it is the symbol which flags a place that has been displaced, or nil to disable the feature. If *terpri-p* is nil, grind-top-level does not advance to a fresh line before printing.

If *notify-fun* is non-nil, it is a function that takes three arguments, which is called for each "token" in the pretty-printed output. Tokens are atoms, open and close parentheses, and reader macro characters such as '. The arguments to *notify-fun* are the token, its "location" (see next paragraph), and t if it is an atom or nil if it is a character.

loc is the "location" (typically a cons) whose car is *obj*. As the grinder recursively descends through the structure being printed, it keeps track of the location where each thing came from, for the benefit of the *notify-fun*, if any. This makes it possible for a program to correlate the printed output with the list structure. The "location" of a close parenthesis is t, because close parentheses have no associated location.

21.7 Rubout Handling

The rubout handler is a feature of all interactive streams, that is, streams that connect to terminals. Its purpose is to allow the user to edit minor mistakes in type-in. At the same time, it is not supposed to get in the way; input is to be seen by Lisp as soon as a syntactically complete form has been typed. The definition of "syntactically complete form" depends on the function that is reading from the stream; for **read**, it is a Lisp expression.

Some interactive streams ("editing Lisp listeners") have a rubout handler that allows input to be edited with the full power of the ZWEI editor. Most windows have a rubout handler that apes ZWEI, implementing about twenty common ZWEI commands. The cold load stream has a simple rubout handler that allows just rubbing out of single characters, and a few simple commands like clearing the screen and erasing the entire input typed so far. All three kinds of rubout handler use the same protocol, which is described in this section. We also say a little about the most common of the three rubout handlers.
[Eventually some version of ZWEI will be used for all streams except the cold load stream]

The tricky thing about the rubout handler is the need for it to figure out when you are all done. The idea of a rubout handler is that you can type in characters, and they are saved up in a buffer so that if you change your mind, you can rub them out and type different characters. However, at some point, the rubout handler has to decide that the time has come to stop putting characters into the buffer and to let the function parsing the input, such as **read**, return. This is called "activating". The right time to activate depends on the function calling the rubout handler, and may be very complicated (if the function is **read**, figuring out when one Lisp expression has been typed requires knowledge of all the various printed representations, what all currently-defined reader macros do, and so on). Rubout handlers should not have to know how to parse the characters in the buffer to figure out what the caller is reading and when to activate; only the caller should have to know this. The rubout handler interface is organized so that the calling function can do all the parsing, while the rubout handler does all the handling of rubouts, and the two are kept completely separate.

The basic way that the rubout handler works is as follows. When an input function that reads characters from a stream, such as **read** or **readline** (but not **tyi**), is invoked with a stream which has **:rubout-handler** in its **:which-operations** list, that function "enters" the rubout handler. It then goes ahead :tyi'ing characters from the stream. Because control is inside the rubout handler, the stream will echo these characters so the user can see what he is typing. (Normally echoing is considered to be a higher-level function outside of the province of streams, but when the higher-level function tells the stream to enter the rubout handler it is also handing it the responsibility for echoing.) The rubout handler is also saving all these characters in a buffer, for reasons disclosed in the following paragraph. When the function, **read** or whatever, decides it has enough input, it returns and control "leaves" the rubout handler. That was the easy case.

If the user types a rubout, a ***throw** is done out of all recursive levels of **read**, reader macros, and so forth, back to the point where the rubout handler was entered. Also the rubout is echoed by erasing from the screen the character which was rubbed out. Now the **read** is tried over again, re-reading all the characters that have not been rubbed out, not echoing them this time. When the saved characters have been exhausted, additional input is read from the user in the usual fashion.

The effect of this is a complete separation of the functions of rubout handling and parsing, while at the same time mingling the execution of these two functions in such a way that input is always "activated" at just the right time. It does mean that the parsing function (in the usual case, read and all macro-character definitions) must be prepared to be thrown through at any time and should not have non-trivial side-effects, since it may be called multiple times.

If an error occurs while inside the rubout handler, the error message is printed and then additional characters are read. When the user types a rubout, it rubs out the error message as well as the character that caused the error. The user can then proceed to type the corrected expression; the input will be reparsed from the beginning in the usual fashion.

The rubout handler based on the ZWEI editor interprets control characters in the usual ZWEI way: as editing commands, allowing you to edit your buffered input.

The common rubout handler also recognizes a subset of the editor commands, including Rubout, Control-F and Meta-F and others. Typing Help while in the rubout handler displays a list of the commands. The kill and yank commands in the rubout handler use the same kill ring as the editor, so you can kill an expression in the editor and yank it back into a rubout handler with Control-Y, or kill an expression in the rubout handler with Control-K or Clear-input and yank it back in the editor. The rubout processor also keeps a ring buffer of most recent input strings (a separate ring for each stream), and the commands Control-C and Meta-C retrieve from this ring just as Control-Y and Meta-Y do from the kill ring.

When not inside the rubout handler, and when typing at a program that uses control characters for its own purposes, control characters are treated the same as ordinary characters.

Some programs such as the debugger allow the user to type either a control character or an expression. In such programs, you are really not inside the rubout handler unless you have typed the beginning of an expression. When the input buffer is empty, a control character is treated as a command for the program (such as, Control-C to continue in the debugger); when there is text in the rubout handler buffer, the same character is treated as a rubout handler command. Another consequence of this is that the message you get by typing Help varies, being either the rubout handler's documentation or the debugger's documentation.

The following explanation tells you how to write your own function that invokes the rubout handler. The functions read and readline both work this way. You should use the readline1 example, below, as a template for writing your own function.

The way that the rubout handler is entered is complicated, since a *catch must be established. The variable rubout-handler is non-nil if the current process is inside the rubout handler. This is used to handle recursive calls to read from inside reader macros and the like. If rubout-handler is nil, and the stream being read from has :rubout-handler in its :which-operations, functions such as read send the :rubout-handler message to the stream with arguments of a list of options, the function, and its arguments. The rubout handler initializes itself and establishes its *catch, then calls back to the specified function with rubout-handler bound to t. User-written input reading functions should follow this same protocol to get the same input editing benefits as **read** and **readline**.

rubout-handler *Variable*

t if control is inside the rubout handler in this process.

As an example of how to use the rubout handler, here is a simplified version of the **readline** function. It doesn't bother about end-of-file handling, use of **:line-in** for efficiency, etc.

```
(defun readline1 (stream)
  ;; If stream does rubout handling, get inside rubout handler
  (cond ((and (not rubout-handler)
              (memq ':rubout-handler
                    (funcall stream ':which-operations)))
         (funcall stream ':rubout-handler '() #'readline1 stream))
        ;; Accumulate characters until Return
        (t (do ((ch (funcall stream ':tyi)
                    (funcall stream ':tyi))
                (len 100)
                (string (make-array 100 ':type 'art-string))
                (idx 0))
               ((or (null ch) (= ch #\return))
                (adjust-array-size string idx)
                string)
             (if (= idx len)
                 (adjust-array-size string (setq len (+ len 40))))
             (aset ch string idx)
             (setq idx (1+ idx))))))
```

The first argument to the **:rubout-handler** message is a list of options. The second argument is the function that the rubout handler should call to do the reading, and the rest of the arguments are passed to that function. Note that in the example above, **readline1** is sending the **:rubout-handler** message passing itself as the function, and its own arguments as the arguments. This is the usual thing to do. It isn't passing any options. The returned values of the message are normally the returned values of the function (except sometimes when the **:full-rubout** option is used; see below).

Each option in the list of options given as the first argument to the **:rubout-handler** message consists of a list whose first element is a keyword and whose remaining elements are "arguments" to that keyword. Note that this is not the same format as the arguments to a typical function that takes keyword arguments; rather this is an a-list of options. The standard options are:

(:full-rubout *val***)**

If the user rubs out all the characters he typed, then control will be returned from the rubout handler immediately. Two values are returned; the first is **nil** and the second is *val*. (If the user doesn't rub out all the characters, then the rubout handler propagates multiple values back from the function that it calls, as usual.) In the absence of this option, the rubout handler would simply wait for more characters to be typed in and would ignore any additional rubouts.

(:pass-through *char1 char2*...)
: The characters *char1*, *char2*, etc. are not to be treated as special by the rubout handler. You can use this to override the default processing of characters such as **Clear-input** and to receive control characters. Any function that reads input and uses non-printing characters for anything should list them in a **:pass-through** option. This way, if input is being rubout-handled by the editor, those non-printing characters will get their desired meaning rather than their meaning as editor commands.

(:prompt *function*)
(:reprompt *function*)
: When it is time for the user to be prompted, *function* is called with two arguments. The first is a stream it may print on; the second is the character which caused the need for prompting, e.g. **#\clear-input** or **#\clear-screen**, or **nil** if the rubout handler was just entered.

The difference between **:prompt** and **:reprompt** is that the latter does not call the prompt function when the rubout handler is first entered, but only when the input is redisplayed (e.g. after a screen clear). If both options are specified then **:reprompt** overrides **:prompt** except when the rubout handler is first entered.

function may also be a string. Then it is simply printed.

If the rubout handler is exited with an empty buffer due to the **:full-rubout** option, whatever prompt was printed is erased.

(:initial-input *string*)
: Pretends that the user typed *string*. When the rubout handler is entered, *string* is typed out. The user can input more characters or rub out characters from it.

(:do-not-echo *char-1 char-2*...)
: The characters *char-1*, *char-2*, etc. are not to be echoed when the user types them. The comparison is done with **=**, not **char-equal**. You can use this to suppress echoing of the **return** character that terminated a **readline**, for example.

21.8 The :read and :print Stream Operations

A stream can specially handle the reading and printing of objects by handling the **:read** and **:print** stream operations. Note that these operations are optional and most streams do not support them.

If the **read** function is given a stream that has **:read** in its which-operations, then instead of reading in the normal way it sends the **:read** message to the stream with one argument, **read's** *eof-option* if it had one or a magic internal marker if it didn't. Whatever the stream returns is what **read** returns. If the stream wants to implement the **:read** operation by internally calling **read**, it must use a different stream that does not have **:read** in its which-operations.

If a stream has :print in its which-operations, it may intercept all object printing operations, including those due to the **print**, **prin1**, and **princ** functions, those due to **format**, and those used internally, for instance in printing the elements of a list. The stream receives the :print message with three arguments: the object being printed, the *prindepth* (for comparison against the **prinlevel** variable), and *slashify-p* (**t** for **prin1**, **nil** for **princ**). If the stream returns **nil**, then normal printing takes place as usual. If the stream returns non-**nil**, then **print** does nothing; the stream is assumed to have output an appropriate printed representation for the object. The two following functions are useful in this connection; however, they are in the **system-internals** package and may be changed without much notice.

si:print-object *object prindepth slashify-p stream* &optional *which-operations*

Outputs the printed-representation of *object* to *stream*, as modified by *prindepth* and *slashify-p*. This is the internal guts of the Lisp printer. When a stream's :print handler calls this function, it should supply the list (:string-out) for *which-operations*, to prevent itself from being called recursively. Or it can supply **nil** if it does not want to receive :string-out messages.

If you want to customize the behavior of all printing of Lisp objects, advising (see section 27.10, page 592) this function is the way to do it. See section 21.2.1, page 370.

si:print-list *list prindepth slashify-p stream which-operations*

This is the part of the Lisp printer that prints lists. A stream's :print handler can call this function, passing along its own arguments and its own which-operations, to arrange for a list to be printed the normal way and the stream's :print hook to get a chance at each of the list's elements.

21.9 Accessing Files

The Lisp Machine can access files on a variety of remote file servers, which are typically (but not necessarily) accessed through the Chaosnet, as well as accessing files on the Lisp Machine itself, if the machine has its its own file system. This section tells you how to get a stream which reads or writes a given file, and what the device-dependent operations on that stream are. Files are named with *pathnames*. Since pathnames are quite complex they have their own chapter; see chapter 22, page 453. You are not allowed to refer to files without first logging in, and you may also need to specify a username and password for the host on which the file is stored; see page 648.

with-open-file (*stream pathname options...*) *body...* *Special Form*

Evaluates the *body* forms with the variable *stream* bound to a stream that reads or writes the file named by the value of *pathname*. The *options* forms evaluate to the file-opening options to be used; see page 434.

When control leaves the body, either normally or abnormally (via ***throw**), the file is closed. If a new output file is being written and control leaves abnormally, the file is aborted and it is as if it were never written. Because it always closes the file, even when an error exit is taken, **with-open-file** is preferred over **open**. Opening a large number of files and forgetting to close them tends to break some remote file servers, ITS's for example.

pathname is the name of the file to be opened; it can be a pathname object, a string, a symbol, or a Maclisp-compatible "namelist". It can be anything acceptable to fs:parse-pathname; the complete rules for parsing pathnames are explained in chapter 22, page 453.

If an error, such as file not found, occurs, the result is whatever open does, based on the value of the :error option passed to it.

with-open-file-case (*stream pathname options...*) *clauses...* *Special Form*

This opens and closes the file like with-open-file, but what happens afterward is determined by *clauses* that are like the clauses of a condition-case. Each clause begins with a condition name or a list of condition names and is executed if open signals a condition that possesses any of those names. A clause beginning with the symbol :no-error is executed if open returns. This would be where the reading or writing of the file would be done.

Example:

```
(with-open-file-case (stream (send generic-pathname
                                   ':source-pathname))
  (sys:remote-network-error (format t "~&Host down."))
  (fs:file-not-found (format t "~&(New file)"))
  (:no-error (setq list (read stream))))
```

file-retry-new-pathname (*pathname-var condition-names...*) *body...* *Special Form*

file-retry-new-pathname-if *cond-form* (*pathname-var condition-names...*) *body...* *Special Form*

file-retry-new-pathname executes *body*. If *body* does not signal any of the conditions in *condition-names*, *body*'s values are simply returned. If any of *condition-names* is signaled, file-retry-new-pathname reads a new pathname, setq's *pathname-var* to it, and executes *body* again.

file-retry-new-pathname-if is similar, but the conditions are handled only if *cond-form*'s value is non-nil.

with-open-file-retry (*stream* (*pathname-var condition-names...*) *options...*) *body...* *Special Form*

Like with-open-file, except that if an error occurs inside open with one of the specified *condition-names*, a new pathname is read, the variable *pathname-var* is setq'd to it, and the open is retried.

open *pathname* &rest *options*

Returns a stream that is connected to the specified file. Unlike Maclisp, the open function creates streams only for *files*; streams of other kinds are created by other functions. The *pathname* and *options* arguments are the same as in with-open-file; see above.

When the caller is finished with the stream, it should close the file by using the :close operation or the close function. The with-open-file special form does this automatically and so is usually preferred. open should only be used when the control structure of the program necessitates opening and closing of a file in some way more complex than the simple way provided by **with-open-file**. Any program that uses **open** should set up **unwind-protect** handlers (see page 56) to close its files in the event of an abnormal exit.

close *stream*
The **close** function simply sends the **:close** message to *stream*.

deletef *file* &optional (*error-p* t) *query?*
file can be a pathname or a stream that is open to a file. The specified file is deleted. *pathname* may contain wildcard characters, in which case multiple files are deleted.

If *query?* is non-**nil**, the user is queried about each file (whether there are wildcards or not). Only the files that the user confirms are actually deleted.

If *error-p* is **t**, then if an error occurs it will be signalled as a Lisp error. If *error-p* is **nil** and an error occurs, the error message will be returned as a string. Otherwise, the value is a list of elements, one for each file considered. The car of each element is the truename of the file, and the cadr is non-**nil** if the file was actually deleted (it will always be **t** unless querying was done).

undeletef *file* &optional (*error-p* t) *query?*
file can be a pathname or a stream that is open to a file. The specified file is undeleted. Wildcards are allowed, just as in **deletef**. The rest of the calling conventions are the same as well.

Not all file systems support undeletion, and if it is not supported on the one you are using, it get an error or return a string according to *error-p*. To find out whether a particular file system supports this, send the **:undeletable-p** operation to a pathname. If it returns **t**, the file system of that pathname supports undeletion.

renamef *file new-name* &optional (*error-p* t) *query?*
file can be a pathname or a stream that is open to a file. The specified file is renamed to *new-name* (a pathname). *file* may contain wildcards, in which case multiple files are renamed. Each file's new name is produced by passing *new-name* to **merge-pathname-defaults** with the file's truename as the defaults. Therefore, *new-name* should be a string in this case.

If *query?* is non-**nil**, the user is queried about each file (whether there are wildcards or not). Only the files that the user confirms are actually renamed.

If *error-p* is **t**, then if an error occurs it will be signalled as a Lisp error. If *error-p* is **nil** and an error occurs, the error message will be returned as a string. Otherwise, the value is a list of elements, one for each file considered. The car of each element is the original truename of the file, the cadr is the name it was to be renamed to, and the caddr is non-**nil** if the file was renamed. The caddr is **nil** if the user was queried and said no.

copy-file *file new-file* &optional (*error-p* t) *copy-mode*

copy-file is called almost like renamef. Instead of renaming files, it copies them. Querying is not done.

copy-mode tells copy-file whether to copy the file as characters or as binary; or else, how to decide which of those to do. The possible values are :characters, :binary, :ask, :never-ask, and nil. :characters and :binary specify straightforwardly which kind of transfer to do. :ask says to always ask the user about each file. :never-ask says to guess about each file from its byte size and filetype. nil says to guess if guessing appears to be reliable, otherwise ask the user.

probef *pathname*

Returns nil if there is no file named *pathname*, otherwise returns a pathname that is the true name of the file, which can be different from *pathname* because of file links, version numbers, etc.

Any problem in opening the file except for fs:file-not-found signals an error.

viewf *pathname* &optional (*output-stream* standard-output) *leader*

Copies the contents of the file named *pathname*, opened in character mode, onto output-stream. Normally this has the effect of printing the file on the terminal. *leader* is passed along to stream-copy-until-eof (see page 388).

fs:close-all-files

Closes all open files. This is useful when a program has run wild opening files and not closing them. It closes all the files in :abort mode (see page 394), which means that files open for output will be deleted. Using this function is dangerous, because you may close files out from under various programs like ZMACS and ZMAIL; only use it if you have to and if you feel that you know what you're doing.

The *options* used when opening a file are normally alternating keywords and values, like any other function that takes keyword arguments. In addition, for compatibility with the Maclisp open function, if only a single option is specified it is either a keyword or a list of keywords (not alternating with values).

The file-opening options control things like whether the stream is for input from a existing file or output to a new file, whether the file is text or binary, etc.

The following option keywords are standardly recognized; additional keywords can be implemented by particular file system hosts.

[Are all these keywords supported by all file systems?]

:direction	The possible values are :input (the default), :output, and nil. The first two should be self-explanatory. nil means that this is a "probe" opening; no data are to be transferred, the file is being opened only to access or change its properties.
:characters	The possible values are t (the default), nil, which means that the file is a binary file, and :default, which means that the file system should decide whether the file contains characters or binary data and open it in the appropriate mode.

:byte-size — The possible values are nil (the default), a number, which is the number of bits per byte, and :default, which means that the file system should choose the byte size based on attributes of the file. If the file is being opened as characters, nil selects the appropriate system-dependent byte size for text files; it is usually not useful to use a different byte size. If the file is being opened as binary, nil selects the default byte size of 16 bits.

:error — The possible values are t (the default), :reprompt and nil. If an error occurs, this option controls whether the error is signalled to the user (t), a new pathname is read (:reprompt) or a string containing an error message is returned instead of a stream (nil).

:reprompt should be used whenever the caller does not need to know which file was ultimately opened. More sophisticated callers should use with-open-file-retry or file-retry-new-pathname (see page 432).

:new-file — If the value is t, the file system is allowed to create a new file. If the value is nil, an existing file must be opened. The default is t if :direction :output is specified, otherwise nil.

:new-version — This controls what happens if the version field of the pathname being opened is :newest. If the value is nil, the newest existing version of the file is found. If the value is t (the default when :direction :output is specified), then the next-higher-numbered version of the file is to be created.

:old-file — This keyword controls what happens if a file with the specified name already exists when :direction :output is specified. Possible values are:

nil or :replace	The existing file is to be replaced when the new file is closed (providing :abort is not specified when closing). This is the default if :new-file is specified as, or defaults to, t.
t or :rewrite	The new data should be stored into the existing file. This is the default if :new-file is specified as nil.
:append	Append new data to the end of the file.
:error	Signal an error (file already exists).
:rename	The old file is to be renamed to some other name, to get it out of the way.
:rename-and-delete	The old file is renamed, possibly immediately, and deleted when the new file is closed.
:new-version	Create a new version, instead of opening the version number specified in the pathname.

:inhibit-links — The default is nil. If the pathname is the name of a file-system link, and this option is t, the link itself is opened rather than the file it points to. This is useful only with probe openings, since links contain no data.

:deleted — The default is nil. If t is specified, and the file system has the concept of deleted but not expunged files, it is possible to open a deleted file. Otherwise deleted files are invisible.

:temporary	The default is nil. If t is specified, the file is marked as temporary, if the file system has that concept.
:preserve-dates	The default is nil. If t is specified, the file's reference and modification dates are not updated.
:flavor	This controls the kind of file to be opened. The default is nil, a normal file. Other possible values are :directory and :link.
:link-to	When creating a file with :flavor :link, this keyword must be specified; its value is a pathname that becomes the target of the link.
:estimated-size	The value may be nil (the default), which means there is no estimated size, or a number of bytes. Some file systems use this to optimize disk allocation.
:physical-volume	The value may be nil (the default), or a string that is the name of a physical volume on which the file is to be stored. This is not meaningful for all file systems.
:logical-volume	The value may be nil (the default), or a string that is the name of a logical volume on which the file is to be stored. This is not meaningful for all file systems.
:incremental-update	The value may be nil (the default), or t to cause the file system to take extra pains to write data onto the disk more often.
:super-image	The value may be nil (the default), or t, which disables the special treatment of rubout in ascii files. Normally, rubout is an escape which causes the following character to be interpreted specially, allowing all characters from 0 through 376 to be stored. This applies to PDP-10 file servers only.
:raw	The value may be nil (the default), or t, which disables all character set translation in ascii files. This applies to PDP-10 file servers only.

In the Maclisp compatibility mode, there is only one *option*, and it is either a symbol or a list of symbols. These symbols are recognized no matter what package they are in, since Maclisp does not have packages. The following symbols are recognized:

in, read	Select opening for input (the default).
out, write, print	Select opening for output; a new file is to be created.
binary, fixnum	Select binary mode; otherwise character mode is used. Note that fixnum mode uses 16-bit binary words and is not compatible with Maclisp fixnum mode, which uses 36-bit words. On the PDP-10, fixnum files are stored with two 16-bit words per PDP-10 word, left-justified and in PDP-10 byte order.
character, ascii	The opposite of fixnum. This is the default.

single, block Ignored for compatibility with the Maclisp **open** function.

byte-size Must be followed by a number in the options list, and must be used in combination with **fixnum**. The number is the number of bits per byte, which can be from 1 to 16. On a PDP-10 file server these bytes will be packed into words in the standard way defined by the **ILDB** instruction. The **:tyi** stream operation will (of course) return the bytes one at a time.

probe, error, noerror, raw, super-image, deleted, temporary
These are not available in Maclisp. The corresponding keywords in the normal form of file-opening options are preferred over these.

21.9.1 Loading Files

To *load* a file is to read through the file, evaluating each form in it. Programs are typically stored in files; the expressions in the file are mostly special forms such as **defun** and **defvar** which define the functions and variables of the program.

Loading a compiled (or QFASL) file is similar, except that the file does not contain text but rather pre-digested expressions created by the compiler which can be loaded more quickly.

These functions are for loading single files. There is a system for keeping track of programs which consist of more than one file; for further information refer to chapter 25, page 520.

load *pathname* &optional *pkg nonexistent-ok dont-set-default*
This function loads the file named by *pathname* into the Lisp environment. If the file is a QFASL file, it calls **fasload**; otherwise it calls **readfile**. Normally the file is read into its "home" package, but if *pkg* is supplied it is the package in which the file is to be read. *pkg* can be either a package or the name of a package as a string or a symbol. If *pkg* is not specified, **load** prints a message saying what package the file is being loaded into. If *nonexistent-ok* is specified, **load** just returns if the file cannot be opened.

pathname can be anything acceptable to **fs:parse-pathname**; pathnames and the complete rules for parsing them are explained in chapter 22, page 453. *pathname* is defaulted from **fs:load-pathname-defaults** (see page 464), which is the set of defaults used by **load**, **qc-file**, and similar functions. Normally **load** updates the pathname defaults from *pathname*, but if *dont-set-default* is specified this is suppressed.

If *pathname* contains an name but no type, **load** will first look for the file with a type of **QFASL**, then look for a type of **LISP**.

readfile *pathname* &optional *pkg no-msg-p*
readfile is the version of **load** for text files. It reads and evaluates each expression in the file. As with **load**, *pkg* can specify what package to read the file into. Unless *no-msg-p* is t, a message is printed indicating what file is being read into what package. The defaulting of *pathname* is the same as in **load**.

fasload *pathname* &optional *pkg* *no-msg-p*

fasload is the version of load for QFASL files. It defines functions and performs other actions as directed by the specifications inserted in the file by the compiler. As with load, *pkg* can specify what package to read the file into. Unless *no-msg-p* is t, a message is printed indicating what file is being read into what package. The defaulting of *pathname* is the same as in load.

21.9.2 File Attribute Lists

Any text file can contain an "attribute list" that specifies several attributes of the file. The above loading functions, the compiler, and the editor look at this property list. Attribute lists are especially useful in program source files, i.e. a file that is intended to be loaded (or compiled and then loaded). QFASL files also contain attribute lists, copied from their source files.

If the first non-blank line in a text file contains the three characters "-*-", some text, and "-*-" again, the text is recognized as the file's attribute list. Each attribute consists of the attribute name, a colon, and the attribute value. If there is more than one attribute they are separated by semicolons. An example of such an attribute list is:

```
; -*- Mode:Lisp; Package:Cellophane; Base:10 -*-
```

The semicolon makes this line look like a comment rather than a Lisp expression. This defines three properties: mode, package, and base. Another example is:

```
.c Part of the Lisp machine manual.  -*- Mode:Bolio -*-
```

An attribute name is made up of letters, numbers, and otherwise-undefined punctuation characters such as hyphens. An attribute value can be such a name, or a decimal number, or several such items separated by commas. Spaces may be used freely to separate tokens. Upper and lower-case letters are not distinguished. There is *no* quoting convention for special characters such as colons and semicolons.

If the attribute list text contains no colons, it is an old Emacs format, containing only the value of the Mode attribute.

The file attribute list format actually has nothing to do with Lisp; it is just a convention for placing some information into a file that is easy for a program to interpret. The Emacs editor on the PDP-10 knows how to interpret these attribute lists (primarily in order to look at the Mode attribute).

The Lisp Machine handles the attribute list stored in the file by parsing it into a Lisp data structure, a property list. Attribute names are interpreted as Lisp symbols and are interned on the keyword package. Numbers are interpreted as Lisp fixnums and are read in decimal. If a attribute value contains any commas, then the commas separate several expressions that are formed into a list.

When a file is compiled, its attribute list data structure is stored in the QFASL file. It can be loaded back from the QFASL file as well. The representation in the QFASL file resembles nothing described here, but when the attribute list is extracted from there, the same Lisp data structure described above is obtained.

When a file is edited, loaded, or compiled, its file attribute list is read in and the properties are stored on the property list of the generic pathname (see section 22.5, page 467) for that file, where they can be retrieved with the **:get** and **:plist** messages. This is done using the function **fs:read-attribute-list**, below. So the way you examine the properties of a file is usually to use messages to a pathname object that represents the generic pathname of a file. Note that there are other properties there, too.

Here the attribute names with standard meanings:

Mode	The editor major mode to be used when editing this file. This is typically the name of the language in which the file is written. The most common values are **Lisp** and **Text**.
Package	This attribute specifies the package in which symbols in the file should be interned. The attribute may be either the name of a package, or a list that specifies both the package name and how to create the package if it does not exist. If it is a list, it should look like (*name superpackage initial-size ...options...*). See chapter 24, page 506 for more information about packages.
Base	The number base in which the file is written (remember, it is always parsed in decimal). This affects both **ibase** and **base**, since it is confusing to have the input and output bases be different. The most common values are 8 and 10.
Lowercase	If the attribute value is not **nil**, the file is written in lower-case letters and the editor does not translate to upper case. (The editor does not translate to upper case by default unless the user selects "Electric Shift Lock" mode.)
Fonts	The attribute value is a list of font names, separated by commas. The editor uses this for files that are written in more than one font.
Backspace	If the attribute value is not **nil**, **Overstrike** characters in the file should cause characters to overprint on each other. The default is to disallow overprinting and display **Overstrike** the way other special function keys are displayed. This default is to prevent the confusion that can be engendered by overstruck text.
Patch-File	If the attribute value is not **nil**, the file is a "patch file". When it is loaded the system will not complain about function redefinitions. In a patch file, the **defvar** special-form turns into **defconst**; thus patch files will always reinitialize variables.
Cold-Load	A non-nil value for this attribute identifies files that are part of the cold load, the core from which a new system version is built. Certain features that do not work in the cold load check this flag to give an error or a compiler warning if used in such files, so that the problem can be detected sooner.

You are free to define additional file attributes of your own. However, to avoid accidental name conflicts, you should choose names that are different from all the names above, and from any names likely to be defined by anybody else's programs.

The following functions are used to examine file attribute lists:

fs:file-attribute-list *pathname*

Returns the attribute list of the file specified by the pathname. This works on both text files and QFASL files.

fs:extract-attribute-list *stream*

Returns the attribute list read from the specified stream, which should be pointing to the beginning of a file. This works on both text streams and QFASL file binary streams. After the attribute list is read, the stream's pointer is set back to the beginning of the file using the :set-pointer file stream operation (see page 397).

fs:read-attribute-list *pathname stream*

pathname should be a pathname object (*not* a string or namelist, but an actual pathname); usually it is a generic pathname (see section 22.5, page 467). *stream* should be a stream that has been opened and is pointing to the beginning of the file whose file attribute list is to be parsed. The attribute list is read from the stream and then corresponding properties are placed on the specified *pathname*. The attribute list is also returned.

The fundamental way that programs in the Lisp Machine notice the presence of properties on a file's attribute list is by examining the property list in the generic pathname. However, there is another way that is more convenient for some applications. File attributes can cause special variables to be bound whenever Lisp expressions are being read from the file—when the file is being loaded, when it is being compiled, when it is being read from by the editor, and when its QFASL file is being loaded. This is how the Package and Base attributes work. You can also deal with attributes this way, by using the following function:

fs:file-attribute-bindings *pathname*

This function examines the property list of *pathname* and finds all those property names that have fs:file-attribute-bindings properties. Each such property name specifies a set of variables to bind and a set of values to which to bind them. This function returns two values, a list of all the variables and a list of all the corresponding values. Usually you use this function by calling it on a generic pathname that has had fs:read-attribute-list done on it, and then you use the two returned values as the first two arguments of a progv special form (see page 19). Inside the body of the progv the specified bindings will be in effect.

Usually *pathname* is a generic pathname. It can also be a locative, in which case it is interpreted to be the property list itself.

Of the standard attribute names, the following ones have fs:file-attribute-bindings, with the following effects. Package binds the variable package (see page 512) to the package. Base binds the variables base (see page 367) and ibase (see page 371) to the value. Patch-file binds fs:this-is-a-patch-file to the value. Cold-load binds si:file-in-cold-load to the value.

Any properties whose names do not have fs:file-attribute-bindings properties are ignored completely.

You can also add your own attribute names that affect bindings. If an indicator symbol has an fs:file-attribute-bindings property, the value of that property is a function that is called when a file with a file attribute of that name is going to be read from. The function is given three arguments: the file pathname, the attribute name, and the attribute value. It must return two values: a list of variables to be bound and a list of values to bind them to. The function for the Base keyword could have been defined by:

```
(defun (:base file-attribute-bindings) (file ignore bse)
  (if (not (and (typep bse 'fixnum)
                (> bse 1)
                (< bse 37.)))
      (ferror 'fs:invalid-file-attrbute
              "File ~A has an illegal -*- Base:~D -*-"
              file bse))
  (values (list 'base 'ibase) (list bse bse)))
```

fs:invalid-file-attribute (error) *Condition*

An attribute in the file attribute list had a bad value. This is detected within fs:file-attribute-bindings.

21.9.3 File Stream Operations

The following operations may be used on file streams, in addition to the normal I/O operations which work on all streams. Note that several of these operations are useful with file streams that have been closed. Some operations use pathnames; refer to chapter 22, page 453 for an explanation of pathnames.

:pathname *Operation on file streams*

Returns the pathname that was opened to get this stream. This may not be identical to the argument to open, since missing components will have been filled in from defaults. The pathname may have been replaced wholesale if an error occurred in the attempt to open the original pathname.

:truename *Operation on file streams*

Returns the pathname of the file actually open on this stream. This can be different from what :pathname returns because of file links, logical devices, mapping of "newest" version to a particular version number, etc. For an output stream the truename is not meaningful until after the stream has been closed, at least when the file server is an ITS.

:generic-pathname *Operation on file streams*

Returns the generic pathname of the pathname that was opened to get this stream. Normally this is the same as the result of sending the :generic-pathname message to the value of the :pathname operation on the stream; however, it does special things when the Lisp system is bootstrapping itself.

:qfaslp *Operation on file streams*

Returns t if the file has a magic flag at the front that says it is a QFASL file, nil if it is an ordinary file.

:length *Operation on file streams*

Returns the length of the file, in bytes or characters. For text files on PDP-10 file servers, this is the number of PDP-10 characters, not Lisp Machine characters. The numbers are different because of character-set translation; see page 364 for a full explanation. For an output stream the length is not meaningful until after the stream has been closed, at least when the file server is an ITS.

:creation-date *Operation on file streams*

Returns the creation date of the file, as a number that is a universal time. See the chapter on the time package (chapter 31, page 628).

:info *Operation on file streams*

Returns a cons of the file's truename and its creation date. This can be used to tell if the file has been modified between two opens. For an output stream the info is guaranteed to be correct until after the stream has been closed.

:set-byte-size *new-byte-size* *Operation on file streams*

This is only allowed on binary ("fixnum mode") file streams. The byte size can be changed to any number of bits from 1 to 16.

:delete &optional (*error-p* t) *Operation on file streams*

Deletes the file open on this stream. For the meaning of *error-p*, see the deletef function. The file doesn't really go away until the stream is closed.

:undelete &optional (*error-p* t) *Operation on file streams*

If you have used the :deleted option in open to open a deleted file, this operation undeletes the file.

:rename *new-name* &optional (*error-p* t) *Operation on file streams*

Renames the file open on this stream. For the meaning of *error-p*, see the renamef function.

File output streams implement the :finish and :force-output messages.

21.10 Accessing Directories

To understand the functions in this section, it helps to have read the following chapter, on *pathnames*.

fs:directory-list *pathname* &rest *options*

Finds all the files that match *pathname* and returns a list with one element for each file. Each element is a list whose car is the pathname of the file and whose cdr is a list of the properties of the file; thus the element is a disembodied property list and get may be used to access the file's properties. The car of one element is nil; the properties in this element are properties of the file system as a whole rather than of a specific file.

The matching is done using both host-independent and host-dependent conventions. **:wild** as a component of a pathname matches anything; all files that match the remaining components of *pathname* will be listed, regardless of their value for the wild component. In addition, there is host-dependent matching. Typically this uses the asterisk character (*) as a wild-card character. A pathname component that consists of just a * matches any value of that component (the same as **:wild**). A pathname component that contains * and other characters matches any character (on ITS) or any string of characters (on TOPS-20) in the starred positions and requires the specified characters otherwise. In a TOPS-20 pathname, the % character is used to match a single arbitrary character. Other hosts will follow similar but not necessarily identical conventions.

The *options* are keywords which modify the operation. The following options are currently defined:

:noerror If a file-system error (such as no such directory) occurs during the operation, normally an error will be signalled and the user will be asked to supply a new pathname. However, if **:noerror** is specified then, in the event of an error, a string describing the error will be returned as the result of **fs:directory-list**. This is identical to the **:noerror** option to **open**.

:deleted This is for file servers on which deletion is not permanent. It specifies that deleted (but not yet expunged) files are to be included in the directory listing.

:sorted This requests that the directory list be sorted by filenames before it is returned.

The properties that may appear in the list of property lists returned by **fs:directory-list** are host-dependent to some extent. The following properties are those that are defined for both ITS and TOPS-20 file servers. This set of properties is likely to be extended or changed in the future.

:length-in-bytes
The length of the file expressed in terms of the basic units in which it is written (characters in the case of a text file).

:byte-size The number of bits in one of those units.

:length-in-blocks
The length of the file in terms of the file system's unit of storage allocation.

:block-size The number of bits in one of those units.

:creation-date The date the file was created, as a universal time. See chapter 31, page 628.

:reference-date
The most recent date on which the file was used, as a universal time.

:modified-date
The most recent date on which the file's contents were changed, as a universal time.

:author	The name of the person who created the file, as a string.
:not-backed-up	t if the file exists only on disk, nil if it has been backed up on magnetic tape.
:directory	t if this file is actually a directory.
:temporary	t if this file is temporary.
:deleted	t if this file is deleted. Deleted files are included in the directory list only if you specify the :deleted option.
:dont-delete	t indicates that the file is not allowed to be deleted.
:dont-supersede	t indicates that the file may not be superseded; that is, a file with the same name and higher version may not be created.
:dont-reap	t indicates that this file is not supposed to be deleted automatically for lack of use.
:dont-dump	t indicates that this file is not supposed to be dumped onto magnetic tape for backup purposes.
:characters	t indicates that this file contains characters (that is, text). nil indicates that the file contains binary data. This property, rather than the file's byte size, should be used to decide whether it is a text file.
:link-to	If the file is a link, this property is a string containing the name that the link points to.

The element in the directory list that has nil instead of a file's pathname describes the directory as a whole. One property that will usually be found in this element is the :pathname property, whose value is a pathname whose directory is the one you listed. This will usually be the same as the argument to fs:directory-list, plus defaults. But if that directory did not exist, and the user typed another name on the keyboard, this property would reflect the name that was typed.

:physical-volume-free-blocks	This property is an alist in which each element maps a physical volume name (a string) into a number, the number of free blocks on that volume.
:settable-properties	This property is a list of file property names that may be set. This information is provided in the directory list because it is different for different file systems.
:pathname	This property is the pathname from which this directory list was made.
:block-size	This is the number of words in a block in this directory. It can be used to interpret the numbers of free blocks.

fs:directory-list-stream *pathname* &rest *options*

This is like fs:directory-list but returns the information in a different form. Instead of returning the directory list all at once, it returns a special kind of stream which gives out one element of the directory list at a time.

The directory list stream supports two operations: :entry and :close. :entry asks for the next element of the directory stream. :close closes any connection to a remote file server.

The purpose of using fs:directory-list-stream instead of fs:directory-list is that, when communicating with a remote file server, the directory list stream can give you some of the information without waiting for it to all be transmitted and parsed. This is desirable if the directory is being printed on the console.

fs:expunge-directory *pathname* &key &optional (*error* t)

Expunge the directory specified in *pathname*; that is, permanently eliminate any deleted files in that directory. If *error* is nil, there is no error if the directory does not exist.

Note that not all file systems support this function. To find out whether a particular one does, send the :undeletable-p operation to a pathname. If it returns t, the file system of that pathname supports undeletion (and therefore expunging).

fs:create-directory *pathname* &key &optional (*error* t)

Creates the directory specified in *pathname*. If *error* is nil, there is no error if the directory cannot be created; instead an error string is returned. Not all file servers support creation of directories.

fs:remote-connect *pathname* &key &optional (*error* t) *access*

Performs the TOPS-20 "connect" or "access" function, or their equivalents, in a remote file server. *access* specifies which one; "access" is done if it is non-nil.

The "connect" operation grants you full access to the specified directory. The "access" operation grants you whatever access to all files and directories you would have if logged in on the specified directory. Both operations affect access only, since the connected directory of the remote server is never used by transactions from a Lisp Machine.

This function may ask you for a password if one is required for the directory you specify. If the operation cannot be performed, then if *error* is nil, an error string will be returned.

fs:change-file-properties *pathname error-p* &rest *properties*

Some of the properties of a file may be changed; for instance, its creation date or its author. Exactly which properties may be changed depends on the host file system; a list of the changeable property names is the :settable-properties property of the file system as a whole, returned by fs:directory-list as explained above.

fs:change-file-properties changes one or more properties of a file. *pathname* names the file. The *properties* arguments are alternating keywords and values. The *error-p* argument is the same as with renamef; if an error occurs and it is nil a string describing the error will be returned; if it is t a Lisp error will be signalled. If no error occurs, fs:change-file-properties returns t.

fs:file-properties *pathname* &optional (*error-p* t)

Returns a disembodied property list for a single file (compare this to fs:directory-list). The car of the returned list is the truename of the file and the cdr is an alternating list of indicators and values. The *error-p* argument is the same as with renamef; if an error occurs and it is nil a string describing the error will be returned; if it is t (the default) a Lisp error will be signalled.

fs:complete-pathname *defaults string type version* &rest *options*

string is a partially-specified file name. (Presumably it was typed in by a user and terminated with the Altmode key or the End key to request completion.) fs:complete-pathname looks in the file system on the appropriate host and returns a new, possibly more specific string. Any unambiguous abbreviations are expanded out in a host-dependent fashion.

defaults, *type*, and *version* are the arguments that will be given to fs:merge-pathname-defaults (see page 465) when the user's input is eventually parsed and defaulted.

options are keywords (without following values) that control how the completion will be performed. The following option keywords are allowed:

:deleted	Looks for files which have been deleted but not yet expunged.
:read or :in	The file is going to be read. This is the default.
:print or :write or :out	The file is going to be written (i.e. a new version is going to be created).
:old	Looks only for files that already exist. This is the default.
:new-ok	Allows either a file that already exists or a file that does not yet exist. An example of the use of this is the C-X C-F (Find File) command in the editor.

The first value returned is always a string containing a file name, either the original string or a new, more specific string. The second value returned indicates the success or failure of the completion. It is nil if an error occurred. One possible error is that the file is on a file system that does not support completion, in which case the original string will be returned unchanged. Other possible second values are :old, which means that the string completed to the name of a file that exists, :new, which means that the string completed to the name of a file that could be created, and nil again, which means that there is no possible completion.

fs:balance-directories *pathname1 pathname2* &rest *options*

fs:balance-directories is a function for maintaining multiple copies of a directory. Often it is useful to maintain copies of your files on more than one machine; this function provides a simple way of keeping those copies up to date.

The function first parses *pathname1*, filling in missing components with wildcards (except for the version, which is :newest). Then *pathname2* is parsed with *pathname1* as the default. The resulting pathnames are used to generate directory lists using fs:directory-list. Note that the resulting directory lists need not be entire directories; any subset of a directory that fs:directory-list can produce will do.

First the directory lists are matched up on the basis of file name and type. All of the files in either directory list which have both the same name and the same type are grouped together.

The directory lists are next analyzed to determine if the directories are consistent, meaning that two files with the same name and type have equal creation-dates when their versions match, and greater versions have later creation-dates. If any inconsistencies are found, a warning message will be printed on the console.

If the version specified for both *pathname1* and *pathname2* was :newest (the default), then the newest version of each file in each directory will be copied to the other directory if it is not already there. The result will be that each directory will have the newest copy of every file in either of the two directories.

If one or both of the specified versions is not :newest, then *every* file that appears in one directory list and not in the other will be copied. This has the result that the two directories are completely the same. (Note that this is probably not the right thing to use to *copy* an entire directory. Use copy-file with a wildcard argument instead.)

The *options* are keywords which modify the operation. The following options are currently defined:

:ignore	This option takes one argument, which is a list of file names to ignore when making the directory lists. The default value is '().
:error	This option is identical to the :error option to **open**.
:query-mode	This option takes one argument, which indicates whether or not the user should be asked before files are transferred. If the argument is **nil**, no querying is done. If it is ':1->2, then only files being transferred from *pathname2* to *pathname1* will be queried, while if it is ':2->1, then files transferred from *pathname1* to *pathname2* will be queried. If the argument is ':**always**, then the user will be asked about all files.
:copy-mode	This option is identical to the :copy-mode option of **copy-file**, and is used to control whether files are treated as binary or textual data.

21.11 Errors in Accessing Files

fs:file-error (error) *Condition Flavor*

This flavor is the basis for all errors signaled by the file system.

It defines two special operations, :pathname and :operation. Usually, these return the pathname of the file being operated on, and the operation used. This operation was performed either on the pathname object itself, or on a stream.

It defines prompting for the proceed types :retry-file-operation and :new-pathname, both of which are provided for many file errors. :retry-file-operation tries the operation again exactly as it was requested by the program; :new-pathname expects on argument, a pathname, and tries the same operation on this pathname instead of the original one.

fs:file-operation-failure (fs:file-error) *Condition*

This condition name signifies a problem with the file operation requested. It is an alternative to fs:file-request-failure (page 452), which means that the file system was unable to consider the operation properly.

All the following conditions in this section are always accompanied by fs:file-operation-failure, fs:file-error, and error, so they will not be mentioned.

fs:file-open-for-output *Condition*

The request cannot be performed because the file is open for output.

fs:file-locked *Condition*

The file cannot be accessed because it is already being accessed. Just which kinds of simultaneous access are allowed depends on the file system.

fs:circular-link *Condition*

A link could not be opened because it pointed, directly or indirectly through other links, to itself. In fact, some systems report this condition whenever a chain of links exceeds a fixed length.

fs:invalid-byte-size *Condition*

In open, the specified byte size was not valid for the particular file server or file.

fs:no-more-room *Condition*

Processing a request requires resources not available, such as space in a directory, or free disk blocks.

fs:filepos-out-of-range *Condition*

The :set-pointer operation was used with a pointer value outside the bounds of the file.

fs:not-available *Condition*

A requested pack, file, etc. exists but is currently off line or not available to users.

fs:file-lookup-error *Condition*

This condition name categorizes all sorts of failure to find a specified file, for any operation.

fs:device-not-found (fs:file-lookup-error) *Condition*

The specified device does not exist.

fs:directory-not-found (fs:file-lookup-error) *Condition*

The specified directory does not exist.

fs:file-not-found (fs:file-lookup-error) *Condition*

There is no file with the specified name, type and version. This implies that the device and directory do exist, or an error would have happened for them previously.

fs:link-target-not-found (fs:file-lookup-error) *Condition*
The file specified was a link, but the link's target filename fails to be found.

fs:access-error *Condition*
The operation is possible, but the file server is insubordinate and refuses to obey you.

fs:incorrect-access-to-file (access-error). *Condition*
fs:incorrect-access-to-directory (access-error). *Condition*
The file server refuses to obey you because of protection attached to the file (or, the directory).

fs:invalid-wildcard *Condition*
A pathname had a wildcard in a place where the particular file server does not support them. Such pathnames will not be created by pathname parsing, but they can be created with the :new-pathname operation.

fs:wildcard-not-allowed *Condition*
A pathname with a wildcard was used in an operation that does not support it. For example, opening a file with a wildcard in its name.

fs:wrong-kind-of-file *Condition*
An operation was done on the wrong kind of file. If files and directories share one name space and it is an error to open a directory, the error will possess this condition name.

fs:creation-failure *Condition*
An attempt to create a file or directory failed for a reason specifically connected with creation.

fs:file-already-exists (fs:creation-failure) *Condition*
The file or directory to be created already exists.

fs:superior-not-directory (fs:creation-failure fs:wrong-kind-of-file) *Condition*
In file systems where directories and files share one name space, this error results from an attempt to create a file whose "directory" is some other kind of file (not a directory).

fs:delete-failure *Condition*
A file to be deleted exists, but for some reason cannot be deleted.

fs:directory-not-empty (fs:delete-failure) *Condition*
A file could not be deleted because it is a directory and has files in it.

fs:dont-delete-flag-set (fs:delete-failure) *Condition*
A file could not be deleted because its "don't delete" flag is set.

fs:rename-failure *Condition*
A file to be renamed exists, but the renaming could not be done. The :new-pathname operation on the condition instance returns the specified new pathname, which may be a pathname or a string.

fs:rename-to-existing-file (fs:rename-failure) *Condition*
Renaming cannot be done because there is already a file with the specified new name.

fs:rename-across-directories (fs:rename-failure) *Condition*
Renaming cannot be done because the new pathname contains a different device or directory from the one the file is on. This may not always be an error—some file systems support it in certain cases—but when it is an error, it has this condition name.

fs:unknown-property (fs:change-property-failure) *Condition*
A property name specified in a :change-properties operation is not supported by the file server. (Some file servers support only a fixed set of property names.) The :property operation on the condition instance returns the problematical property name.

fs:invalid-property-value (fs:change-property-failure) *Condition*
In a :change-properties operation, some property was given a value that is not valid for it. The :property operation on the condition instance returns the property name, and the :value operation returns the specified value.

fs:invalid-property-name (fs:change-property-failure) *Condition*
In a :change-properties operation, a syntactically invalid property name was specified. This may be because it is too long to be stored. The :property operation on the condition instance returns the property name.

21.12 File Servers

Files on remote file servers are accessed using *file servers* over the Chaosnet. Normally connections to servers are established automatically when you try to use them, but there are a few ways you can interact with them explicitly.

When a file server is first created for you on a particular host, you must tell the server how to log in on that host. This involves specifying a *username*, and, if the obstructionists are in control of your site, a password. The Lisp machine will prompt you for these on the terminal when they are needed.

Logging in a file server is not the same thing as logging in on the Lisp machine (see **login**, page 648). The latter identifies you as a user in general and involves specifying one host, your login host. The former identifies you to a particular file server host and must be done for each host on which you access files. However, logging in on the Lisp machine does specify the username for your login host and logs in a file server there.

The Lisp machine will always try your username (or the part that follows the last period) as a first guess for your password (this happens to take no extra time). If that does not work, you will be asked to type a password, or else a username and a password, on the keyboard. You do not have to give the same user name that you are logged in as, since you may have or use different user names on different machines.

fs:user-unames *Variable*

This is an alist matching host names with the usernames you have specified on those hosts. Each element is the cons of a host object and the username, as a string.

For hosts running ITS. the symbol **fs:its** is used instead of a host object. This is because every user has the same user name on all ITS hosts.

fs:user-host-password-alist *Variable*

Once you have specified a password for a given username and host, it will be remembered for the duration of the session in this variable. The value is a list of elements, each of the form

((*username hostname*) *password*)

All three data are strings.

The remembered passwords are used if more than one file server is needed on the same host, or if the connection is broken and a new file server needs to be created.

If you are very scared of your password being known, you can turn off the recording by setting this variable:

fs:record-passwords-flag *Variable*

Passwords are recorded when typed in if this variable is non-**nil**.

You should set the variable at the front of your init file, and also set the preceding variable to **nil**, since it will already have recorded your password when you logged in.

If you do not use a file server for a period of time, it is killed to save resources on the server host.

fs:host-unit-lifetime *Variable*

This is the length of time after which an idle file server connection should be closed, in 60'ths of a second. The default is 20 minutes.

Some hosts have a caste system in which all users are not equal. It is sometimes necessary to enable one's privileges in order to exercise them. This is done with these functions:

fs:enable-capabilities *host* &rest *capabilities*

Enable the named capabilities on file servers for the specified host. *capabilities* is a list of strings, whose meanings depend on the particular file system that is available on *host*. If *capabilities* is **nil**, a default list of capabilities is enabled; the default is also dependent on the operating system type.

fs:disable-capabilities *host* &rest *capabilities*

Disable the named capabilities on file servers for the specified host. *capabilities* is a list of strings, whose meanings depend on the particular file system that is available on *host*. If *capabilities* is **nil**, a default list of capabilities is disabled; the default is also dependent on the operating system type.

The PEEK system has a mode that displays the status of all your file connections, and of the *host unit* data structures that record them. Clicking on a connection with the mouse gets a menu of operations, of which the most interesting is **reset**. Resetting a host unit may be useful if the connection becomes hung.

21.12.1 Errors in Communication with File Servers

fs:file-request-failure (fs:file-error error) *Condition*

This condition name categorizes errors that prevent the file system from processing the request made by the program.

The following condition names are always accompanied by the more general classifications fs:file-request-failure, fs:file-error, and error.

fs:data-error *Condition*

This condition signifies inconsistent data found in the file system, indicating a failure in the file system software or hardware.

fs:host-not-available *Condition*

This condition signifies that the file server host is up, but refusing connections for file servers.

fs:network-lossage *Condition*

This condition signifies certain problems in the use of the chaosnet by a file server, such as failure to open a data connection when it is expected.

fs:not-enough-resources *Condition*

This condition signifies a shortage of resources needed to consider processing a request, as opposed to resources used up by the request itself. This may include running out of network connections or job slots on the server host. It does not include running out of space in a directory or running out of disk space, because these are resources whose requirements come from processing the request.

fs:unknown-operation *Condition*

This condition signifies that the particular file system fails to implement a standardly defined operation; such as, expunging or undeletion on ITS.