23. Expression Input and Output

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 related functions are known as the *reader*. (Streams are explained in section 22.3, page 459.)

For the rest of the chapter, the phrase 'printed representation' is abbreviated as 'p.r.'

23.1 What the Printer Produces

The printed representation of an object depends on its type. In this section, we consider each type of object and explain how it is printed. There are several variables which you can set before calling the printer to control how certain kinds of objects print. They are mentioned where relevant in this section and summarized in the following section, but one of them is so important it must be described now. This is the *escaping* feature, controlled by the value of ***print-escape***.

Escaping means printing extra syntactical delimiters and *escape characters* when necessary to avoid ambiguity. Without escaping, a symbol is printed by printing the contents of its name; therefore, the symbol whose name consists of the three characters 1, . and 5 prints just like the floating point number 1.5. Escaping causes the symbol to print as |1.5| to differentiate the two. | is a kind of escape character; see page 516 for more information on escape characters and what they mean syntactically.

Escaping also involves printing package prefixes for symbols, printing double-quotes or suitable delimiters around the contents of strings, pathnames, host names, editor buffers, condition objects, and many other things. For example, without escaping, the pathname SYS: SYS; QCP1 LISP prints as exactly those characters. The string with those contents prints indistinguishably. With escaping, the pathname prints as

```
#⊂FS:LOGICAL-PATHNAME "SYS: SYS; QCP1 LISP"⊃
```

and the string prints as "SYS: SYS; QCP1 LISP".

The non-escaped version is nicer looking in general, but if you give it to **read** it won't do the right thing. The escaped version is carefully set up so that **read** will be able to read it in. Printing with escaping is useful in writing expressions into files. Printing without escaping is useful when constructing messages for the user. However, when the purpose of a message printed for the user is to *mention* an object, the object should be printed with escaping:

```
     Your output is in the file SYS: SYS; QCP1 QFASL.
vs
     Expected pathname properties missing from
      #⊂FS:LOGICAL-PATHNAME "SYS: SYS; QCP1 LISP"⊃.
```

The printed representation of an object also may depend on whether Common Lisp syntax is in use. Common Lisp syntax and traditional Zetalisp syntax are incompatible in some aspects of their specifications. In order to print objects so that they can be read back in, the printer needs to know which syntax rules the reader will use. This decision is based on the current readtable: the value of ***readtable*** at the time printing is done.

Now we describe how each type of object is standardly printed.

Integers:

For an integer (a fixnum or a bignum): the printed representation consists of

* a possible radix prefix
* a minus sign, if the number is negative
* the representation of the number's absolute value
* a possible radix suffix.

The radix used for printing the number's absolute value is found as the value of ***print-base***. This should be either a positive fixnum or a symbol with an **si:princ-function** property. In the former case, the number is simply printed in that radix. In the latter case, the property is called as a function with two arguments, minus the absolute value of the number, and the stream to print on. The property is responsible for all printing. If the value of ***print-base*** is unsuitable, an error is signaled.

A radix prefix or suffix is used if either ***nopoint** is **nil** and the radix used is ten, or if ***nopoint** is non-**nil** and ***print-radix*** is non-**nil**. For radix ten, a period is used as the suffix. For any other radix, a prefix of the form *#radix*r is used. A radix prefix or suffix is useful to make sure that **read** parses the number using the same radix used to print it, or for reminding the user how to interpret the number.

Ratios:

The printed representation of a ratio consists of

* a possible radix prefix
* a minus sign, if the number is negative
* the numerator
* a ratio delimiter
* the denominator

If Common Lisp syntax is in use, the ratio delimiter is a slash (/). If traditional syntax is in use, backslash (\) is used. The numerator and denominator are printed according to ***print-base***.

The condition for printing a radix prefix is the same as for integers, but a prefix `#10r` is used to indicate radix ten, rather than a period suffix.

Floating Point Numbers:

* a minus sign, if the number is negative
* one or more decimal digits
* a decimal point
* one or more decimal digits
* an exponent, if the number is small enough or large enough to require one. The exponent, if present, consists of
  * a delimiter, the letter **e**, **s** or **f**
  * a minus sign, if the exponent is negative

    one to three decimal digits

The number of digits printed is just enough to represent all the significant mantissa bits the number has. Feeding the p.r. of a float back to the reader is always supposed to produce an equal float. Floats are always printed in decimal; they are not affected by escaping or by ***print-base***, and there are never any radix prefixes or suffixes.

The Lisp Machine supports two floating point number formats. At any time, one of them is the default; this is controlled by the value of ***read-default-float-format***. When a floating point number whose format is *not* currently the default is printed, it must be printed with an exponent so that the exponent delimiter can specify the format. The exponent is introduced in this case by **f** or **s** to specify the format. To the reader, **f** specifies **single-float** format and **s** specifies **short-float** format.

A floating point number of the default format is printed with no exponent if this looks nice; namely, if this does not require too many extra zeros to be printed before or after the decimal point. Otherwise, an exponent is printed and is delimited with **e**. To the reader, **e** means 'use the default format'.

Normally the default float format is **single-float**. Therefore, the printer may print full size floats without exponents or with **e** exponents, but short floats are always printed with exponents introduced by **s** so as to tell the reader to make a short float.

Complex Numbers:

The traditional printed representation of a complex number consists of

* the real part
* a plus sign, if the imaginary part is positive
* the imaginary part
* the letter i, printed in lower case

If the imaginary part is negative, the + is omitted since the initial - of the imaginary part serves to separate it from the real part.

In Common Lisp syntax, a complex number is printed as #C(*realpart imagpart*); for example, #C(5 3). Common Lisp inexplicably does not allow the more natural 5+3i syntax.

The real and imaginary parts are printed individually according to the specifications above.

Symbols:

If escaping is off, the p.r. is simply the successive characters of the print-name of the symbol. If escaping 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 chapter on packages (chapter 27, page 636) for an explanation of the package name prefix. If the symbol is one which would have another symbol substituted for it if printed normally and read back, such as the symbol **member** printed using Common Lisp syntax which would be replaced with **cli:member** if read in thus, it is printed with a package prefix (e.g., **global:member**) to make it read in properly. See page 519 for more information on this.

If the symbol is uninterned, **#:** is printed instead of a package prefix, provided ***print-gensym*** is non-nil.

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 escape characters are added so as to suppress the other reading. Two kinds of escape characters may be used: single-character escapes and multiple escapes. A single-character escape can be used in front of a character to overrule its special syntactic meaning. Multiple escapes are used in pairs, and all the characters between the pair have their special syntactic meanings suppressed *except single-character escapes*. If the symbol name contains escape characters, they are escaped with single-character escapes. If the symbol name contains anything else problematical, a pair of multiple escape characters are printed around it.

The single-character and multiple escape characters are determined by the current readtable. Standardly the multiple escape character is vertical bar (|), in both traditional and Common Lisp syntax. The single-character escape character is slash (/) in traditional syntax and backslash (\) in Common Lisp syntax.

```
FOO             ;typical symbol, name composed of upper case letters
A/|B            ;symbol with a vertical bar in its name
|Symbol with lower case and spaces in its name|
|One containing slash (//) and vertical bar (/|) also|
```

Except when multiple escape characters are printed, any upper case letters in the symbol's name may be printed as lower case, according to the value of the variable ***print-case***. This is true whether escaping is enabled or not. See the next section for details.

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 printer produces representations such as **(a b (foo bar) c)** in preference to synonymous forms such as **(a . (b . ((foo . (bar . nil)) . (c . nil))))**.

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 ***print-length***, **print** terminates 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 ***print-level***, then the character **#** is printed instead of the list. 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. In Common Lisp read syntax, either one causes **read** to signal an error.

If ***print-pretty*** is non-nil, conses are given to the grinder to print.

If ***print-circle*** is non-nil, a check is made for cars or cdrs that are circular or shared structure, and any object (except for an interned symbol) already mentioned is replaced by a ***#n#*** label reference. See page 524 for more information on them.

```
(let ((*print-circle* t))
  (prin1 (circular-list 3 4)))
```

prints

```
#1= (3 4 . #1#)
```

Character Objects:

When escaping is off, a character object is printed by printing the character itself, with no delimiters.

In Common Lisp syntax, a character object is printed with escaping as *#font\character-or-name*. *font* is the character's font number, in decimal, or is omitted if zero. *character-or-name* begins with prefixes for any modifier bits (control, meta, etc.) present in the character, each followed by a hyphen. Then comes a representation of the character sans font and modifier bits. If this reduced character is a graphic character, it represents itself. Otherwise, it certainly has a standard name; the name is used. If a graphic characters has special syntactic properties (such as whitespace, paretheses, and macro characters) and modifier bit prefixes have been printed then a single-character escape character is printed before it.

In traditional syntax, the p.r. is the similar except that the \ is replaced by =/.

Strings:

If escaping is off, the p.r. is simply the successive characters of the string. If escaping is on, double-quote characters ('"') are printed surrounding the contents, and any single-character escape characters or double-quotes inside the contents are preceded by single-character escapes. If the string contains a Return character followed by an open parenthesis, a single-character escape is printed before the open parenthesis. Examples:

```
"Foo"
"/"Foo/", he said."
```

Named Structures:

If the named structure type symbol has a named-structure-invoke property, the property is called as a function with four arguments: the symbol :print-self, the named structure itself, the stream to print on, and the current *depth* of list structure (see below). It is this function's responsibility to output a suitable printed representation 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 390). Typically the printed representation used starts with either #< if it is not supposed to be readable or #c (see page 527) if it is supposed to be readable.

If the named structure symbol does not have a named-structure-invoke property, the printed-representation depends on whether escaping is in use. If it is, #s syntax is used:

#s(*named-structure-symbol*
  component value
  component value
  ...)

Named structure component values are checked for circular or shared structure if *print-circle* is non-nil.

If escaping is off, the p.r. is like that used for miscellaneous data-types: #<, the named structure symbol, the numerical address of the structure, and >.

Other Arrays:

If *print-array* is non-nil, the array is printed in a way which shows the elements of the array. Bit vectors use #* syntax, other vectors use #(...) syntax, and arrays of rank other than one use #*n*a(...) syntax. The printed representation does not indicate the array type (that is, what elements it is allowed to contain). If the printed representation is read in, a general array (array type art-q) is always created. See page 523 for more information on these syntaxes. Examples:

```
(vector 1 2 5) => #(1 2 5)
(make-array '(2 4) :initial-element t) => #2a((t t t t) (t t t t))
```

Vector and array groupings count like list groupings in maintaining the depth value that is compared with *print-level* for cutting off things that get too deep. More than *print-length* elements in a given vector or array grouping level are cut off with an ellipsis just like a list that is so long.

Array elements are checked for circular or shared structure if *print-circle* is non-nil.

If *print-array* is nil, the p.r. starts with #<. 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 >, as in #<ART-COMPLEX-FLOAT-3-6 34030451>.

Instances and Entities:

If the object says it can handle 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 escaping is enabled. The object should print a suitable p.r. on the stream. See chapter 21, page 401 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 escaping 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 527. If the object cannot handle :print-self, it is printed like "any other data type".

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 in octal. Examples:

```
#'equal => #<DTP-U-ENTRY EQUAL 410>
(value-cell-location nil) => #<DTP-LOCATIVE 1>
```

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 signals 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 :typep)
  (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.

:type This prints the result of (type-of *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 390), or an instance of a flavor (see chapter 21, page 401). 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 543), 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 30.10, page 742).

23.2 Options that Control Printing

Several special variables are defined by the system for the user to set or bind before calling print or other printing functions. Their values, as set up by the user, control how various kinds of objects are printed.

print-escape *Variable*

Escaping is done if this variable is non-nil. See the previous section for a description of the many effects of escaping. Most of the output functions bind this variable to t or to nil, so you rarely use the variable itself.

print-base *Variable*

base *Variable*

The radix to use for printing integers and ratios. The value must be either an integer from 2 to 36 or a symbol with a valid si:princ-function property, such as :roman or :english.

The default value of *print-base* is ten. In input from files, the Base attribute (see section 25.5, page 594) controls the value of *print-base* (and of *read-base*).

The synonym base is from Maclisp.

print-radix *Variable*

If non-nil, integers and ratios are output with a prefix or suffix indicating the radix used to print them. For integers and radix ten, a period is printed as a suffix. Otherwise, a prefix such as #x or #3r is printed. The default value of *print-radix* is nil.

***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 *read-base* is not 10 at the time of reading. The default value of *nopoint is t. *nopoint has no effect if *print-radix* is non-nil.

*nopoint exists for Maclisp compatibility. But to get truly compatible behavior, you must set *nopoint to nil (and, by default, base and ibase to eight).

print-circle *Variable*

If non-nil, the printer recognizes circular and shared structure and prints it using *#n=* labels so that it has a finite printed representation (which can be read back in). The default is nil, since t makes printing slower. See page 524 for information on the *#n=* construct.

print-pretty *Variable*

If non-nil, the printer actually calls grind-top-level so that it prints extra whitespace for the sake of formatting. The default is nil.

print-gensym *Variable*

If non-nil, uninterned symbols are printed with the prefix #: to mark them as such (but only when *print-escape* is non-nil). The prefix causes the reader to construct a similar uninterned symbol when the expression is read. If nil, no prefix is used for uninterned symbols. The default is t.

print-array *Variable*

If non-nil, non-string arrays are printed using the #(...), #* or #*n*a(...) syntax so that you can see their contents (and so that they can be read back in). If nil, such arrays are printed using #<...> syntax and do not show their contents. The default is nil. The printing of strings is not affected by this variable.

print-case *Variable*

Controls the case used for printing upper-case letters in the names of symbols. Its value should be :upcase, :downcase or :capitalize. These mean, respectively, to print those letters as upper case, to print them as lower case, or to capitalize each word (see string-capitalize, page 213). Any lower case letters in the symbol name are printed as lower case and escaped suitably; this flag does not affect them. Note that the case used for printing the upper case letters has no effect on reading the symbols back in, since they are case-converted by read. Any upper case letters that happen to be escaped are always printed in upper case.

```
(dolist (*print-case* '(:upcase :downcase :capitalize))
  (prin1-then-space 'foo)
  (prin1-then-space '|Foo|))
```

prints FOO |Foo| foo |Foo| Foo |Foo| .

print-level *Variable*
prinlevel *Variable*

print-level can be set to the maximum number of nested lists that can be printed before the printer gives up and just prints a # instead of a list element. If it is nil, which it is initially, any number of nested lists can be printed. Otherwise, the value of *print-level* must be a fixnum. Example:

```
(let ((*print-level* 2))
  (prin1 '(a (b (c (d e))))))
```

prints (a (b #)).

The synonym prinlevel is from Maclisp.

print-length *Variable*
prinlength *Variable*

print-length can be set to the maximum number of elements of a list that can be printed before the printer gives up and prints an ellipsis (three periods). If it is nil, which it is initially, any length list may be printed. Example:

```
(let ((*print-length* 3))
  (prin1 '((a b c d) #(e f g h) (i j k l) (m n o p))))
```

prints ((a b c ...) #(e f g ...) (i j k ...) ...).

The synonym prinlength is from Maclisp.

23.3 What The Reader Accepts

The purpose of the reader is to accept characters, interpret them as the p.r. of a Lisp object, and create and return such an object. The reader cannot accept everything that the printer produces; for example, the p.r.'s of 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.

The syntax specified for Common Lisp is incompatible with the traditional Zetalisp syntax. Therefore, the Lisp Machine supports both traditional and Common Lisp syntax, but read must be told in advance which one to use. This is controlled by the choice of readtable (see section 23.6, page 535). When reading input from a file, the Lisp system chooses the syntax according to the file's attribute list: Common Lisp syntax is used if the Common Lisp attribute is present (see section 25.5, page 594).

The main difference between traditional and Common Lisp syntax is that traditionally the single-character escape is slash (/), whereas in Common Lisp syntax it is backslash (\). Thus, the division function which in traditional syntax is written // is written just / in Common Lisp syntax. The other differences are obscure and are mentioned below where they occur.

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.

Escape characters can be used to suppress the special syntactic significance of any character, including :, Space, (or ". There are two kinds of escape character: the *single-character escape* (/ in traditional syntax, \ in Common Lisp syntax) suppresses the significance of the immediately following character; *multiple escapes* (vertical bar, |) are used in pairs, and suppress the special significance of all the characters except escapes between the pair. Escaping a character causes it to be treated as a token constituent and causes the token containing it to be read as a symbol. For example, (12 5 x) represents a list of three elements, two of which are integers, but (/12 5/ x) or (|15| |5 X|) represents a list of two elements, both symbols. Escaping also prevents conversion of letters to upper case, so that |x| is the symbol whose print name contains a lower-case x.

The circle-cross (⊛) character an *octal escape character* 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-cross and the digits in the input stream. This character is always treated as a token constituent and forces the token to be read as a symbol. ⊛ is allowed in both traditional and Common Lisp syntax, but it is not valid Common Lisp.

Integers:

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

Let a *simple integer* 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 integer is interpreted by **read** as an integer. If the trailing decimal point is present, the digits are interpreted in decimal radix; otherwise, they are considered as a number whose radix is the value of the variable ***read-base***.

read-base *Variable*
ibase *Variable*

The value of **ibase** or ***read-base*** is an integer between 2 and 36 that is the radix in which integers and ratios are read. The initial value of is ten. For input from files or editor buffers, the **Base** attribute specifies the value to be used (see section 25.5, page 594); if it is not given, the ambient value is used.

The synonym **ibase** is from Maclisp.

If the input radix is greater than ten, letters starting with **a** are used as additional "digits" with values ten and above. For example, in radix 16, the letters **a** through **f** are digits with values ten through 15. Alphabetic case is not significant. These additional digits can be used wherever a simple integer is expected and are parsed using the current input radix. For example, if ***read-base*** is 16 then **ff** is recognized as an integer (255 decimal). So is **10e5**, which is a float when ***read-base*** is ten.

Traditional syntax also permits a simple integer, followed by an underscore (**_**) or a circumflex (**^**), followed by another simple integer. The two simple integers are interpreted in the usual way; the character in between indicates an operation that is then performed on the two integers. The underscore indicates a binary "left shift"; that is, the integer to its left is doubled the number of times indicated by the integer to its right. The circumflex multiplies the integer to its left by ***read-base*** the number of times indicated by the integer to its right. (The second simple integer is not allowed to have a leading minus sign.) Examples: **3_2** means **12** and **645^3** means **645000.**

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

```
4
23456.
-546
+45^+6      ;means 45000000
2_11        ;4096
72361356126536125376512375126535123712635
-123456789.
105_1000    ;(ash 105 1000) has this value.
105_1000.
```

Floating Point Numbers:

Floats can be written with or without exponent. The syntax for a float without exponent is an optional plus or minus sign, optionally some digits, a decimal point, and one or more digits. A float with exponent consists of a simple integer or a float without exponent, followed by an exponent delimiter (a letter) and a simple integer (the exponent itself) which is the power of ten by which the number is to be scaled. The exponent may not have a trailing decimal point. Both the mantissa and the exponent are always interpreted in base ten, regardless of the value of ***read-base***.

Only certain letters are allowed for delimiting the exponent: **e**, **s**, **f**, **d** and **l**. The case of the letter is not significant. **s** specifies that the number should be a short float; **f**, that it should be a full-size float. **d** or **l** are equivalent to **f**; Common Lisp defines them to mean 'double float' or 'long float', but the Lisp Machine does not support anything longer than a full-size float, so it regards **d** and **l** as synonymous with **f**. **e** tells the reader to use the current default format, whatever it may be, as specified by the value of ***read-default-float-format***.

read-default-float-format *Variable*

The value is the type for read to produce by default for floats whose precise type is not specified by the syntax. The value should be either **global:small-float** or **global:single-float**, these being the only distinct floating formats that the Lisp Machine has. The default is **single-float**, to make full-size floats.

Here are some examples of printed-representations that always read as full-size floats:

```
6.03f23   1F-9      1.f3      3d6
```

Here are some examples of printed-representations that always read as short floats:

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

These read as floats or as a short floats according to ***read-default-float-format***:

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

Rationals:

The syntax for a rational is an integer, a ratio delimiter, and another integer. The integers may not include the ^ and _ scaling characters or decimal points, and only the first one may have a sign. The ratio delimiter is backslash (\) in traditional syntax, slash (/) in Common Lisp syntax. Here are examples:

```
1\2       -100000000000000\3      80\10     traditional
1/2       -100000000000000/3      80/10     Common Lisp
```

Recall that rationals include the integers; **80\10** as input to the reader is equivalent to **8**.

Complex Numbers:

The traditional syntax for a complex number is a number (for the real part), a sign (+ or -), an unsigned number (for the imaginary part), and the letter i. The real and imaginary parts can be any type of number, but they are converted to be of the same type (both floating of the same format, or both rational). For example:

```
1-3\4i
1.2s0+3.45s8i
```

The Common Lisp syntax for a complex number is `#c(`*real imag*`)`, where *real* is the real part and *imag* is the imaginary part. This construction is allowed in traditional syntax too.

```
#c(1 -3/4)
#c(1.2s0 3.45s8)
```

Symbols:

A string of letters, numbers, and characters without special syntactic meaning 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 unless escaped. When the reader sees the p.r. of a symbol, it *interns* it on a *package* (see chapter 27, page 636, for an explanation of interning and the package system). Symbols may start with digits; you could even have one named -345t; **read** accepts 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, they must be escaped. If the symbol's name would look like a number, at least one character in the name must be escaped, but it matters not which one.
Examples of symbols:

```
foo
bar/(baz/)      ; traditional
bar\(baz\)      ; Common Lisp
34w23
|Frob Sale| and F|rob |S|ale| are equivalent

|a/|b|      ; traditional
|a\|b|      ; Common Lisp
```

In Common Lisp syntax, a symbol composed only of two or more periods is not allowed unless escaping is used.

The reader can be directed to perform substitutions on the symbols it reads. Symbol substitutions are used to implement the incompatible Common Lisp definitions of various system functions. Reading of Common Lisp code is done with substitutions that replace **subst** with **cli:subst**, **member** with **cli:member**, and so on. This is why, when a Common Lisp program uses the function **member**, it gets the standard Common Lisp **member** function rather than the traditional one. This is why we say that **cli:member** is "the Common Lisp version of **member**". While **cli:member** can be referred to from any program in just that way, it exists primarily to be referred to from a Common Lisp program which says simply **member**.

Symbol substitutions do not apply to symbols written with package prefixes, so one can use a package prefix to force a reference to a symbol that is normally substituted for, such as using global:member in a Common Lisp program.

Strings:

Strings are written with double-quote characters (") before and after the string contents. To include a double-quote character or single-character escape character in the contents, write an extra single-character escape character in front of it.
Examples of strings:

```
"This is a typical string."
"That is a /"cons cell/"."      ;; traditional
"That is a \"cons cell\"."      ;; Common Lisp
"Strings are often used for I//O."      ;; traditional
"Strings are often used for I/O."       ;; Common Lisp
"Here comes one backslash: \\"          ;; Common Lisp
```

Conses:

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. The following are valid p.r.'s of conses:

```
(foo . bar)
(foo "bar" 33)
(foo . ("bar" . (33 . 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 equivalent to 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.

The reader always allocates new cons cells to represent parentheses. They are never shared with other structure, not even part of the same read. For example,

```
(let ((x (read)))
  (eq (car x) (cdr x)))
((a b) . (a b))          ;; data for read
  => nil
```

because each time (a b) is read, a new list is constructed. 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.

Comments:

A comment begins with a semicolon (;) and continues to the end of the line. Comments are ignored completely by the reader. If the semicolon is escaped or inside a string, it is not recognized as starting a comment; it is part of a symbol or part of the string.

```
;; This is a comment.
"This is a string; but no comment."
```

Another way to write a comment is to start it with `#|` and end it with `|#`. This is useful for commenting out multiple-line segments of code. The two delimiters nest, so that `#| #| |# |#` is a single comment. This prevents surprising results if you use this construct to comment out code which already contains such a comment.

```
(cond ((atom x) y)
 #|
       ((foo x)
        (do-it y))
    |#
       (t (hack y)))
```

Abbreviations:

The single-quote character (') is an abbreviation for a list starting with the symbol quote. The following pairs of p.r.'s produce equal lists:

```
'a    and    (quote a)
'(x (y))    and    (quote (x (y)))
```

The backquote character (`) and comma are used in a syntax that abbreviates calls to the list and vector construction functions. For example,

```
`(a ,b c)
```

reads as a list whose meaning as a Lisp form is equivalent to

```
(list 'a b 'c)
```

See section 18.2.2, page 325 for full details about backquote.

23.3.1 Sharp-sign Constructs

Sharp-sign (`#`) is used to introduce syntax extensions. It is the beginning of a two-character sequence whose meaning depends on the second character. Sharp-sign is only recognized with a special meaning if it occurs at the beginning of a token. If encountered while a token is in progress, it is a symbol constituent. For example, `#xff` is a sharp-sign construct that interprets `ff` as a hexidecimal number, but `1#xff` is just a symbol.

If the sharp-sign is followed by decimal digits, the digits form a parameter. The first non-digit determines which sharp-sign construct is actually in use, and the decimal integer parsed from the digits is passed to it. For example, `#r` means "read in specified radix"; it must actually be used with a radix notated in decimal between the `#` and the `r`, as in `#8r`.

It is possible for a sharp-sign construct to have different meanings in Common Lisp and traditional syntax. The only constructs which differ are #\ and #/.

The function set-dispatch-macro-character (see page 541) can be used to define additional sharp sign abbreviations.

Here are the currently-defined sharp sign constructs:

#/ #/ is used in traditional syntax only to represent the number that is the character code for a character. You can follow the #/ with the character itself, or with the character's name. The name is preferable for nonprinting characters, and it is the only way to represent characters which have control bits since they cannot go in files. Here are examples of #/:

#/a	#o141
#/A	#o101
#/(	#o50
#/c-a	the character code for Control-A
#/c-/a	the character code for Control-a
#/c-sh-a	the character code for Control-a
#/c-/A	the character code for Control-A
#/c-/(	the character code for Control-(
#/return	the character code for Return
#/h-m-system	the character code for Hyper-Meta-System

To represent a printing character, write #/*x* where *x* is the character. For example, #/a is equivalent to #o141 but clearer in its intent. To avoid ambiguity, the character following *x* should not be a letter; good style would require this anyway.

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. Thus, #/A is equivalent to #o101, and #/(is equivalent to #o50. Note that the slash causes this construct to be parsed correctly by the editors Emacs and Zmacs. Even non-printing characters may be used, but for them it is preferable to use the character's name.

To refer to a character by name, write #/ followed by the name. For example, #/return reads as the numeric code for the character Return. The defined character names are documented below (see section 10.1.6, page 211). 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 letters are converted to upper case unless escaped, 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 that #/ uses; therefore, any character name typed out is acceptable as input.

#/ can also be used to read in the names of characters that have modifier bits (Control, Meta, Super and Hyper). 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 the single letters c, m, h and s, and control may be spelled ctrl as it is on the keyboard. 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-/(.

An obsolete method of specifying control bits in a character is to insert the characters α, β, ϵ, π and λ 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.

greek (or front), top, and shift (or sh) are also allowed as prefixes of names. 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.

#\ In traditional syntax, #\ is a synonym for #/. In the past, #/ had to be used before a single character and #\ had to be used in all other cases. Now either one is allowed in either case.

In Common Lisp syntax, #\ produces a character object rather than a fixnum representing a character.

#*/ #*/*x* is the traditional syntax way to produce a character object. It is used just like #/. Thus, Common Lisp #\ is equivalent to traditional syntax #*/.

#^ #^*x* is exactly like #/control-*x* if the input is being read by Zetalisp; it generates 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.

#(#(*elements*...) constructs a vector (rank-one array) of type art-q with elements *elements*. The length of the vector is the number of elements written. Thus, #(a 5 "Foo") reads as a vector containing a symbol, an integer and a string. If a decimal integer appears after the #, it specifies the length of the vector. The last element written is replicated to fill the remaining elements.

#a #*n*a *contents* signifies an array of rank *n*, containing *contents*. *contents* is passed to make-array as the *initial-contents* argument. It is a list of lists of lists... or vector of vectors... as deep as *n*. The dimensions of the array are specified by the lengths of the lists or vectors. The rank is specified explicitly so that the reader can distinguish whether

a list or vector in the contents is a sequence of array elements or a single array element. The array type is always art-q.
Examples:

```
#2a ((x y) (a b) ((uu 3) "VV"))
```

produces a 3 by 2 array. (uu 3) is one of the elements.

```
#2a ("foo" "bar")
```

produces a 2 by 3 array whose elements are character objects. Recall that a string is a kind of vector.

```
#0a 5
```

produces a rank-0 array whose sole element is 5.

#* *#*bbb...* signifies a bit vector; *bbb...* are the bits (characters 1 or 0). A vector of type art-1b is created and filled with the specified bits, the first bit specified going in array element 0. The length is however many bits you specify. Alternatively, specify the length with a decimal number between the # and the *. The last 1 or 0 specified is duplicated to fill the additional bits. Thus, #8*0101 is the same as #*01011111.

#s #s(*type slot value slot value slot value ...*) constructs a structure of type *type*. Any structure type defined with defstruct can be used as *type* provided it has a standard constructor taking slot values as keyword arguments. (Standard constructors can be functions or macros; either kind works for #s.) The slot names and values appearing in the read syntax are passed to the constructor so that they initialize the structure. Example:

```
(defstruct (foo :named)
  bar
  lose)
#s (foo :bar 5 :lose haha)
```

produces a foo whose bar component is 5 and whose lose component is haha.

#= ##

Are used to represent circular structure or shared structure. #*n*= preceding an object "labels" that object with the label *n*, a decimal integer. This has no effect on the way the object labeled is read, but it makes the label available for use in a #*n*# construct within that object (to create circular structure) or later on (to create shared structure). #*n*# counts as an object in itself, and reads as the object labeled by *n*.

For example, #1=(a . #1#) is a way of notating a circular list such as would be produced by (circular-list 'a). The list is labeled with label 1, and then its cdr is given as a reference to label 1. (#1=#:foo #1#) is an example of shared structure. An uninterned symbol named foo is used as the first element of the list, and labeled. The second element of the list is the very same uninterned symbol, by virtue of a reference to the label.

Printing outputs #*n*= and #*n*# to represent circular or shared structure when *print-circle is non-nil.

#, Evaluate a form at load time. #, *foo* evaluates *foo* (the p.r. of a Lisp form) at read time, except that during file-to-file compilation 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:

```
#'(rename-file ,("foo" "bar") ,("ofoo" "obar"))
```

expands into

```
(progn (rename-file "foo" "ofoo")
       (rename-file "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 *read-base*. Actually, any expression can be prefixed by #o; it is read with *read-base* bound to 8.

#b Like #o but reads in binary.

#x Like #x but reads in radix 16 (hexadecimal). The letters a through f are used as the digits beyond 9.

#r *#radixr number* reads *number* in radix *radix* regardless of the setting of *read-base*. As with #o, any expression can be prefixed by *#radixr*; it is read with *read-base* bound to *radix*. *radix* must be a valid decimal integer between 2 and 36.

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.

#c *#c(real imag)* constructs a complex number with real part *real* and imaginary *part*. It is equivalent to *real* + *imag*i, except that #c is allowed in Common Lisp syntax and the other is not.

#+ This abbreviation provides a read-time conditionalization facility. It is used as *#+feature form*. If *feature* is a symbol, then this is read as *form* if *feature* is present in the list *features* (see page 803). Otherwise, the construct is regarded as whitespace.

Alternately, *feature* may be a boolean expression composed of and, or, and not operators and symbols representing items that may appear on *features*. Thus, #+(or lispm amber) causes the following object to be seen if either of the features lispm or amber is present.

For example, #+lispm *form* makes *form* count if being read by Zetalisp, and is thus equivalent to #q *form*. Similarly, #+maclisp *form* is equivalent to #m *form*. #+(or lispm nil) *form* makes *form* count on either Zetalisp or in NIL.

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. May you be spared the dishonor of using it.

#+, and the other read-time conditionalization constructs that follow, discard the following expression by reading it with *read-suppress* bound to t if the specified condition is false.

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

#q #q *foo* reads as *foo* if the input is being read by Zetalisp, otherwise it reads as nothing (whitespace). This is considered obsolete; use #+lispm instead.

#m #m *foo* reads as *foo* if the input is being read into Maclisp, otherwise it reads as nothing (whitespace). This is considered obsolete; use #+maclisp instead.

#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 is considered obsolete; use #+nil instead.

#◇ #◇ introduces an expression in infix notation. ◇ should be used to terminate it. The text in between describes a Lisp object such as a symbol, number or list but using a nonstandard, infix-oriented syntax. For example,

```
#◇x:y+car(a1[i,j])◇
```

is equivalent to

```
(setq x (+ y (car (aref a1 i j))))
```

It is not strictly true that the Lisp object produced in this way has to be an expression. Since the conversion is done at read time, you can use a list expressed this way for any purpose. But the infix syntax is designed to be used for expressions.

For full details, refer to the file SYS: IO1; INFIX LISP.

#< 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 #< signals 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. si:print-readably-mixin is a useful implementation the :read-instance operation for general purposes; see page 446.

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 signals 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 type-of 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.

#| #| is used to comment out entire pieces of code. Such a comment begins with #| and ends with |#. The text in between should be one or more properly balanced p.r.'s of Lisp objects, possibly including nested #|...|# comments. This text is skipped over by the reader, and does not contribute to the value returned by read.

23.4 Expression 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). 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 22.3, page 459.

prin1 *object* &optional *stream*

Outputs the printed representation of *object* to *stream*, with escaping (see page 506). *object* is returned.

prin1-then-space *object* &optional *stream*
Like prin1 except that output is followed by a space.

print *object* &optional *stream*
Like prin1 except that output is preceded by a carriage return and followed by a space. *object* is returned.

princ *object* &optional *stream*
Like prin1 except that the output is not escaped. *object* is returned.

write *object* &key *stream escape radix base circle pretty level length case gensym array*
Prints *object* on *stream*, having bound all the printing flags according to the keyword arguments if specified. For example, the keyword argument *array* specifies how to bind *print-array*; if *array* is omitted, the ambient value of *print-array* is used. This function is sometimes cleaner than binding a printing control variable explicitly. The value is *object*.

23.4.1 Pretty-Printing Output Functions

pprint *object* &optional *stream*
pprint is like prin1 except that *print-pretty* is bound to t so that the grinder is used. pprint returns zero values, just as the form (values) does.

grindef *function-spec...* *Macro*
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 says so and tries to find its previous interpreted definition by looking on an associated property list (see uncompile (page 301). This works only if the function's interpreted definition was once in force; if the definition of the function was simply loaded from a QFASL file, grindef cannot not find the interpreted definition.

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 tries 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 are 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 should be a function that to be called with three arguments for each "token" in the pretty-printed output. Tokens are atoms, open and close parentheses, and reader macro characters such as '. The arguments given 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*. 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.

23.4.2 Non-Stream Printing Functions

write-to-string *object* &key *escape radix base circle pretty level length case gensym array*

prin1-to-string *object*

princ-to-string *object*

Like write, prin1 and princ, respectively, but put the output in a string and return the string (see page 528).

See also the with-output-to-string special form (page 474).

The following obsolete functions are for Maclisp compatibility only. The examples use traditional syntax.

exploden *object*

Returns a list of characters (represented as fixnums) that are the characters that would be typed out by (princ *object*) (i.e. the unescaped printed representation of *object*).

Example:

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

explodec *object*

Returns a list of characters represented by symbols, interned in the current package, whose names are the characters that would be typed out by (princ *object*) (i.e. the unescaped printed representation of *object*).

Example:

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

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

explode *object*

Like explodec but uses the escaped printed representation.

Example:

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

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

flatsize *object*

Returns the number of characters in the escaped printed representation of *object*.

flatc *object*

Returns the number of characters in the unescaped printed representation of *object*.

23.5 Expression Input Functions

Most expression input functions read characters from an input stream. This argument is called *stream*. If unsupplied it defaults to the value of *standard-input*.

All of these functions echo their input and permit editing if used on an interactive stream (one which supports the :rubout-handler operation; see below.)

The functions accept an argument *eof-option* or two arguments *eof-error* and *eof-value* to tell them what to do if end of file is encountered instead of an object's p.r. The functions that take two *eof-* arguments are the Common Lisp ones.

In functions that accept the *eof-option* argument, if no argument is supplied, an error is signaled at eof. If the argument is supplied, end of file causes the function to return that argument. 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*.

In functions that accept two arguments *eof-error* and *eof-value*, end of file is an error if *eof-error* is non-nil or if it is unsupplied. If *eof-error* is nil, then the function returns *eof-value* at end of file.

An error is always signaled if end of file is encountered in the middle of an object; for example, if a file does not contain enough right parentheses to balance the left parentheses in it. Mere whitespace does not count as starting an object. If a file contains a symbol or a number immediately followed by end-of-file, it can be read normally without error; if an attempt is made to read further, end of file is encountered immediately and the *eof-* argument(s) obeyed.

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.

For Maclisp compatibility, nil as the *stream* argument also means to use the value of *standard-input*, and t as the *stream* argument means to use the value of *terminal-io*. This is only advertised to work in functions that Maclisp has, and should not be written in new programs. Instead of the variable *standard-input* Maclisp has several variables and complicated rules. For detailed documentation of streams, refer to section 22.3, page 459.

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 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.

read &optional *stream eof-option rubout-handler-options*

Reads the printed representation of a Lisp object from *stream*, builds a corresponding Lisp object, and returns the object. *rubout-handler-options* are used as options for the rubout handler, if *stream* supports one; see section 22.5, page 500 for more information on this.

cli:read &optional *stream* (*eof-errorp* t) *eof-value recursive-p*

The Common Lisp version of read differs only in how its arguments are passed.

recursive-p should be non-nil when calling from the reader or from the defining function of a read-macro character; that is, when reading a subexpression as part of the task of reading a larger expression. This has two effects: the subexpression is allowed to share *#n#* labels with the containing expression, and whitespace which terminates the subexpression (if it is a symbol or number) is not discarded.

read-or-end &optional *stream eof-option rubout-handler-options*

Like read, but on an interactive stream if the input is just the character End it returns the two values nil and :end.

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 throws away the delimiting character if it is whitespace, but preserves it (using the :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, the delimiting character is never thrown away, even if it is whitespace. This may be useful for certain reader macros or special syntaxes.

read-preserving-whitespace &optional *stream* (*eof-errorp* t) *eof-value recursive-p*
Like cli:read but binds read-preserve-delimiters to t. This is the Common Lisp way of requesting the read-preserve-delimiters feature.

read-delimited-list *char* &optional *stream recursive-p*
Reads expressions from *stream* until the character *char* is seen at top level when an expression is expected; then returns a list of the objects read. *char* may be a fixnum or a character object. For example, if *char* is #/], and the text to be read from *stream* is a (b c)] ... then the objects a and (b c) are read, the] is seen as a terminator and discarded, and the value returned is (a (b c)). *recursive-p* is as for cli:read. End of file within this function is always an error since it is always "within an object"—the object whose textual representation is terminated by *char*.

Note that use of this function does not cause *char* to terminate tokens. Usually you want that to happen, but it is purely under the control of the readtable. So you must modify the readtable to make this so. The usual way is to define *char* as a macro character whose defining function just signals an error. The defining function is not called when *char* is encountered in the expected context by read-delimited-list; if *char* is encountered anywhere else, it is an unbalanced bracket and an error is appropriate.

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.

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.

23.5.1 Non-Stream Parsing Functions

The following functions do expression input but get the characters from a string or a list instead of a stream.

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 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 in **read**. *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 is the length of the string.

Example:

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

cli:read-from-string *string* &optional (*eof-errorp* t) *eof-value* &key (*start* 0) *end* *preserve-whitespace*

The Common Lisp version of **read-from-string** uses a different calling convention. The arguments mean the same thing but are arranged differently. There are three arguments with no counterparts: *eof-errorp* and *eof-value*, which are simply passed on to **cli:read**, and *preserve-whitespace*, which if non-**nil** means that the reading is done with **read-preserve-delimiters** bound to **t**.

See also the **with-input-from-string** special form (page 473).

parse-integer *string* &key (*start* 0) *end* (*radix* 10.) *junk-allowed*

Parses the contents of *string* (or the portion from *start* to *end*) as a numeral for an integer using the specified radix, and returns the integer. Radices larger than ten are allowed, and they use letters as digits beyond 9. Leading whitespace is always allowed and ignored. A leading sign is also allowed and considered part of the number.

When *junk-allowed* is **nil**, the entire specified portion of string must consist of an integer and leading and trailing whitespace. Otherwise, an error happens.

If *junk-allowed* is non-**nil**, parsing just stops when a non-digit is encountered. The number parsed so far is returned as the first value, and the index in *string* at which parsing stopped is returned as the second value. This number equals *end* (or the length of *string*) if there is nothing but a number. If non-digits are found without finding a number first, the first value is **nil**. Examples:

```
(parse-integer " 1A " :radix 16.) => 26.
(parse-integer " 15X " :end 3) => 15.
(parse-integer " -15X " :junk-allowed t) => -15. 3
(parse-integer " 15X ") => error!
```

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: character objects, 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 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, some kind of sys:read-end-of-file error is signaled.

23.5.2 Input Error Conditions

sys:read-error (sys:parse-error error) *Condition*

This condition name classifies all errors detected by the reader per se. Since sys:parse-error is implied, all sys:read-error errors must provide the proceed type :no-action so that automatic proceed is possible if the error happens during compilation. See page 505.

Since this condition name implies sys:parse-error and error, those two are not mentioned as implications below when sys:read-error is.

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

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

Since sys:end-of-file is implied, the :stream operation on the condition instance returns the stream on which end of file was reached.

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

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 *Condition*

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

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

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

Proceed type :no-action terminates the string and returns. If the string is within other constructs that are unterminated, another end of file error is will be signaled later.

sys:read-symbol-end-of-file *Condition*

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

This is signaled when read detects end of file within a multiple escape construct.

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

Proceed type :no-action terminates the symbol and returns. If the symbol is within other constructs that are unterminated, another end of file error is will be signaled later.

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.

23.6 The Readtable

The syntax used by the reader is controlled by a data structure called the *readtable*. (Some aspects of printing are also controlled by the readtable.) There can be many readtables, but the one that is used is the one which is the value of *readtable*. A particular syntax can be selected for use by setting or binding *readtable* to a readtable which specifies that syntax before reading or printing. In particular, this is how Common Lisp or traditional syntax is selected. The readtable also controls the symbol substitutions which implement the distinction between the traditional and Common Lisp versions of functions such as subst, memberand defstruct.

The functions in this section allow you to modify the syntax of individual characters in a readtable in limited ways. You can also copy a readtable; then you can modify one copy and leave the other unchanged.

A readtables may have one or more names. Named readtables are recorded in a central data base so that you can find a readtable by name. When you copy a readtable, the new one is anonymous and is not recorded in the data base.

readtable *Variable*
readtable *Variable*

The value of readtable or *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.

The two names are synonymous.

si:standard-readtable *Constant*

This is copied into *readtable* every time the machine is booted. Therefore, it is normally the same as *readtable* unless you make *readtable* be some other readtable. If you alter the contents of *readtable* without setting or binding it to some other readtable, this readtable is changed.

si:initial-readtable *Constant*

The value of si:initial-readtable is a read-only copy of the default current readtable. Its purpose is to preserve a copy of the standard read syntax in case you modify the contents of *readtable* and regret it later. You could use si:initial-readtable as the *from-readtable* argument to copy-readtable or set-syntax-from-char to restore all or part of the standard syntax.

si:common-lisp-readtable *Constant*

A readtable which initially is set up to define Common Lisp read syntax. Reading of Common Lisp programs is done using this readtable.

si:initial-common-lisp-readtable *Constant*

A read-only copy of si:common-lisp-readtable, whose purpose is to preserve a copy of the standard Common Lisp syntax in case you modify si:common-lisp-readtable (such as, by reading a Common Lisp program which modifies the current readtable).

si:rdtbl-names *readtable*

Returns the list of names of readtable. You may setf this to add or remove names.

si:find-readtable-named *name*

Returns the readtable named name, or nil if none is recorded.

readtablep *object*

t if *object* is a readtable.

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 *from-readtable* is nil, the standard Common Lisp syntax is copied. If *to-readtable* is unsupplied or nil, a fresh copy is made. Otherwise *to-readtable* is clobbered with the copied syntax.

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*

Copies the syntax of *from-char* in *from-readtable* to character *to-char* in *to-readtable*. *to-readtable* defaults to the current readtable and *from-readtable* defaults to si:initial-standard-readtable (standard traditional syntax).

cli:set-syntax-from-char *to-char from-char* &optional *to-readtable from-readtable*

Is a Common Lisp function which copies the syntax of *from-char* in *from-readtable* to character *to-char* in *to-readtable*. *to-readtable* defaults to the current readtable and *from-readtable* defaults to si:initial-common-lisp-readtable (standard Common Lisp syntax).

Common Lisp has a peculiar idea of what it means to copy the syntax of a character. The only aspect of syntax that the readtable supposedly specifies is the choice among

* token constituent: digits, letters, random things like @, !, $, and also colon!
* whitespace: spaces, Tab, Return.
* single escape character: / traditionally, \ in Common Lisp.
* multiple escape character: vertical-bar.
* macro character: standardly ()",.';
* nonterminating macro character: # is the only such character standardly defined.

The differences among macro characters are determined entirely by the functions that they invoke. The differences among token constituents (including the difference between A and colon) are fixed! You can make A be a macro character, or whitespace, or a quote character, but if you make it a token constituent then it always behaves the way it normally does. You can make colon be a macro character, or whitespace, etc., but if it is a token constituent it always delimits package names. If you make open-parenthesis into a token constituent, there is only one kind of token constituent it can be (it forces the token to be a symbol, like $ or @ or %).

This is not how Lisp Machine readtables really work, but since cli:set-syntax-from-char is provided just for Common Lisp, the behavior specified by Common Lisp is laboriously provided. So, if *from-char* is some kind of token constituent, this function makes *to-char* into a token constituent of the kind that *to-char* is supposed to be—not the kind of token constituent that *from-char* is.

By contrast, the non-Common-Lisp set-syntax-from-char would make *to-char* have exactly the same syntactic properties that *from-char* has.

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 translation is inhibited by escape characters and within strings. *readtable* defaults to the current readtable.

The following syntax-setting functions are more or less obsolete.

set-syntax-from-description *char description* &optional *readtable*
Sets the syntax of *char* in *readtable* to be that described by the symbol *description*. *readtable* defaults to the current readtable.

Each readtable has its own set of descriptions which it defines. 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:escape — The character quoter. In the initial readtable this is '/'.

si:multiple-escape — The symbol print-name quoter. In the initial readtable this is '|'.

si:macro — A macro character. Don't use this; use set-macro-character (page 540).

si:non-terminating-macro
A macro character recognized only at the start of a token. In the initial readtable, '#' is such a character. (It is also a dispatching macro, but that is another matter.) The correct way to make a character be a macro is with set-macro-character.

si:character-code-escape
The octal escape for special characters. In the initial readtable this is '⊛'.

si:digitscale — a character for shifting an integer by digits. In the initial readtable this is '^'.

si:bitscale — A character for shifting an integer by bits. In the initial readtable this is '_' (underscore).

si:slash
si:circlecross — Obsolete synonyms for si:escape and si:character-code-escape.

Unfortunately it is no longer possible to provide si:doublequote as double-quote is now an ordinary macro character.

These symbols may be moved to the keyword package at some point.

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.

23.7 Read-Macro Characters

A *read-macro character* (or just *macro character*) is a character whose syntax is defined by a function which the reader calls whenever that character is seen (unless it is escaped). This function can optionally read additional characters and is then responsible for returning the object which they represent.

The standard meanings of the characters open-parenthesis, semicolon, single-quote, double-quote, #, backquote (`) and comma are implemented by making them macro characters.

For example, open-parenthesis is implemented as a macro character whose defining function reads expressions until a close-parenthesis is found, throws away the close-parenthesis, and returns a list of the expressions read. (It actually must be more complicated than this in order to deal properly with dotted lists and with indentation checking.) Semicolon is implemented as a macro character whose defining function swallows characters until a Return and then returns no values.

Close-parenthesis and close-horseshoe (⊃) are also macro characters so that they will terminate symbols. Their defining functions signal errors if actually called; but when these delimiters are encountered in their legitimate contexts they are recognized and handled specially before the defining function is called.

The user can also define macro characters.

When a macro's defining function is called, it receives two arguments: the input stream, and the macro character being handled. The function may read characters from the stream, and should return zero or more values, which are the objects that the macro construct "reads as". Zero values causes the macro construct to be ignored (the semicolon macro character does this), and one value causes the macro construct to read as a single object (most macro characters do this). More than one value is allowed only within a list.

Macro characters may be *terminating* or *non-terminating*. A non-terminating macro character is only recognized as a macro character when it appears at the beginning of a token. If it appears when a token is already in progress, it is treated as a symbol constituent. Of the standard macro characters, all but # are terminating.

One kind of macro character is the *dispatch* macro character. This kind of character is handled by reading one more character, converting it to upper case, and looking it up in a table. Thus, the dispatch macro character is the start of a two-character sequence, with which is associated a defining function. # is the only standardly defined dispatch macro character.

When a dispatch macro character is used, it may be followed by a decimal integer which serves as a parameter. The character for the dispatch is actually the first non-digit seen.

The defining function for a dispatch macro two-character sequence is almost like that of an ordinary macro character. However, it receives one more argument. This is the parameter, the decimal integer that followed the dispatch macro character, or nil if no parameter was written. Also, the second argument is the subdispatch character, the second character of the sequence. The dispatch macro character itself is not available.

set-macro-character *char function* &optional *non-terminating-p in-readtable*

Sets the syntax of character *char* in readtable *in-readtable* to be that of a macro character which is handled by *function*. When that character is read by **read**, *function* is called.

char is made a non-terminating macro character if *non-terminating-p* is non-nil, a terminating one otherwise.

get-macro-character *char in-readtable*

Returns two values that describe the macro character status of *char* in *in-readtable*. If *char* is not a macro character, both values are nil. Otherwise, the first value is the *function* and the second value is the *non-terminating-p* for this character.

Those two values, passed to **set-macro-character**, are usually sufficient to recreate exactly the syntax *char* has now; however, since one of the arguments that the function receives is the macro character that invoked it, it may not behave the same if installed on a different character or in a different readtable. In particular, the definition of a dispatch macro character is a standard function that looks the macro character up in the readtable. Thus, the definition only records that the macro character *is* a dispatch macro character; it does not say what subcharacters are allowed or what they mean.

make-dispatch-macro-character *char* &optional *non-terminating-p in-readtable*
Makes *char* be a dispatch macro character in *in-readtable*. This means that when *char* is seen read will read one more character to decide what to do. # is an example of a dispatch macro character. *non-terminating-p* means the same thing as in set-macro-character.

set-dispatch-macro-character *char subchar function* &optional *in-readtable*
Sets the syntax of the two-character sequence *char subchar*, assuming that *char* is already a dispatch macro character. *function* becomes the defining function for this sequence.

If *subchar* is lower case, it is converted to upper case. Case is never significant for the character that follows a dispatch macro character. The decimal digits may not be defined as subchars since they are always used for infix numeric arguments as in #5r.

get-dispatch-macro-character *char subchar* &optional *in-readtable*
Returns the *function* for *subchar* following dispatch macro character *char* in readtable *in-readtable*. The value is nil if *subchar* is not defined for following *char*.

These subroutines are for use by the defining functions of macro characters. Ordinary read should not be used for reading subexpressions, and the ordinary :tyi operation or functions read-char or tyi should not be used for single-character input. The functions below should be used instead.

si:read-recursive &optional *stream*
Equivalent to (cli:read *stream* t nil t). See page 531. This is the recommended way for a macro character's defining function to read a subexpression.

si:xr-xrtyi *stream ignore-whitespace no-chars-special no-multiple-escapes*
Reads the next input character from *stream*, for a macro character's defining function. If *ignore-whitespace* is non-nil, any whitespace characters seen are discarded and the first non-whitespace character is returned.

The first value is the character as translated; the third value is the original character, before translation. The second value is a syntax code which is of no interest to users except to be passed to si:xr-xruntyi if this character must be unread.

Normally, this function processes all the escape characters, and performs translations (such as from lower case letters to upper case letters) on characters not escaped. Font specifiers (epsilons followed by digits or *) are ignored if the file is formatted using them.

If *no-multiple-escapes* is non-nil, multiple escapes (vertical bar characters) are not processed; they are returned to the caller. This mode is used for reading the contents of strings. If *no-chars-special* is non-nil, no escape characters are processed. All characters are simply returned to the caller (except that font specifiers are still discarded if appropriate).

si:xr-xruntyi *stream char num*

Unreads *char*, for a macro character's defining function. *char* should be the third value returned by the last call to si:xr-xrtyi, and *num* should be the second value.

read-suppress *Variable*

If this variable is non-nil, all the standard read functions and macro characters do their best to avoid any errors, and any side effects except for removing characters from the input stream. For example, symbols are not interned to avoid either errors (for nonexistent packages) or side effects (adding new symbols to packages). In fact, nil is used in place of any symbol that is written.

User macro characters should also notice this variable when appropriate.

The purpose of the variable is to allow expressions to be skipped and discarded. The read-time conditional constructs #+ and #- bind it to t to skip the following expression if it is not wanted.

The following functions for defining macro characters are more or less obsolete.

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.

Note that the convention for values returned by *function* is different from that used for functions specified in set-macro-character, above. set-syntax-macro-char works by encapsulating *function* in a closure to convert the values to the sort that set-macro-character wants and then passing the closure to set-macro-character.

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*.

setsyntax-sharp-macro *character type function* &optional *readtable*
This exists only for Maclisp compatibility. set-dispatch-macro-character should be used instead. 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*.

23.8 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 *depth* (for comparison against the *print-level* variable), and *escape-p* (which is the value of *print-escape*). 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 depth stream* &optional *which-operations*
Outputs the printed-representation of *object* to *stream*, as modified by *depth* and the values of the *print-... variables.

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 30.10, page 742) this function is the way to do it. See section 23.1, page 513.

si:print-list *list depth 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.