22. Printer

22.1 The Lisp Printer

22.1.1 Overview of The Lisp Printer

Common Lisp provides a representation of most *objects* in the form of printed text called the printed representation. Functions such as **print** take an *object* and send the characters of its printed representation to a *stream*. The collection of routines that does this is known as the (Common Lisp) printer.

Reading a printed representation typically produces an *object* that is **equal** to the originally printed *object*.

22.1.1.1 Multiple Possible Textual Representations

Most *objects* have more than one possible textual representation. For example, the positive *integer* with a magnitude of twenty-seven can be textually expressed in any of these ways:

```
 27    27.    #o33    #x1B    #b11011    #.(* 3 3 3)    81/3
```

A list containing the two symbols `A` and `B` can also be textually expressed in a variety of ways:

```
 (A B)    (a b)    (  a  b )    (\A |B|)
(|\A|
  B
)
```

In general, from the point of view of the *Lisp reader*, wherever *whitespace* is permissible in a textual representation, any number of *spaces* and *newlines* can appear in *standard syntax*.

When a function such as **print** produces a printed representation, it must choose from among many possible textual representations. In most cases, it chooses a program readable representation, but in certain cases it might use a more compact notation that is not program-readable.

A number of option variables, called *printer control variables*, are provided to permit control of individual aspects of the printed representation of *objects*. The next figure shows the *standardized printer control variables*; there might also be *implementation-defined printer control variables*.

```
*print-array*   *print-gensym*       *print-pprint-dispatch*
*print-base*    *print-length*       *print-pretty*
*print-case*    *print-level*        *print-radix*
*print-circle*  *print-lines*        *print-readably*
*print-escape*  *print-miser-width*  *print-right-margin*
```

Figure 22-1. Standardized Printer Control Variables

In addition to the *printer control variables*, the following additional *defined names* relate to or affect the behavior of the *Lisp printer*:

```
*package*                    *read-eval*  readtable-case
*read-default-float-format*  *readtable*
```

Figure 22-2. Additional Influences on the Lisp printer.

22.1.1.1.1 Printer Escaping

The *variable* ***print-escape*** controls whether the *Lisp printer* tries to produce notations such as escape characters and package prefixes.

The *variable* ***print-readably*** can be used to override many of the individual aspects controlled by the other *printer control variables* when program-readable output is especially important.

One of the many effects of making the *value* of ***print-readably*** be *true* is that the *Lisp printer* behaves as if ***print-escape*** were also *true*. For notational convenience, we say that if the value of either ***print-readably*** or ***print-escape*** is *true*, then *printer escaping* is "enabled"; and we say that if the values of both ***print-readably*** and ***print-escape*** are *false*, then *printer escaping* is "disabled".

22.1.2 Printer Dispatching

The *Lisp printer* makes its determination of how to print an *object* as follows:

If the *value* of ***print-pretty*** is *true*, printing is controlled by the *current pprint dispatch table*; see Section 22.2.1.4 (Pretty Print Dispatch Tables).

Otherwise (if the *value* of ***print-pretty*** is *false*), the object's **print-object** method is used; see Section 22.1.3 (Default Print-Object Methods).

22.1.3 Default Print-Object Methods

This section describes the default behavior of **print-object** methods for the *standardized types*.

22.1.3.1 Printing Numbers

22.1.3.1.1 Printing Integers

Integers are printed in the radix specified by the *current output base* in positional notation, most significant digit first. If appropriate, a radix specifier can be printed; see ***print-radix***. If an *integer* is negative, a minus sign is printed and then the absolute value of the *integer* is printed. The *integer* zero is represented by the single digit `0` and never has a sign. A decimal point might be printed, depending on the *value* of ***print-radix***.

For related information about the syntax of an *integer*, see Section 2.3.2.1.1 (Syntax of an Integer).

22.1.3.1.2 Printing Ratios

Ratios are printed as follows: the absolute value of the numerator is printed, as for an *integer*; then a `/`; then the denominator. The numerator and denominator are both printed in the radix specified by the *current output base*; they are obtained as if by **numerator** and **denominator**, and so *ratios* are printed in reduced form (lowest terms). If appropriate, a radix specifier can be printed; see ***print-radix***. If the ratio is negative, a minus sign is printed before the numerator.

For related information about the syntax of a *ratio*, see Section 2.3.2.1.2 (Syntax of a Ratio).

22.1.3.1.3 Printing Floats

If the magnitude of the *float* is either zero or between 10^-3 (inclusive) and 10^7 (exclusive), it is printed as the integer part of the number, then a decimal point, followed by the fractional part of the number; there is always at least one digit on each side of the decimal point. If the sign of the number (as determined by **float-sign**) is negative, then a minus sign is printed before the number. If the format of the number does not match that specified by ***read-default-float-format***, then the *exponent marker* for that format and the digit `0` are also printed. For example, the base of the natural logarithms as a *short float* might be printed as `2.71828S0`.

For non-zero magnitudes outside of the range 10^-3 to 10^7, a *float* is printed in computerized scientific notation. The representation of the number is scaled to be between 1 (inclusive) and 10 (exclusive) and then printed, with one digit before the decimal point and at least one digit after the decimal point. Next the *exponent marker* for the format is printed, except that if the format of the number matches that specified by ***read-default-float-format***, then the *exponent marker* `E` is used. Finally, the power of ten by which the fraction must be multiplied to equal the original number is printed as a decimal integer. For example, Avogadro's number as a *short float* is printed as `6.02S23`.

For related information about the syntax of a *float*, see Section 2.3.2.2 (Syntax of a Float).

22.1.3.1.4 Printing Complexes

A *complex* is printed as `#C`, an open parenthesis, the printed representation of its real part, a space, the printed representation of its imaginary part, and finally a close parenthesis.

For related information about the syntax of a *complex*, see Section 2.3.2.3 (Syntax of a Complex) and Section 2.4.8.11 (Sharpsign C).

22.1.3.1.5 Note about Printing Numbers

The printed representation of a number must not contain *escape characters*; see Section 2.3.1.1.1 (Escape Characters and Potential Numbers).

22.1.3.2 Printing Characters

When *printer escaping* is disabled, a *character* prints as itself; it is sent directly to the output *stream*. When *printer escaping* is enabled, then `#\` syntax is used.

When the printer types out the name of a *character*, it uses the same table as the `#\` *reader macro* would use; therefore any *character* name that is typed out is acceptable as input (in that *implementation*). If a *non-graphic character* has a *standardized name*[5], that *name* is preferred over non-standard *names* for printing in `#\` notation. For the *graphic standard characters*, the *character* itself is always used for printing in `#\` notation---even if the *character* also has a *name*[5].

For details about the `#\` *reader macro*, see Section 2.4.8.1 (Sharpsign Backslash).

22.1.3.3 Printing Symbols

When *printer escaping* is disabled, only the characters of the *symbol*'s *name* are output (but the case in which to print characters in the *name* is controlled by ***print-case***; see Section 22.1.3.3.2 (Effect of Readtable Case on the Lisp Printer)).

The remainder of Section 22.1.3.3 applies only when *printer escaping* is enabled.

When printing a *symbol*, the printer inserts enough *single escape* and/or *multiple escape* characters (*backslashes* and/or *vertical-bars*) so that if **read** were called with the same ***readtable*** and with ***read-base*** bound to the *current output base*, it would return the same *symbol* (if it is not *apparently uninterned*) or an *uninterned symbol* with the same *print name* (otherwise).

For example, if the *value* of ***print-base*** were `16` when printing the symbol `face`, it would have to be printed as `\FACE` or `\Face` or |`FACE`|, because the token `face` would be read as a hexadecimal number (decimal value 64206) if the *value* of ***read-base*** were `16`.

For additional restrictions concerning characters with nonstandard *syntax types* in the *current readtable*, see the *variable* ***print-readably***

For information about how the *Lisp reader* parses *symbols*, see Section 2.3.4 (Symbols as Tokens) and Section 2.4.8.5 (Sharpsign Colon).

nil might be printed as `()` when ***print-pretty*** is *true* and *printer escaping* is enabled.

22.1.3.3.1 Package Prefixes for Symbols

Package prefixes are printed if necessary. The rules for *package prefixes* are as follows. When the *symbol* is printed, if it is in the `KEYWORD` package, then it is printed with a preceding *colon*; otherwise, if it is *accessible* in the *current package*, it is printed without any *package prefix*; otherwise, it is printed with a *package prefix*.

A *symbol* that is *apparently uninterned* is printed preceded by "`#:`" if ***print-gensym*** is *true* and *printer escaping* is enabled; if ***print-gensym*** is *false* or *printer escaping* is disabled, then the *symbol* is printed without a prefix, as if it were in the *current package*.

Because the `#:` syntax does not intern the following symbol, it is necessary to use circular-list syntax if ***print-circle*** is *true* and the same uninterned symbol appears several times in an expression to be printed. For example, the result of

```
 (let ((x (make-symbol "FOO"))) (list x x))
```

would be printed as `(#:foo #:foo)` if ***print-circle*** were *false*, but as `(#1=#:foo #1#)` if ***print-circle*** were *true*.

A summary of the preceding package prefix rules follows:

`foo:bar`
: `foo:bar` is printed when *symbol* `bar` is external in its *home package* `foo` and is not *accessible* in the *current package*.

`foo::bar`
: `foo::bar` is printed when `bar` is internal in its *home package* `foo` and is not *accessible* in the *current package*.

`:bar`
: `:bar` is printed when the home package of `bar` is the `KEYWORD` package.

`#:bar`
: `#:bar` is printed when `bar` is *apparently uninterned*, even in the pathological case that `bar` has no *home package* but is nevertheless somehow *accessible* in the *current package*.

22.1.3.3.2 Effect of Readtable Case on the Lisp Printer

When *printer escaping* is disabled, or the characters under consideration are not already quoted specifically by *single escape* or *multiple escape* syntax, the *readtable case* of the *current readtable* affects the way the *Lisp printer* writes *symbols* in the following ways:

`:upcase`
: When the *readtable case* is `:upcase`, *uppercase characters* are printed in the case specified by ***print-case***, and *lowercase characters* are printed in their own case.

`:downcase`
: When the *readtable case* is `:downcase`, *uppercase characters* are printed in their own case, and *lowercase characters* are printed in the case specified by ***print-case***.

`:preserve`
: When the *readtable case* is `:preserve`, all *alphabetic characters* are printed in their own case.

`:invert`
: When the *readtable case* is `:invert`, the case of all *alphabetic characters* in single case symbol names is inverted. Mixed-case symbol names are printed as is.

The rules for escaping *alphabetic characters* in symbol names are affected by the **readtable-case** if *printer escaping* is enabled. *Alphabetic characters* are escaped as follows:

`:upcase`
: When the *readtable case* is `:upcase`, all *lowercase characters* must be escaped.

`:downcase`
: When the *readtable case* is `:downcase`, all *uppercase characters* must be escaped.

`:preserve`
: When the *readtable case* is `:preserve`, no *alphabetic characters* need be escaped.

`:invert`
: When the *readtable case* is `:invert`, no *alphabetic characters* need be escaped.

22.1.3.3.2.1 Examples of Effect of Readtable Case on the Lisp Printer

```
(defun test-readtable-case-printing ()
  (let ((*readtable* (copy-readtable nil))
        (*print-case* *print-case*))
    (format t "READTABLE-CASE *PRINT-CASE*  Symbol-name  Output~
             ~%--------------------------------------------------~
             ~%")
    (dolist (readtable-case '(:upcase :downcase :preserve :invert))
      (setf (readtable-case *readtable*) readtable-case)
      (dolist (print-case '(:upcase :downcase :capitalize))
        (dolist (symbol '(|ZEBRA| |Zebra| |zebra|))
          (setq *print-case* print-case)
          (format t "~&:~A~15T:~A~29T~A~42T~A"
                  (string-upcase readtable-case)
                  (string-upcase print-case)
                  (symbol-name symbol)
                  (prin1-to-string symbol)))))))
```

The output from `(test-readtable-case-printing)` should be as follows:

```
READTABLE-CASE *PRINT-CASE*  Symbol-name  Output
--------------------------------------------------
:UPCASE        :UPCASE       ZEBRA        ZEBRA
:UPCASE        :UPCASE       Zebra        |Zebra|
:UPCASE        :UPCASE       zebra        |zebra|
:UPCASE        :DOWNCASE     ZEBRA        zebra
:UPCASE        :DOWNCASE     Zebra        |Zebra|
```

```
:UPCASE         :DOWNCASE     zebra        |zebra|
:UPCASE         :CAPITALIZE   ZEBRA        Zebra
:UPCASE         :CAPITALIZE   Zebra        |Zebra|
:UPCASE         :CAPITALIZE   zebra        |zebra|
:DOWNCASE       :UPCASE       ZEBRA        |ZEBRA|
:DOWNCASE       :UPCASE       Zebra        |Zebra|
:DOWNCASE       :UPCASE       zebra        ZEBRA
:DOWNCASE       :DOWNCASE     ZEBRA        |ZEBRA|
:DOWNCASE       :DOWNCASE     Zebra        |Zebra|
:DOWNCASE       :DOWNCASE     zebra        zebra
:DOWNCASE       :CAPITALIZE   ZEBRA        |ZEBRA|
:DOWNCASE       :CAPITALIZE   Zebra        |Zebra|
:DOWNCASE       :CAPITALIZE   zebra        Zebra
:PRESERVE       :UPCASE       ZEBRA        ZEBRA
:PRESERVE       :UPCASE       Zebra        Zebra
:PRESERVE       :UPCASE       zebra        zebra
:PRESERVE       :DOWNCASE     ZEBRA        ZEBRA
:PRESERVE       :DOWNCASE     Zebra        Zebra
:PRESERVE       :DOWNCASE     zebra        zebra
:PRESERVE       :CAPITALIZE   ZEBRA        ZEBRA
:PRESERVE       :CAPITALIZE   Zebra        Zebra
:PRESERVE       :CAPITALIZE   zebra        zebra
:INVERT         :UPCASE       ZEBRA        zebra
:INVERT         :UPCASE       Zebra        Zebra
:INVERT         :UPCASE       zebra        ZEBRA
:INVERT         :DOWNCASE     ZEBRA        zebra
:INVERT         :DOWNCASE     Zebra        Zebra
:INVERT         :DOWNCASE     zebra        ZEBRA
:INVERT         :CAPITALIZE   ZEBRA        zebra
:INVERT         :CAPITALIZE   Zebra        Zebra
:INVERT         :CAPITALIZE   zebra        ZEBRA
```

22.1.3.4 Printing Strings

The characters of the *string* are output in order. If *printer escaping* is enabled, a *double-quote* is output before and after, and all *double-quotes* and *single escapes* are preceded by *backslash*. The printing of *strings* is not affected by ***print-array***. Only the *active elements* of the *string* are printed.

For information on how the *Lisp reader* parses *strings*, see Section 2.4.5 (Double-Quote).

22.1.3.5 Printing Lists and Conses

Wherever possible, list notation is preferred over dot notation. Therefore the following algorithm is used to print a *cons* x:

1. A *left-parenthesis* is printed.
2. The *car* of x is printed.
3. If the *cdr* of x is itself a *cons*, it is made to be the current *cons* (i.e., x becomes that *cons*), a *space* is printed, and step 2 is re-entered.
4. If the *cdr* of x is not *null*, a *space*, a *dot*, a *space*, and the *cdr* of x are printed.
5. A *right-parenthesis* is printed.

Actually, the above algorithm is only used when ***print-pretty*** is *false*. When ***print-pretty*** is *true* (or when **pprint** is used), additional *whitespace*[1] may replace the use of a single *space*, and a more elaborate algorithm with similar goals but more presentational flexibility is used; see Section 22.1.2 (Printer Dispatching).

Although the two expressions below are equivalent, and the reader accepts either one and produces the same *cons*, the printer always prints such a *cons* in the second form.

```
(a . (b . ((c . (d . nil)) . (e . nil))))
(a b (c d) e)
```

The printing of *conses* is affected by ***print-level***, ***print-length***, and ***print-circle***.

Following are examples of printed representations of *lists*:

```
(a . b)     ;A dotted pair of a and b
(a.b)       ;A list of one element, the symbol named a.b
(a. b)      ;A list of two elements a. and b
(a .b)      ;A list of two elements a and .b
(a b . c)   ;A dotted list of a and b with c at the end; two conses
.iot        ;The symbol whose name is .iot
(. b)       ;Invalid -- an error is signaled if an attempt is made to read
            ;this syntax.
(a .)       ;Invalid -- an error is signaled.
(a .. b)    ;Invalid -- an error is signaled.
(a . . b)   ;Invalid -- an error is signaled.
(a b c ...) ;Invalid -- an error is signaled.
(a \. b)    ;A list of three elements a, ., and b
(a |.| b)   ;A list of three elements a, ., and b
(a \... b)  ;A list of three elements a, ..., and b
(a |...| b) ;A list of three elements a, ..., and b
```

For information on how the *Lisp reader* parses *lists* and *conses*, see Section 2.4.1 (Left-Parenthesis).

22.1.3.6 Printing Bit Vectors

A *bit vector* is printed as `#*` followed by the bits of the *bit vector* in order. If ***print-array*** is *false*, then the *bit vector* is printed in a format (using #<) that is concise but not readable. Only the *active elements* of the *bit vector* are printed.

For information on *Lisp reader* parsing of *bit vectors*, see Section 2.4.8.4 (Sharpsign Asterisk).

22.1.3.7 Printing Other Vectors

If ***print-array*** is *true* and ***print-readably*** is *false*, any *vector* other than a *string* or *bit vector* is printed using general-vector syntax; this means that information about specialized vector representations does not appear. The printed representation of a zero-length *vector* is `#()`. The printed representation of a non-zero-length *vector* begins with `#(`. Following that, the first element of the *vector* is printed. If there are any other elements, they are printed in turn, with each such additional element preceded by a *space* if ***print-pretty*** is *false*, or *whitespace*[1] if ***print-pretty*** is *true*. A *right-parenthesis* after the last element terminates the printed representation of the *vector*. The printing of *vectors* is affected by ***print-level*** and ***print-length***. If the *vector* has a *fill pointer*, then only those elements below the *fill pointer* are printed.

If both ***print-array*** and ***print-readably*** are *false*, the *vector* is not printed as described above, but in a format (using #<) that is concise but not readable.

If ***print-readably*** is *true*, the *vector* prints in an *implementation-defined* manner; see the *variable* ***print-readably***.

For information on how the *Lisp reader* parses these "other *vectors*," see Section 2.4.8.3 (Sharpsign Left-Parenthesis).

22.1.3.8 Printing Other Arrays

If ***print-array*** is *true* and ***print-readably*** is *false*, any *array* other than a *vector* is printed using #nA format. Let n be the *rank* of the *array*. Then # is printed, then n as a decimal integer, then A, then n open parentheses. Next the *elements* are scanned in row-major order, using **write** on each *element*, and separating *elements* from each other with *whitespace*[1]. The array's dimensions are numbered 0 to n-1 from left to right, and are enumerated with the rightmost index changing fastest. Every time the index for dimension j is incremented, the following actions are taken:

* If j < n-1, then a close parenthesis is printed.
* If incrementing the index for dimension j caused it to equal dimension j, that index is reset to zero and the index for dimension j-1 is incremented (thereby performing these three steps recursively), unless j=0, in which case the entire algorithm is terminated. If incrementing the index for dimension j did not cause it to equal dimension j, then a space is printed.
* If j < n-1, then an open parenthesis is printed.

This causes the contents to be printed in a format suitable for `:initial-contents` to **make-array**. The lists effectively printed by this procedure are subject to truncation by ***print-level*** and ***print-length***.

If the *array* is of a specialized *type*, containing bits or characters, then the innermost lists generated by the algorithm given above can instead be printed using bit-vector or string syntax, provided that these innermost lists would not be subject to truncation by ***print-length***.

If both ***print-array*** and ***print-readably*** are *false*, then the *array* is printed in a format (using #<) that is concise but not readable.

If ***print-readably*** is *true*, the *array* prints in an *implementation-defined* manner; see the *variable* ***print-readably***. In particular, this may be important for arrays having some dimension 0.

For information on how the *Lisp reader* parses these "other *arrays*," see Section 2.4.8.12 (Sharpsign A).

22.1.3.9 Examples of Printing Arrays

```
 (let ((a (make-array ’(3 3)))
       (*print-pretty* t)
       (*print-array* t))
   (dotimes (i 3) (dotimes (j 3) (setf (aref a i j) (format nil "<~D,~D>" i j))))
   (print a)
   (print (make-array 9 :displaced-to a)))
>>  #2A(("<0,0>" "<0,1>" "<0,2>")
>>      ("<1,0>" "<1,1>" "<1,2>")
>>      ("<2,0>" "<2,1>" "<2,2>"))
>>  #("<0,0>" "<0,1>" "<0,2>" "<1,0>" "<1,1>" "<1,2>" "<2,0>" "<2,1>" "<2,2>")
=>  #<ARRAY 9 indirect 36363476>
```

22.1.3.10 Printing Random States

A specific syntax for printing *objects* of *type* **random-state** is not specified. However, every *implementation* must arrange to print a *random state object* in such a way that, within the same implementation, **read** can construct from the printed representation a copy of the *random state* object as if the copy had been made by **make-random-state**.

If the type *random state* is effectively implemented by using the machinery for **defstruct**, the usual structure syntax can then be used for printing *random state* objects; one might look something like

```
#S(RANDOM-STATE :DATA #(14 49 98436589 786345 8734658324 ... ))
```

where the components are *implementation-dependent*.

22.1.3.11 Printing Pathnames

When *printer escaping* is enabled, the syntax `#P"..."` is how a *pathname* is printed by **write** and the other functions herein described. The `"..."` is the namestring representation of the pathname.

When *printer escaping* is disabled, **write** writes a *pathname P* by writing `(namestring` *P*`)` instead.

For information on how the *Lisp reader* parses *pathnames*, see Section 2.4.8.14 (Sharpsign P).

22.1.3.12 Printing Structures

By default, a *structure* of type S is printed using `#S` syntax. This behavior can be customized by specifying a `:print-function` or `:print-object` option to the **defstruct** *form* that defines S, or by writing a **print-object** *method* that is *specialized* for *objects* of type S.

Different structures might print out in different ways; the default notation for structures is:

```
#S(structure-name {slot-key slot-value}*)
```

where `#S` indicates structure syntax, *structure-name* is a *structure name*, each *slot-key* is an initialization argument *name* for a *slot* in the *structure*, and each corresponding *slot-value* is a representation of the *object* in that *slot*.

For information on how the *Lisp reader* parses *structures*, see Section 2.4.8.13 (Sharpsign S).

22.1.3.13 Printing Other Objects

Other *objects* are printed in an *implementation-dependent* manner. It is not required that an *implementation* print those *objects readably*.

For example, *hash tables*, *readtables*, *packages*, *streams*, and *functions* might not print *readably*.

A common notation to use in this circumstance is `#<...>`. Since `#<` is not readable by the *Lisp reader*, the precise format of the text which follows is not important, but a common format to use is that provided by the **print-unreadable-object** *macro*.

For information on how the *Lisp reader* treats this notation, see Section 2.4.8.20 (Sharpsign Less-Than-Sign). For information on how to notate *objects* that cannot be printed *readably*, see Section 2.4.8.6 (Sharpsign Dot).

22.1.4 Examples of Printer Behavior

```
 (let ((*print-escape* t)) (fresh-line) (write #\a))
>>  #\a
=>  #\a
 (let ((*print-escape* nil) (*print-readably* nil))
   (fresh-line)
   (write #\a))
>>  a
=>  #\a
 (progn (fresh-line) (prin1 #\a))
>>  #\a
=>  #\a
 (progn (fresh-line) (print #\a))
>>
```

```
>>  #\a
=>  #\a
 (progn (fresh-line) (princ #\a))
>>  a
=>  #\a

 (dolist (val '(t nil))
   (let ((*print-escape* val) (*print-readably* val))
     (print '#\a)
     (prin1 #\a) (write-char #\Space)
     (princ #\a) (write-char #\Space)
     (write #\a)))
>>  #\a #\a a #\a
>>  #\a #\a a a
=>  NIL

 (progn (fresh-line) (write '(let ((a 1) (b 2)) (+ a b))))
>>  (LET ((A 1) (B 2)) (+ A B))
=>  (LET ((A 1) (B 2)) (+ A B))

 (progn (fresh-line) (pprint '(let ((a 1) (b 2)) (+ a b))))
>>  (LET ((A 1)
>>        (B 2))
>>    (+ A B))
=>  (LET ((A 1) (B 2)) (+ A B))

 (progn (fresh-line)
        (write '(let ((a 1) (b 2)) (+ a b)) :pretty t))
>>  (LET ((A 1)
>>        (B 2))
>>    (+ A B))
=>  (LET ((A 1) (B 2)) (+ A B))

 (with-output-to-string (s)
    (write 'write :stream s)
    (prin1 'prin1 s))
=>  "WRITEPRIN1"
```

22.2 The Lisp Pretty Printer

22.2.1 Pretty Printer Concepts

The facilities provided by the *pretty printer* permit *programs* to redefine the way in which *code* is displayed, and allow the full power of *pretty printing* to be applied to complex combinations of data structures.

Whether any given style of output is in fact "pretty" is inherently a somewhat subjective issue. However, since the effect of the *pretty printer* can be customized by *conforming programs*, the necessary flexibility is provided for individual *programs* to achieve an arbitrary degree of aesthetic control.

By providing direct access to the mechanisms within the pretty printer that make dynamic decisions about layout, the macros and functions **pprint-logical-block**, **pprint-newline**, and **pprint-indent** make it possible to specify pretty printing layout rules as a part of any function that produces output. They also make it very easy for the detection of circularity and sharing, and abbreviation based on length and nesting depth to be supported by the function.

The *pretty printer* is driven entirely by dispatch based on the *value* of ***print-pprint-dispatch***. The *function* **set-pprint-dispatch** makes it possible for *conforming programs* to associate new pretty printing functions with a *type*.

22.2.1.1 Dynamic Control of the Arrangement of Output

The actions of the *pretty printer* when a piece of output is too large to fit in the space available can be precisely controlled. Three concepts underlie the way these operations work---*logical blocks*, *conditional newlines*, and *sections*. Before proceeding further, it is important to define these terms.

The first line of the next figure shows a schematic piece of output. Each of the characters in the output is represented by `"-"`. The positions of conditional newlines are indicated by digits. The beginnings and ends of logical blocks are indicated by "`<`" and "`>`" respectively.

The output as a whole is a logical block and the outermost section. This section is indicated by the `0`'s on the second line of Figure 1. Logical blocks nested within the output are specified by the macro **pprint-logical-block**. Conditional newline positions are specified by calls to **pprint-newline**. Each conditional newline defines two sections (one before it and one after it) and is associated with a third (the section immediately containing it).

The section after a conditional newline consists of: all the output up to, but not including, (a) the next conditional newline immediately contained in the same logical block; or if (a) is not applicable, (b) the next newline that is at a lesser level of nesting in logical blocks; or if (b) is not applicable, (c) the end of the output.

The section before a conditional newline consists of: all the output back to, but not including, (a) the previous conditional newline that is immediately contained in the same logical block; or if (a) is not applicable, (b) the beginning of the immediately containing logical block. The last four lines in Figure 1 indicate the sections before and after the four conditional newlines.

The section immediately containing a conditional newline is the shortest section that contains the conditional newline in question. In the next figure, the first conditional newline is immediately contained in the section marked with `0`'s, the second and third conditional newlines are immediately contained in the section before the fourth conditional newline, and the fourth conditional newline is immediately contained in the section after the first conditional newline.

```
 <-1---<--<--2---3->--4-->->
 000000000000000000000000000
 11 111111111111111111111111
           22 222
              333 3333
        44444444444444 44444
```

Figure 22-3. Example of Logical Blocks, Conditional Newlines, and Sections

Whenever possible, the pretty printer displays the entire contents of a section on a single line. However, if the section is too long to fit in the space available, line breaks are inserted at conditional newline positions within the section.

22.2.1.2 Format Directive Interface

The primary interface to operations for dynamically determining the arrangement of output is provided through the functions and macros of the pretty printer. The next figure shows the defined names related to *pretty printing*.

```
*print-lines*            pprint-dispatch                pprint-pop
*print-miser-width*      pprint-exit-if-list-exhausted  pprint-tab
*print-pprint-dispatch*  pprint-fill                    pprint-tabular
*print-right-margin*     pprint-indent                  set-pprint-dispatch
copy-pprint-dispatch     pprint-linear                  write
format                   pprint-logical-block
formatter                pprint-newline
```

Figure 22-4. Defined names related to pretty printing.

The next figure identifies a set of *format directives* which serve as an alternate interface to the same pretty printing operations in a more textually compact form.

```
~I   ~W       ~<...~:>
~:T  ~/.../  ~_
```

Figure 22-5. Format directives related to Pretty Printing

22.2.1.3 Compiling Format Strings

A *format string* is essentially a program in a special-purpose language that performs printing, and that is interpreted by the *function* **format**. The **formatter** *macro* provides the efficiency of using a *compiled function* to do that same printing but without losing the textual compactness of *format strings*.

A *format control* is either a *format string* or a *function* that was returned by the the **formatter** *macro*.

22.2.1.4 Pretty Print Dispatch Tables

A *pprint dispatch table* is a mapping from keys to pairs of values. Each key is a *type specifier*. The values associated with a key are a "function" (specifically, a *function designator* or **nil**) and a "numerical priority" (specifically, a *real*). Basic insertion and retrieval is done based on the keys with the equality of keys being tested by **equal**.

When ***print-pretty*** is *true*, the *current pprint dispatch table* (in ***print-pprint-dispatch***) controls how *objects* are printed. The information in this table takes precedence over all other mechanisms for specifying how to print *objects*. In particular, it has priority over user-defined **print-object** *methods* because the *current pprint dispatch table* is consulted first.

The function is chosen from the *current pprint dispatch table* by finding the highest priority function that is associated with a *type specifier* that matches the *object*; if there is more than one such function, it is *implementation-dependent* which is used.

However, if there is no information in the table about how to *pretty print* a particular kind of *object*, a *function* is invoked which uses **print-object** to print the *object*. The value of ***print-pretty*** is still *true* when this function is *called*, and individual methods for **print-object** might still elect to produce output in a special format conditional on the *value* of ***print-pretty***.

22.2.1.5 Pretty Printer Margins

A primary goal of pretty printing is to keep the output between a pair of margins. The column where the output begins is taken as the left margin. If the current column cannot be determined at the time output begins, the left margin is assumed to be zero. The right margin is controlled by ***print-right-margin***.

22.2.2 Examples of using the Pretty Printer

As an example of the interaction of logical blocks, conditional newlines, and indentation, consider the function `simple-pprint-defun` below. This function prints out lists whose *cars* are **defun** in the standard way assuming that the list has exactly length `4`.

```
(defun simple-pprint-defun (*standard-output* list)
  (pprint-logical-block (*standard-output* list :prefix "(" :suffix ")")
    (write (first list))
    (write-char #\Space)
    (pprint-newline :miser)
    (pprint-indent :current 0)
    (write (second list))
    (write-char #\Space)
    (pprint-newline :fill)
    (write (third list))
    (pprint-indent :block 1)
    (write-char #\Space)
    (pprint-newline :linear)
    (write (fourth list))))
```

Suppose that one evaluates the following:

```
(simple-pprint-defun *standard-output* '(defun prod (x y) (* x y)))
```

If the line width available is greater than or equal to `26`, then all of the output appears on one line. If the line width available is reduced to `25`, a line break is inserted at the linear-style conditional newline before the *expression* `(* x y)`, producing the output shown. The `(pprint-indent :block 1)` causes `(* x y)` to be printed at a relative indentation of `1` in the logical block.

```
 (DEFUN PROD (X Y)
   (* X Y))
```

If the line width available is `15`, a line break is also inserted at the fill style conditional newline before the argument list. The call on `(pprint-indent :current 0)` causes the argument list to line up under the function name.

```
(DEFUN PROD
       (X Y)
  (* X Y))
```

If ***print-miser-width*** were greater than or equal to 14, the example output above would have been as follows, because all indentation changes are ignored in miser mode and line breaks are inserted at miser-style conditional newlines.

```
 (DEFUN
  PROD
  (X Y)
  (* X Y))
```

As an example of a per-line prefix, consider that evaluating the following produces the output shown with a line width of `20` and ***print-miser-width*** of **nil**.

```
 (pprint-logical-block (*standard-output* nil :per-line-prefix ";;; ")
   (simple-pprint-defun *standard-output* '(defun prod (x y) (* x y))))

 ;;; (DEFUN PROD
 ;;;        (X Y)
 ;;;   (* X Y))
```

As a more complex (and realistic) example, consider the function `pprint-let` below. This specifies how to print a **let** *form* in the traditional style. It is more complex than the example above, because it has to deal with nested structure. Also, unlike the example above it contains complete code to readably print any possible list that begins with the *symbol* **let**. The outermost **pprint-logical-block** *form* handles the printing of the input list as a whole and specifies that parentheses should be printed in the output. The second **pprint-logical-block** *form* handles the list of binding pairs. Each pair in the list is itself printed by the innermost **pprint-logical-block**. (A **loop** *form* is used instead of merely decomposing the pair into two *objects* so that readable output will be produced no matter whether the list corresponding to the pair has one element, two elements, or (being malformed) has more than two elements.) A space and a fill-style conditional newline are placed after each pair except the last. The loop at the end of the topmost **pprint-logical-block** *form* prints out the forms in the body of the **let** *form* separated by spaces and linear-style conditional newlines.

```
(defun pprint-let (*standard-output* list)
  (pprint-logical-block (nil list :prefix "(" :suffix ")")
    (write (pprint-pop))
    (pprint-exit-if-list-exhausted)
    (write-char #\Space)
    (pprint-logical-block (nil (pprint-pop) :prefix "(" :suffix ")")
      (pprint-exit-if-list-exhausted)
      (loop (pprint-logical-block (nil (pprint-pop) :prefix "(" :suffix ")")
              (pprint-exit-if-list-exhausted)
              (loop (write (pprint-pop))
                    (pprint-exit-if-list-exhausted)
                    (write-char #\Space)
                    (pprint-newline :linear)))
            (pprint-exit-if-list-exhausted)
            (write-char #\Space)
            (pprint-newline :fill)))
    (pprint-indent :block 1)
    (loop (pprint-exit-if-list-exhausted)
          (write-char #\Space)
          (pprint-newline :linear)
          (write (pprint-pop)))))
```

Suppose that one evaluates the following with ***print-level*** being 4, and ***print-circle*** being *true*.

```
(pprint-let *standard-output*
            '#1=(let (x (*print-length* (f (g 3)))
                        (z . 2) (k (car y)))
                  (setq x (sqrt z)) #1#))
```

If the line length is greater than or equal to `77`, the output produced appears on one line. However, if the line length is `76`, line breaks are inserted at the linear-style conditional newlines separating the forms in the body and the output below is produced. Note that, the degenerate binding pair `x` is printed readably even though it fails to be a list; a depth abbreviation marker is printed in place of `(g 3)`; the binding pair `(z . 2)` is printed readably even though it is not a proper list; and appropriate circularity markers are printed.

```
#1=(LET (X (*PRINT-LENGTH* (F #)) (Z . 2) (K (CAR Y)))
     (SETQ X (SQRT Z))
     #1#)
```

If the line length is reduced to `35`, a line break is inserted at one of the fill-style conditional newlines separating the binding pairs.

```
#1=(LET (X (*PRINT-PRETTY* (F #))
         (Z . 2) (K (CAR Y)))
     (SETQ X (SQRT Z))
     #1#)
```

Suppose that the line length is further reduced to `22` and ***print-length*** is set to `3`. In this situation, line breaks are inserted after both the first and second binding pairs. In addition, the second binding pair is itself broken across two lines. Clause (b) of the description of fill-style conditional newlines (see the *function* **pprint-newline**) prevents the binding pair `(z . 2)` from being printed at the end of the third line. Note that the length abbreviation hides the circularity from view and therefore the printing of circularity markers disappears.

```
(LET (X
      (*PRINT-LENGTH*
       (F #))
      (Z . 2) ...)
  (SETQ X (SQRT Z))
  ...)
```

The next function prints a vector using "`#(...)`" notation.

```
(defun pprint-vector (*standard-output* v)
  (pprint-logical-block (nil nil :prefix "#(" :suffix ")")
    (let ((end (length v)) (i 0))
      (when (plusp end)
        (loop (pprint-pop)
              (write (aref v i))
              (if (= (incf i) end) (return nil))
              (write-char #\Space)
              (pprint-newline :fill))))))
```

Evaluating the following with a line length of 15 produces the output shown.

```
(pprint-vector *standard-output* ’#(12 34 567 8 9012 34 567 89 0 1 23))

#(12 34 567 8
  9012 34 567
  89 0 1 23)
```

As examples of the convenience of specifying pretty printing with *format strings*, consider that the functions `simple-pprint-defun` and `pprint-let` used as examples above can be compactly defined as follows. (The function `pprint-vector` cannot be defined using **format** because the data structure it traverses is not a list.)

```
(defun simple-pprint-defun (*standard-output* list)
  (format T "~:<~W ~@_~:I~W ~:_~W~1I ~_~W~:>" list))

(defun pprint-let (*standard-output* list)
  (format T "~:<~W~^~:<~@{~:<~@{~W~^~_~}~:>~^~:_~}~:>~1I~@{~^~_~W~}~:>" list))
```

In the following example, the first *form* restores ***print-pprint-dispatch*** to the equivalent of its initial value. The next two forms then set up a special way to pretty print ratios. Note that the more specific *type specifier* has to be associated with a higher priority.

```
(setq *print-pprint-dispatch* (copy-pprint-dispatch nil))

(set-pprint-dispatch ’ratio
  #’(lambda (s obj)
      (format s "#.(/ ~W ~W)"
                 (numerator obj) (denominator obj))))

(set-pprint-dispatch ’(and ratio (satisfies minusp))
  #’(lambda (s obj)
      (format s "#.(- (/ ~W ~W))"
               (- (numerator obj)) (denominator obj)))
  5)

(pprint ’(1/3 -2/3))
(#.(/ 1 3) #.(- (/ 2 3)))
```

The following two *forms* illustrate the definition of pretty printing functions for types of *code*. The first *form* illustrates how to specify the traditional method for printing quoted objects using *single-quote*. Note the care taken to ensure that data lists that happen to begin with **quote** will be printed readably. The second form specifies that lists beginning with the symbol `my-let` should print the same way that lists beginning with **let** print when the initial *pprint dispatch table* is in effect.

```
(set-pprint-dispatch '(cons (member quote)) ()
  #'(lambda (s list)
      (if (and (consp (cdr list)) (null (cddr list)))
         (funcall (formatter "'~W") s (cadr list))
         (pprint-fill s list))))

(set-pprint-dispatch '(cons (member my-let))
                     (pprint-dispatch '(let) nil))
```

The next example specifies a default method for printing lists that do not correspond to function calls. Note that the functions **pprint-linear**, **pprint-fill**, and **pprint-tabular** are all defined with optional *colon-p* and *at-sign-p* arguments so that they can be used as **pprint dispatch functions** as well as `~/.../` functions.

```
(set-pprint-dispatch '(cons (not (and symbol (satisfies fboundp))))
                     #'pprint-fill -5)

;; Assume a line length of 9
(pprint '(0 b c d e f g h i j k))
(0 b c d
 e f g h
 i j k)
```

This final example shows how to define a pretty printing function for a user defined data structure.

```
(defstruct family mom kids)

(set-pprint-dispatch 'family
  #'(lambda (s f)
      (funcall (formatter "~@<#<~;~W and ~2I~_~/pprint-fill/~;>~:>")
              s (family-mom f) (family-kids f))))
```

The pretty printing function for the structure `family` specifies how to adjust the layout of the output so that it can fit aesthetically into a variety of line widths. In addition, it obeys the printer control variables ***print-level***, ***print-length***, ***print-lines***, ***print-circle*** and ***print-escape***, and can tolerate several different kinds of malformity in the data structure. The output below shows what is printed out with a right margin of `25`, ***print-pretty*** being *true*, ***print-escape*** being *false*, and a malformed `kids` list.

```
(write (list 'principal-family
             (make-family :mom "Lucy"
                          :kids '("Mark" "Bob" . "Dan")))
       :right-margin 25 :pretty T :escape nil :miser-width nil)
(PRINCIPAL-FAMILY
 #<Lucy and
     Mark Bob . Dan>)
```

Note that a pretty printing function for a structure is different from the structure's **print-object** *method*. While **print-object** *methods* are permanently associated with a structure, pretty printing functions are stored in *pprint dispatch tables* and can be rapidly changed to reflect different printing needs. If there is no pretty printing function for a structure in the current *pprint dispatch table*, its **print-object** *method* is used instead.

22.2.3 Notes about the Pretty Printer's Background

For a background reference to the abstract concepts detailed in this section, see *XP: A Common Lisp Pretty Printing System*. The details of that paper are not binding on this document, but may be helpful in establishing a conceptual basis for understanding this material.

22.3 Formatted Output

format is useful for producing nicely formatted text, producing good-looking messages, and so on. **format** can generate and return a *string* or output to *destination*.

The *control-string* argument to **format** is actually a *format control*. That is, it can be either a *format string* or a *function*, for example a *function* returned by the **formatter** *macro*.

If it is a *function*, the *function* is called with the appropriate output stream as its first argument and the data arguments to **format** as its remaining arguments. The function should perform whatever output is necessary and return the unused tail of the arguments (if any).

The compilation process performed by **formatter** produces a *function* that would do with its *arguments* as the **format** interpreter would do with those *arguments*.

The remainder of this section describes what happens if the *control-string* is a *format string*.

Control-string is composed of simple text (*characters*) and embedded directives.

format writes the simple text as is; each embedded directive specifies further text output that is to appear at the corresponding point within the simple text. Most directives use one or more elements of *args* to create their output.

A directive consists of a *tilde*, optional prefix parameters separated by commas, optional *colon* and *at-sign* modifiers, and a single character indicating what kind of directive this is. There is no required ordering between the *at-sign* and *colon* modifier. The *case* of the directive character is ignored. Prefix parameters are notated as signed (sign is optional) decimal numbers, or as a *single-quote* followed by a character. For example, `~5,'0d` can be used to print an *integer* in decimal radix in five columns with leading zeros, or `~5,'*d` to get leading asterisks.

In place of a prefix parameter to a directive, `V` (or `v`) can be used. In this case, **format** takes an argument from *args* as a parameter to the directive. The argument should be an *integer* or *character*. If the *arg* used by a `V` parameter is **nil**, the effect is as if the parameter had been omitted. # can be used in place of a prefix parameter; it represents the number of *args* remaining to be processed. When used within a recursive format, in the context of `~?` or `~{`, the # prefix parameter represents the number of *format arguments* remaining within the recursive call.

Examples of *format strings*:

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

Figure 22-6. Examples of format control strings

format sends the output to *destination*. If *destination* is **nil**, **format** creates and returns a *string* containing the output from *control-string*. If *destination* is *non-nil*, it must be a *string* with a *fill pointer*, a *stream*, or the symbol **t**. If *destination* is a *string* with a *fill pointer*, the output is added to the end of the *string*. If *destination* is a *stream*, the output is sent to that *stream*. If *destination* is **t**, the output is sent to *standard output*.

In the description of the directives that follows, the term *arg* in general refers to the next item of the set of *args* to be processed. The word or phrase at the beginning of each description is a mnemonic for the directive. **format** directives do not bind any of the printer control variables (***print-...***) except as specified in the following descriptions. Implementations may specify the binding of new, implementation-specific printer control variables for each **format** directive, but they may neither bind any standard printer control variables not specified in description of a **format** directive nor fail to bind any standard printer control variables as specified in the description.

22.3.1 FORMAT Basic Output

22.3.1.1 Tilde C: Character

The next *arg* should be a *character*; it is printed according to the modifier flags.

`~C` prints the *character* as if by using **write-char** if it is a *simple character*. *Characters* that are not *simple* are not necessarily printed as if by **write-char**, but are displayed in an *implementation-defined*, abbreviated format. For example,

```
 (format nil "~C" #\A) =>  "A"
 (format nil "~C" #\Space) =>  " "
```

`~:C` is the same as `~C` for *printing characters*, but other *characters* are "spelled out." The intent is that this is a "pretty" format for printing characters. For *simple characters* that are not *printing*, what is spelled out is the *name* of the *character* (see **char-name**). For *characters* that are not *simple* and not *printing*, what is spelled out is *implementation-defined*. For example,

```
 (format nil "~:C" #\A) =>  "A"
 (format nil "~:C" #\Space) =>  "Space"
;; This next example assumes an implementation-defined "Control" attribute.
 (format nil "~:C" #\Control-Space)
=>  "Control-Space"
OR=>  "c-Space"
```

`~:@C` prints what `~:C` would, and then if the *character* requires unusual shift keys on the keyboard to type it, this fact is mentioned. For example,

```
 (format nil "~:@C" #\Control-Partial) =>  "Control-<PARTIAL> (Top-F)"
```

This is the format used for telling the user about a key he is expected to type, in prompts, for instance. The precise output may depend not only on the implementation, but on the particular I/O devices in use.

`~@C` prints the *character* in a way that the *Lisp reader* can understand, using `#\` syntax.

`~@C` binds ***print-escape*** to **t**.

22.3.1.2 Tilde Percent: Newline

This outputs a `#\Newline` character, thereby terminating the current output line and beginning a new one. `~`*n*`%` outputs *n* newlines. No *arg* is used.

22.3.1.3 Tilde Ampersand: Fresh-Line

Unless it can be determined that the output stream is already at the beginning of a line, this outputs a newline. `~`*n*`&` calls **fresh-line** and then outputs *n*-1 newlines. `~0&` does nothing.

22.3.1.4 Tilde Vertical-Bar: Page

This outputs a page separator character, if possible. ~*n*| does this *n* times.

22.3.1.5 Tilde Tilde: Tilde

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

22.3.2 FORMAT Radix Control

22.3.2.1 Tilde R: Radix

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

If no prefix parameters are given to ~R, then a different interpretation is given. The argument should be an *integer*. For example, if *arg* is 4:

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

For example:

```
(format nil "~,,' ,4:B" 13) =>  "1101"
(format nil "~,,' ,4:B" 17) =>  "1 0001"
(format nil "~19,0,' ,4:B" 3333) =>  "0000 1101 0000 0101"
(format nil "~3,,,' ,2:R" 17) =>  "1 22"
(format nil "~,,'|,2:D" #xFFFF) =>   "6|55|35"
```

If and only if the first parameter, *n*, is supplied, ~R binds ***print-escape*** to *false*, ***print-radix*** to *false*, ***print-base*** to *n*, and ***print-readably*** to *false*.

If and only if no parameters are supplied, ~R binds ***print-base*** to 10.

22.3.2.2 Tilde D: Decimal

An *arg*, which should be an *integer*, is printed in decimal radix. ~D will never put a decimal point after the number.

~*mincol*D uses a column width of *mincol*; spaces are inserted on the left if the number requires fewer than *mincol* columns for its digits and sign. If the number doesn't fit in *mincol* columns, additional columns are used as needed.

~*mincol*,*padchar*D uses *padchar* as the pad character instead of space.

If *arg* is not an *integer*, it is printed in ~A format and decimal base.

The @ modifier causes the number's sign to be printed always; the default is to print it only if the number is negative. The : modifier causes commas to be printed between groups of digits; *commachar* may be used to change the character used as the comma. *comma-interval* must be an *integer* and defaults to 3. When the : modifier is given to any of these directives, the *commachar* is printed between groups of *comma-interval* digits.

Thus the most general form of ~D is ~*mincol*,*padchar*,*commachar*,*comma-interval*D.

~D binds ***print-escape*** to *false*, ***print-radix*** to *false*, ***print-base*** to 10, and ***print-readably*** to *false*.

22.3.2.3 Tilde B: Binary

This is just like `~D` but prints in binary radix (radix 2) instead of decimal. The full form is therefore ~*mincol*`,`*padchar*`,`*commachar*`,`*comma-interval*`B`.

`~B` binds ***print-escape*** to *false*, ***print-radix*** to *false*, ***print-base*** to `2`, and ***print-readably*** to *false*.

22.3.2.4 Tilde O: Octal

This is just like `~D` but prints in octal radix (radix 8) instead of decimal. The full form is therefore ~*mincol*`,`*padchar*`,`*commachar*`,`*comma-interval*`O`.

`~O` binds ***print-escape*** to *false*, ***print-radix*** to *false*, ***print-base*** to `8`, and ***print-readably*** to *false*.

22.3.2.5 Tilde X: Hexadecimal

This is just like `~D` but prints in hexadecimal radix (radix 16) instead of decimal. The full form is therefore ~*mincol*`,`*padchar*`,`*commachar*`,`*comma-interval*`X`.

`~X` binds ***print-escape*** to *false*, ***print-radix*** to *false*, ***print-base*** to `16`, and ***print-readably*** to *false*.

22.3.3 FORMAT Floating-Point Printers

22.3.3.1 Tilde F: Fixed-Format Floating-Point

The next *arg* is printed as a *float*.

The full form is ~*w*`,`*d*`,`*k*`,`*overflowchar*`,`*padchar*`F`. The parameter *w* is the width of the field to be printed; *d* is the number of digits to print after the decimal point; *k* is a scale factor that defaults to zero.

Exactly *w* characters will be output. First, leading copies of the character *padchar* (which defaults to a space) are printed, if necessary, to pad the field on the left. If the *arg* is negative, then a minus sign is printed; if the *arg* is not negative, then a plus sign is printed if and only if the `@` modifier was supplied. Then a sequence of digits, containing a single embedded decimal point, is printed; this represents the magnitude of the value of *arg* times 10^*k*, rounded to *d* fractional digits. When rounding up and rounding down would produce printed values equidistant from the scaled value of *arg*, then the implementation is free to use either one. For example, printing the argument `6.375` using the format `~4,2F` may correctly produce either `6.37` or `6.38`. Leading zeros are not permitted, except that a single zero digit is output before the decimal point if the printed value is less than one, and this single zero digit is not output at all if *w*=*d*+1.

If it is impossible to print the value in the required format in a field of width *w*, then one of two actions is taken. If the parameter *overflowchar* is supplied, then *w* copies of that parameter are printed instead of the scaled value of *arg*. If the *overflowchar* parameter is omitted, then the scaled value is printed using more than *w* characters, as many more as may be needed.

If the *w* parameter is omitted, then the field is of variable width. In effect, a value is chosen for *w* in such a way that no leading pad characters need to be printed and exactly *d* characters will follow the decimal point. For example, the directive `~,2F` will print exactly two digits after the decimal point and as many as necessary before the decimal point.

If the parameter *d* is omitted, then there is no constraint on the number of digits to appear after the decimal point. A value is chosen for *d* in such a way that as many digits as possible may be printed subject to the width constraint imposed by the parameter *w* and the constraint that no trailing zero digits may appear in the fraction, except that if the fraction to be printed is zero, then a single zero digit should appear after the decimal point if permitted by the width constraint.

If both *w* and *d* are omitted, then the effect is to print the value using ordinary free-format output; **prin1** uses this format for any number whose magnitude is either zero or between 10^-3 (inclusive) and 10^7 (exclusive).

If *w* is omitted, then if the magnitude of *arg* is so large (or, if *d* is also omitted, so small) that more than 100 digits would have to be printed, then an implementation is free, at its discretion, to print the number using exponential notation instead, as if by the directive `~E` (with all parameters to `~E` defaulted, not taking their values from the `~F` directive).

If *arg* is a *rational* number, then it is coerced to be a *single float* and then printed. Alternatively, an implementation is permitted to process a *rational* number by any other method that has essentially the same behavior but avoids loss of precision or overflow because of the coercion. If *w* and *d* are not supplied and the number has no exact decimal representation, for example `1/3`, some precision cutoff must be chosen by the implementation since only a finite number of digits may be printed.

If *arg* is a *complex* number or some non-numeric *object*, then it is printed using the format directive `~`*w*`D`, thereby printing it in decimal radix and a minimum field width of *w*.

`~F` binds ***print-escape*** to *false* and ***print-readably*** to *false*.

22.3.3.2 Tilde E: Exponential Floating-Point

The next *arg* is printed as a *float* in exponential notation.

The full form is `~`*w*`,`*d*`,`*e*`,`*k*`,`*overflowchar*`,`*padchar*`,`*exponentchar*`E`. The parameter *w* is the width of the field to be printed; *d* is the number of digits to print after the decimal point; *e* is the number of digits to use when printing the exponent; *k* is a scale factor that defaults to one (not zero).

Exactly *w* characters will be output. First, leading copies of the character *padchar* (which defaults to a space) are printed, if necessary, to pad the field on the left. If the *arg* is negative, then a minus sign is printed; if the *arg* is not negative, then a plus sign is printed if and only if the `@` modifier was supplied. Then a sequence of digits containing a single embedded decimal point is printed. The form of this sequence of digits depends on the scale factor *k*. If *k* is zero, then *d* digits are printed after the decimal point, and a single zero digit appears before the decimal point if the total field width will permit it. If *k* is positive, then it must be strictly less than *d*+2; *k* significant digits are printed before the decimal point, and *d*-*k*+1 digits are printed after the decimal point. If *k* is negative, then it must be strictly greater than -*d*; a single zero digit appears before the decimal point if the total field width will permit it, and after the decimal point are printed first -*k* zeros and then *d*+*k* significant digits. The printed fraction must be properly rounded. When rounding up and rounding down would produce printed values equidistant from the scaled value of *arg*, then the implementation is free to use either one. For example, printing the argument `637.5` using the format `~8,2E` may correctly produce either `6.37E+2` or `6.38E+2`.

Following the digit sequence, the exponent is printed. First the character parameter *exponentchar* is printed; if this parameter is omitted, then the *exponent marker* that **prin1** would use is printed, as determined from the type of the *float* and the current value of ***read-default-float-format***. Next, either a plus sign or a minus sign is printed, followed by *e* digits representing the power of ten by which the printed fraction must be multiplied to properly represent the rounded value of *arg*.

If it is impossible to print the value in the required format in a field of width *w*, possibly because *k* is too large or too small or because the exponent cannot be printed in *e* character positions, then one of two actions is taken. If the parameter *overflowchar* is supplied, then *w* copies of that parameter are printed instead of the scaled value of *arg*. If the *overflowchar* parameter is omitted, then the scaled value is printed using more than *w* characters, as many more as may be needed; if the problem is that *d* is too small for the supplied *k* or that *e* is too small, then a larger value is used for *d* or *e* as may be needed.

If the *w* parameter is omitted, then the field is of variable width. In effect a value is chosen for *w* in such a way that no leading pad characters need to be printed.

If the parameter *d* is omitted, then there is no constraint on the number of digits to appear. A value is chosen for *d* in such a way that as many digits as possible may be printed subject to the width constraint imposed by the parameter *w*, the constraint of the scale factor *k*, and the constraint that no trailing zero digits may appear in the fraction, except that if the fraction to be printed is zero then a single zero digit should appear after the decimal point.

If the parameter *e* is omitted, then the exponent is printed using the smallest number of digits necessary to represent its value.

If all of *w*, *d*, and *e* are omitted, then the effect is to print the value using ordinary free-format exponential-notation output; **prin1** uses a similar format for any non-zero number whose magnitude is less than 10^-3 or greater than or equal to 10^7. The only difference is that the `~E` directive always prints a plus or minus sign in front of the exponent, while **prin1** omits the plus sign if the exponent is non-negative.

If *arg* is a *rational* number, then it is coerced to be a *single float* and then printed. Alternatively, an implementation is permitted to process a *rational* number by any other method that has essentially the same behavior but avoids loss of precision or overflow because of the coercion. If *w* and *d* are unsupplied and the number has no exact decimal representation, for example `1/3`, some precision cutoff must be chosen by the implementation since only a finite number of digits may be printed.

If *arg* is a *complex* number or some non-numeric *object*, then it is printed using the format directive `~`*w*`D`, thereby printing it in decimal radix and a minimum field width of *w*.

`~E` binds ***print-escape*** to *false* and ***print-readably*** to *false*.

22.3.3.3 Tilde G: General Floating-Point

The next *arg* is printed as a *float* in either fixed-format or exponential notation as appropriate.

The full form is `~`*w*`,`*d*`,`*e*`,`*k*`,`*overflowchar*`,`*padchar*`,`*exponentchar*`G`. The format in which to print *arg* depends on the magnitude (absolute value) of the *arg*. Let *n* be an integer such that 10^*n*-1 <= |*arg*| < 10^*n*. Let *ee* equal *e*+2, or 4 if *e* is omitted. Let *ww* equal *w*-*ee*, or **nil** if *w* is omitted. If *d* is omitted, first let *q* be the number of digits needed to print *arg* with no loss of information and without leading or trailing zeros; then let *d* equal `(max` *q* `(min` *n* `7))`. Let *dd* equal *d*-*n*.

If 0 <= *dd* <= *d*, then *arg* is printed as if by the format directives

`~`*ww*`,`*dd*`,,`*overflowchar*`,`*padchar*`F~`*ee*`@T`

Note that the scale factor *k* is not passed to the `~F` directive. For all other values of *dd*, *arg* is printed as if by the format directive

`~`*w*`,`*d*`,`*e*`,`*k*`,`*overflowchar*`,`*padchar*`,`*exponentchar*`E`

In either case, an @ modifier is supplied to the ~F or ~E directive if and only if one was supplied to the ~G directive.

~G binds ***print-escape*** to *false* and ***print-readably*** to *false*.

22.3.3.4 Tilde Dollarsign: Monetary Floating-Point

The next *arg* is printed as a *float* in fixed-format notation.

The full form is ~*d*,*n*,*w*,*padchar*$. The parameter *d* is the number of digits to print after the decimal point (default value 2); *n* is the minimum number of digits to print before the decimal point (default value 1); *w* is the minimum total width of the field to be printed (default value 0).

First padding and the sign are output. If the *arg* is negative, then a minus sign is printed; if the *arg* is not negative, then a plus sign is printed if and only if the @ modifier was supplied. If the : modifier is used, the sign appears before any padding, and otherwise after the padding. If *w* is supplied and the number of other characters to be output is less than *w*, then copies of *padchar* (which defaults to a space) are output to make the total field width equal *w*. Then *n* digits are printed for the integer part of *arg*, with leading zeros if necessary; then a decimal point; then *d* digits of fraction, properly rounded.

If the magnitude of *arg* is so large that more than *m* digits would have to be printed, where *m* is the larger of *w* and 100, then an implementation is free, at its discretion, to print the number using exponential notation instead, as if by the directive ~*w*,*q*,,,,*padchar*E, where *w* and *padchar* are present or omitted according to whether they were present or omitted in the ~$ directive, and where *q*=*d*+*n*-1, where *d* and *n* are the (possibly default) values given to the ~$ directive.

If *arg* is a *rational* number, then it is coerced to be a *single float* and then printed. Alternatively, an implementation is permitted to process a *rational* number by any other method that has essentially the same behavior but avoids loss of precision or overflow because of the coercion.

If *arg* is a *complex* number or some non-numeric *object*, then it is printed using the format directive ~*w*D, thereby printing it in decimal radix and a minimum field width of *w*.

~$ binds ***print-escape*** to *false* and ***print-readably*** to *false*.

22.3.4 FORMAT Printer Operations

22.3.4.1 Tilde A: Aesthetic

An *arg*, any *object*, is printed without escape characters (as by **princ**). If *arg* is a *string*, its *characters* will be output verbatim. If *arg* is **nil** it will be printed as **nil**; the *colon* modifier (~:A) will cause an *arg* of **nil** to be printed as (), but if *arg* is a composite structure, such as a *list* or *vector*, any contained occurrences of **nil** will still be printed as **nil**.

~*mincol*A inserts spaces on the right, if necessary, to make the width at least *mincol* columns. The @ modifier causes the spaces to be inserted on the left rather than the right.

~*mincol*,*colinc*,*minpad*,*padchar*A is the full form of ~A, which allows control of the padding. The *string* is padded on the right (or on the left if the @ modifier is used) with at least *minpad* copies of *padchar*; padding characters are then inserted *colinc* characters at a time until the total width is at least *mincol*. The defaults are 0 for *mincol* and *minpad*, 1 for *colinc*, and the space character for *padchar*.

~A binds ***print-escape*** to *false*, and ***print-readably*** to *false*.

22.3.4.2 Tilde S: Standard

This is just like ~A, but *arg* is printed with escape characters (as by **prin1** rather than princ). The output is therefore suitable for input to **read**. ~S accepts all the arguments and modifiers that ~A does.

~S binds ***print-escape*** to **t**.

22.3.4.3 Tilde W: Write

An argument, any *object*, is printed obeying every printer control variable (as by **write**). In addition, ~W interacts correctly with depth abbreviation, by not resetting the depth counter to zero. ~W does not accept parameters. If given the *colon* modifier, ~W binds ***print-pretty*** to *true*. If given the *at-sign* modifier, ~W binds ***print-level*** and ***print-length*** to **nil**.

~W provides automatic support for the detection of circularity and sharing. If the *value* of ***print-circle*** is not **nil** and ~W is applied to an argument that is a circular (or shared) reference, an appropriate #*n*# marker is inserted in the output instead of printing the argument.

22.3.5 FORMAT Pretty Printer Operations

The following constructs provide access to the *pretty printer*:

22.3.5.1 Tilde Underscore: Conditional Newline

Without any modifiers, ~_ is the same as (pprint-newline :linear). ~@_ is the same as (pprint-newline :miser). ~:_ is the same as (pprint-newline :fill). ~:@_ is the same as (pprint-newline :mandatory).

22.3.5.2 Tilde Less-Than-Sign: Logical Block

~<...~:>

If ~:> is used to terminate a ~<...~>, the directive is equivalent to a call to **pprint-logical-block**. The argument corresponding to the ~<...~:> directive is treated in the same way as the *list* argument to **pprint-logical-block**, thereby providing automatic support for non-*list* arguments and the detection of circularity, sharing, and depth abbreviation. The portion of the *control-string* nested within the ~<...~:> specifies the :prefix (or :per-line-prefix), :suffix, and body of the **pprint-logical-block**.

The *control-string* portion enclosed by ~<...~:> can be divided into segments ~<*prefix*~;*body*~;*suffix*~:> by ~; directives. If the first section is terminated by ~@;, it specifies a per-line prefix rather than a simple prefix. The *prefix* and *suffix* cannot contain format directives. An error is signaled if either the prefix or suffix fails to be a constant string or if the enclosed portion is divided into more than three segments.

If the enclosed portion is divided into only two segments, the *suffix* defaults to the null string. If the enclosed portion consists of only a single segment, both the *prefix* and the *suffix* default to the null string. If the *colon* modifier is used (i.e., ~:<...~:>), the *prefix* and *suffix* default to "(" and ")" (respectively) instead of the null string.

The body segment can be any arbitrary *format string*. This *format string* is applied to the elements of the list corresponding to the `~<...~:>` directive as a whole. Elements are extracted from this list using **pprint-pop**, thereby providing automatic support for malformed lists, and the detection of circularity, sharing, and length abbreviation. Within the body segment, `~^` acts like **pprint-exit-if-list-exhausted**.

`~<...~:>` supports a feature not supported by **pprint-logical-block**. If `~:@>` is used to terminate the directive (i.e., `~<...~:@>`), then a fill-style conditional newline is automatically inserted after each group of blanks immediately contained in the body (except for blanks after a <Newline> directive). This makes it easy to achieve the equivalent of paragraph filling.

If the *at-sign* modifier is used with `~<...~:>`, the entire remaining argument list is passed to the directive as its argument. All of the remaining arguments are always consumed by `~@<...~:>`, even if they are not all used by the *format string* nested in the directive. Other than the difference in its argument, `~@<...~:>` is exactly the same as `~<...~:>` except that circularity detection is not applied if `~@<...~:>` is encountered at top level in a *format string*. This ensures that circularity detection is applied only to data lists, not to *format argument lists*.

`" . #n#"` is printed if circularity or sharing has to be indicated for its argument as a whole.

To a considerable extent, the basic form of the directive `~<...~>` is incompatible with the dynamic control of the arrangement of output by `~W`, `~_`, `~<...~:>`, `~I`, and `~:T`. As a result, an error is signaled if any of these directives is nested within `~<...~>`. Beyond this, an error is also signaled if the `~<...~:;...~>` form of `~<...~>` is used in the same *format string* with `~W`, `~_`, `~<...~:>`, `~I`, or `~:T`.

See also Section 22.3.6.2 (Tilde Less-Than-Sign: Justification).

22.3.5.3 Tilde I: Indent

~*n*`I` is the same as `(pprint-indent :block n)`.

~*n*`:I` is the same as `(pprint-indent :current n)`. In both cases, *n* defaults to zero, if it is omitted.

22.3.5.4 Tilde Slash: Call Function

`~/`*name*`/`

User defined functions can be called from within a format string by using the directive `~/`*name*`/`. The *colon* modifier, the *at-sign* modifier, and arbitrarily many parameters can be specified with the `~/`*name*`/` directive. *name* can be any arbitrary string that does not contain a "/". All of the characters in *name* are treated as if they were upper case. If *name* contains a single *colon* (`:`) or double *colon* (`::`), then everything up to but not including the first `":"` or `"::"` is taken to be a *string* that names a *package*. Everything after the first `":"` or `"::"` (if any) is taken to be a *string* that names a `symbol`. The function corresponding to a `~/name/` directive is obtained by looking up the *symbol* that has the indicated name in the indicated *package*. If *name* does not contain a `":"` or `"::"`, then the whole *name* string is looked up in the `COMMON-LISP-USER` package.

When a `~/name/` directive is encountered, the indicated function is called with four or more arguments. The first four arguments are: the output stream, the *format argument* corresponding to the directive, a *generalized boolean* that is *true* if the *colon* modifier was used, and a *generalized boolean* that is *true* if the *at-sign* modifier was used. The remaining arguments consist of any parameters specified with the directive. The function should print the argument appropriately. Any values returned by the function are ignored.

The three *functions* **pprint-linear**, **pprint-fill**, and **pprint-tabular** are specifically designed so that they can be called by `~/.../` (i.e., `~/pprint-linear/`, `~/pprint-fill/`, and `~/pprint-tabular/`). In particular they take *colon* and *at-sign* arguments.

22.3.6 FORMAT Layout Control

22.3.6.1 Tilde T: Tabulate

This spaces over to a given column. ~*colnum*,*colinc*T will output sufficient spaces to move the cursor to column *colnum*. If the cursor is already at or beyond column *colnum*, it will output spaces to move it to column *colnum*+*k***colinc* for the smallest positive integer *k* possible, unless *colinc* is zero, in which case no spaces are output if the cursor is already at or beyond column *colnum*. *colnum* and *colinc* default to 1.

If for some reason the current absolute column position cannot be determined by direct inquiry, **format** may be able to deduce the current column position by noting that certain directives (such as ~%, or ~&, or ~A with the argument being a string containing a newline) cause the column position to be reset to zero, and counting the number of characters emitted since that point. If that fails, **format** may attempt a similar deduction on the riskier assumption that the destination was at column zero when **format** was invoked. If even this heuristic fails or is implementationally inconvenient, at worst the ~T operation will simply output two spaces.

~@T performs relative tabulation. ~*colrel*,*colinc*@T outputs *colrel* spaces and then outputs the smallest non-negative number of additional spaces necessary to move the cursor to a column that is a multiple of *colinc*. For example, the directive ~3,8@T outputs three spaces and then moves the cursor to a "standard multiple-of-eight tab stop" if not at one already. If the current output column cannot be determined, however, then *colinc* is ignored, and exactly *colrel* spaces are output.

If the *colon* modifier is used with the ~T directive, the tabbing computation is done relative to the horizontal position where the section immediately containing the directive begins, rather than with respect to a horizontal position of zero. The numerical parameters are both interpreted as being in units of *ems* and both default to 1. ~*n*,*m*:T is the same as (pprint-tab :section *n m*). ~*n*,*m*:@T is the same as (pprint-tab :section-relative *n m*).

22.3.6.2 Tilde Less-Than-Sign: Justification

~*mincol*,*colinc*,*minpad*,*padchar*<*str*~>

This justifies the text produced by processing *str* within a field at least *mincol* columns wide. *str* may be divided up into segments with ~;, in which case the spacing is evenly divided between the text segments.

With no modifiers, the leftmost text segment is left justified in the field, and the rightmost text segment is right justified. If there is only one text element, as a special case, it is right justified. The : modifier causes spacing to be introduced before the first text segment; the @ modifier causes spacing to be added after the last. The *minpad* parameter (default 0) is the minimum number of padding characters to be output between each segment. The padding character is supplied by *padchar*, which defaults to the space character. If the total width needed to satisfy these constraints is greater than *mincol*, then the width used is *mincol*+*k***colinc* for the smallest possible non-negative integer value *k*. *colinc* defaults to 1, and *mincol* defaults to 0.

Note that *str* may include **format** directives. All the clauses in *str* are processed in order; it is the resulting pieces of text that are justified.

The ~^ directive may be used to terminate processing of the clauses prematurely, in which case only the completely processed clauses are justified.

If the first clause of a ~< is terminated with ~:; instead of ~;, then it is used in a special way. All of the clauses are processed (subject to ~^, of course), but the first one is not used in performing the spacing and padding. When the padded result has been determined, then if it will fit on the current line of output, it is output, and the text for the first clause is discarded. If, however, the padded text will not fit on the current line, then the text segment for the first clause is output before the padded text. The first clause ought to contain a newline (such as a ~% directive).

The first clause is always processed, and so any arguments it refers to will be used; the decision is whether to use the resulting segment of text, not whether to process the first clause. If the `~:;` has a prefix parameter *n*, then the padded text must fit on the current line with *n* character positions to spare to avoid outputting the first clause's text. For example, the control string

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

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

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

If the second argument is not supplied, then **format** uses the line width of the *destination* output stream. If this cannot be determined (for example, when producing a *string* result), then **format** uses `72` as the line length.

See also Section 22.3.5.2 (Tilde Less-Than-Sign: Logical Block).

22.3.6.3 Tilde Greater-Than-Sign: End of Justification

`~>` terminates a `~<`. The consequences of using it elsewhere are undefined.

22.3.7 FORMAT Control-Flow Operations

22.3.7.1 Tilde Asterisk: Go-To

The next *arg* is ignored. `~`*n*`*` ignores the next *n* arguments.

`~:*` backs up in the list of arguments so that the argument last processed will be processed again. `~`*n*`:*` backs up *n* arguments.

When within a `~{` construct (see below), the ignoring (in either direction) is relative to the list of arguments being processed by the iteration.

`~`*n*`@*` goes to the *n*th *arg*, where 0 means the first one; *n* defaults to 0, so `~@*` goes back to the first *arg*. Directives after a `~`*n*`@*` will take arguments in sequence beginning with the one gone to. When within a `~{` construct, the "goto" is relative to the list of arguments being processed by the iteration.

22.3.7.2 Tilde Left-Bracket: Conditional Expression

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

This is a set of control strings, called *clauses*, one of which is chosen and used. The clauses are separated by `~;` and the construct is terminated by `~]`. For example,

```
"~[Siamese~;Manx~;Persian~] Cat"
```

The *arg*th clause is selected, where the first clause is number 0. If a prefix parameter is given (as `~`*n*`[`), then the parameter is used instead of an argument. If *arg* is out of range then no clause is selected and no error is signaled. After the selected alternative has been processed, the control string continues after the `~]`.

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

```
"~[Siamese~;Manx~;Persian~:;Alley~] Cat"
```

`~:[`*alternative*`~;`*consequent*`~]` selects the *alternative* control string if *arg* is *false*, and selects the *consequent* control string otherwise.

`~@[`*consequent*`~]` tests the argument. If it is *true*, then the argument is not used up by the `~[` command but remains as the next one to be processed, and the one clause *consequent* is processed. If the *arg* is *false*, then the argument is used up, and the clause is not processed. The clause therefore should normally use exactly one argument, and may expect it to be *non-nil*. For example:

```
 (setq *print-level* nil *print-length* 5)
 (format nil
        "~@[ print level = ~D~]~@[ print length = ~D~]"
        *print-level* *print-length*)
=>   " print length = 5"
```

Note also that

```
 (format stream "...~@[str~]..." ...)
==  (format stream "...~:[~;~:*str~]..." ...)
```

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

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

22.3.7.3 Tilde Right-Bracket: End of Conditional Expression

`~]` terminates a `~[`. The consequences of using it elsewhere are undefined.

22.3.7.4 Tilde Left-Brace: Iteration

`~{`*str*`~}`

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

For example:

```
 (format nil "The winners are:~{ ~S~}."
         ’(fred harry jill))
=>  "The winners are: FRED HARRY JILL."
 (format nil "Pairs:~{ <~S,~S>~}."
         ’(a 1 b 2 c 3))
=>  "Pairs: <A,1> <B,2> <C,3>."
```

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

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

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

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

If the iteration is terminated before all the remaining arguments are consumed, then any arguments not processed by the iteration remain to be processed by any directives following the iteration construct.

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

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

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

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

```
    (apply #'format stream string arguments)
 ==  (format stream "~1{~:}" string arguments)
```

This will use `string` as a formatting string. The ~1{ says it will be processed at most once, and the ~:} says it will be processed at least once. Therefore it is processed exactly once, using `arguments` as the arguments. This case may be handled more clearly by the ~? directive, but this general feature of ~{ is more powerful than ~?.

22.3.7.5 Tilde Right-Brace: End of Iteration

~} terminates a ~{. The consequences of using it elsewhere are undefined.

22.3.7.6 Tilde Question-Mark: Recursive Processing

The next *arg* must be a *format control*, and the one after it a *list*; both are consumed by the ~? directive. The two are processed as a *control-string*, with the elements of the *list* as the arguments. Once the recursive processing has been finished, the processing of the control string containing the ~? directive is resumed. Example:

```
 (format nil "~? ~D" "<~A ~D>" '("Foo" 5) 7) =>  "<Foo 5> 7"
 (format nil "~? ~D" "<~A ~D>" '("Foo" 5 14) 7) =>  "<Foo 5> 7"
```

Note that in the second example three arguments are supplied to the *format string* "<~A ~D>", but only two are processed and the third is therefore ignored.

With the @ modifier, only one *arg* is directly consumed. The *arg* must be a *string*; it is processed as part of the control string as if it had appeared in place of the ~@? construct, and any directives in the recursively processed control string may consume arguments of the control string containing the ~@? directive. Example:

```
(format nil "~@? ~D" "<~A ~D>" "Foo" 5 7) =>  "<Foo 5> 7"
(format nil "~@? ~D" "<~A ~D>" "Foo" 5 14 7) =>  "<Foo 5> 14"
```

22.3.8 FORMAT Miscellaneous Operations

22.3.8.1 Tilde Left-Paren: Case Conversion

`~(`*str*`~)`

The contained control string *str* is processed, and what it produces is subject to case conversion.

With no flags, every *uppercase character* is converted to the corresponding *lowercase character*.

`~:(` capitalizes all words, as if by **string-capitalize**.

`~@(` capitalizes just the first word and forces the rest to lower case.

`~:@(` converts every lowercase character to the corresponding uppercase character.

In this example `~@(` is used to cause the first word produced by `~@R` to be capitalized:

```
 (format nil "~@R ~(~@R~)" 14 14)
=>  "XIV xiv"
 (defun f (n) (format nil "~@(~R~) error~:P detected." n)) =>  F
 (f 0) =>  "Zero errors detected."
 (f 1) =>  "One error detected."
 (f 23) =>  "Twenty-three errors detected."
```

When case conversions appear nested, the outer conversion dominates, as illustrated in the following example:

```
(format nil "~@(how is ~:(BOB SMITH~)?~)")
=>  "How is bob smith?"
NOT=>  "How is Bob Smith?"
```

22.3.8.2 Tilde Right-Paren: End of Case Conversion

`~)` terminates a `~(`. The consequences of using it elsewhere are undefined.

22.3.8.3 Tilde P: Plural

If *arg* is not **eql** to the integer `1`, a lowercase `s` is printed; if *arg* is **eql** to `1`, nothing is printed. If *arg* is a floating-point `1.0`, the `s` is printed.

`~:P` does the same thing, after doing a `~:*` to back up one argument; that is, it prints a lowercase `s` if the previous argument was not `1`.

`~@P` prints `y` if the argument is `1`, or `ies` if it is not. `~:@P` does the same thing, but backs up first.

```
(format nil "~D tr~:@P/~D win~:P" 7 1) =>  "7 tries/1 win"
(format nil "~D tr~:@P/~D win~:P" 1 0) =>  "1 try/0 wins"
(format nil "~D tr~:@P/~D win~:P" 1 3) =>  "1 try/3 wins"
```

22.3.9 FORMAT Miscellaneous Pseudo-Operations

22.3.9.1 Tilde Semicolon: Clause Separator

This separates clauses in ~[and ~< constructs. The consequences of using it elsewhere are undefined.

22.3.9.2 Tilde Circumflex: Escape Upward

~^

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

```
 (setq donestr "Done.~^ ~D warning~:P.~^ ~D error~:P.")
=>  "Done.~^ ~D warning~:P.~^ ~D error~:P."
 (format nil donestr) =>  "Done."
 (format nil donestr 3) =>  "Done. 3 warnings."
 (format nil donestr 1 5) =>  "Done. 1 warning. 5 errors."
```

If a prefix parameter is given, then termination occurs if the parameter is zero. (Hence ~^ is equivalent to ~#^.) If two parameters are given, termination occurs if they are equal. If three parameters are given, termination occurs if the first is less than or equal to the second and the second is less than or equal to the third. Of course, this is useless if all the prefix parameters are constants; at least one of them should be a # or a V parameter.

If ~^ is used within a ~:{ construct, then it terminates the current iteration step because in the standard case it tests for remaining arguments of the current step only; the next iteration step commences immediately. ~:^ is used to terminate the iteration process. ~:^ may be used only if the command it would terminate is ~:{ or ~:@{. The entire iteration process is terminated if and only if the sublist that is supplying the arguments for the current iteration step is the last sublist in the case of ~:{, or the last **format** argument in the case of ~:@{. ~:^ is not equivalent to ~#:^; the latter terminates the entire iteration if and only if no arguments remain for the current iteration step. For example:

```
 (format nil "~:{ ~@?~:^ ...~} " '(("a") ("b"))) =>  "a...b"
```

If ~^ appears within a control string being processed under the control of a ~? directive, but not within any ~{ or ~< construct within that string, then the string being processed will be terminated, thereby ending processing of the ~? directive. Processing then continues within the string containing the ~? directive at the point following that directive.

If ~^ appears within a ~[or ~(construct, then all the commands up to the ~^ are properly selected or case-converted, the ~[or ~(processing is terminated, and the outward search continues for a ~{ or ~< construct to be terminated. For example:

```
 (setq tellstr "~@(~@[~R~]~^ ~A!~)")
=>  "~@(~@[~R~]~^ ~A!~)"
 (format nil tellstr 23) =>  "Twenty-three!"
 (format nil tellstr nil "losers") =>  " Losers!"
 (format nil tellstr 23 "losers") =>  "Twenty-three losers!"
```

Following are examples of the use of ~^ within a ~< construct.

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

22.3.9.3 Tilde Newline: Ignored Newline

Tilde immediately followed by a *newline* ignores the *newline* and any following non-newline *whitespace*[1] characters. With a `:`, the *newline* is ignored, but any following *whitespace*[1] is left in place. With an `@`, the *newline* is left in place, but any following *whitespace*[1] is ignored. For example:

```
 (defun type-clash-error (fn nargs argnum right-type wrong-type)
   (format *error-output*
           "~&~S requires its ~:[~:R~;~*~]~
           argument to be of type ~S,~%but it was called ~
           with an argument of type ~S.~%"
           fn (eql nargs 1) argnum right-type wrong-type))
 (type-clash-error 'aref nil 2 'integer 'vector)  prints:
AREF requires its second argument to be of type INTEGER,
but it was called with an argument of type VECTOR.
NIL
 (type-clash-error 'car 1 1 'list 'short-float)  prints:
CAR requires its argument to be of type LIST,
but it was called with an argument of type SHORT-FLOAT.
NIL
```

Note that in this example newlines appear in the output only as specified by the `~&` and `~%` directives; the actual newline characters in the control string are suppressed because each is preceded by a tilde.

22.3.10 Additional Information about FORMAT Operations

22.3.10.1 Nesting of FORMAT Operations

The case-conversion, conditional, iteration, and justification constructs can contain other formatting constructs by bracketing them. These constructs must nest properly with respect to each other. For example, it is not legitimate to put the start of a case-conversion construct in each arm of a conditional and the end of the case-conversion construct outside the conditional:

```
 (format nil "~:[abc~:@(def~;ghi~
:@(jkl~]mno~)" x) ;Invalid!
```

This notation is invalid because the `~[...~;...~]` and `~(...~)` constructs are not properly nested.

The processing indirection caused by the `~?` directive is also a kind of nesting for the purposes of this rule of proper nesting. It is not permitted to start a bracketing construct within a string processed under control of a `~?` directive and end the construct at some point after the `~?` construct in the string containing that construct, or vice versa. For example, this situation is invalid:

```
 (format nil "~@?ghi~)" "abc~@(def") ;Invalid!
```

This notation is invalid because the `~?` and `~(...~)` constructs are not properly nested.

22.3.10.2 Missing and Additional FORMAT Arguments

The consequences are undefined if no *arg* remains for a directive requiring an argument. However, it is permissible for one or more *args* to remain unprocessed by a directive; such *args* are ignored.

22.3.10.3 Additional FORMAT Parameters

The consequences are undefined if a format directive is given more parameters than it is described here as accepting.

22.3.10.4 Undefined FORMAT Modifier Combinations

The consequences are undefined if *colon* or *at-sign* modifiers are given to a directive in a combination not specifically described here as being meaningful.

22.3.11 Examples of FORMAT

```
 (format nil "foo") =>  "foo"
 (setq x 5) =>  5
 (format nil "The answer is ~D." x) =>  "The answer is 5."
 (format nil "The answer is ~3D." x) =>  "The answer is   5."
 (format nil "The answer is ~3,'0D." x) =>  "The answer is 005."
 (format nil "The answer is ~:D." (expt 47 x))
=>  "The answer is 229,345,007."
 (setq y "elephant") =>  "elephant"
 (format nil "Look at the ~A!" y) =>  "Look at the elephant!"
 (setq n 3) =>  3
 (format nil "~D item~:P found." n) =>  "3 items found."
 (format nil "~R dog~:[s are~; is~] here." n (= n 1))
=>  "three dogs are here."
 (format nil "~R dog~:*~[s are~; is~:;s are~] here." n)
=>  "three dogs are here."
 (format nil "Here ~[are~;is~:;are~] ~:*~R pupp~:@P." n)
=>  "Here are three puppies."

 (defun foo (x)
   (format nil "~6,2F|~6,2,1,'*F|~6,2,,'?F|~6F|~,2F|~F"
           x x x x x x)) =>  FOO
 (foo 3.14159)  =>  "  3.14| 31.42|  3.14|3.1416|3.14|3.14159"
 (foo -3.14159) =>  " -3.14|-31.42| -3.14|-3.142|-3.14|-3.14159"
 (foo 100.0)    =>  "100.00|******|100.00| 100.0|100.00|100.0"
 (foo 1234.0)   =>  "1234.00|******|??????|1234.0|1234.00|1234.0"
 (foo 0.006)    =>  "  0.01|  0.06|  0.01| 0.006|0.01|0.006"

 (defun foo (x)
    (format nil
           "~9,2,1,,'*E|~10,3,2,2,'?,,'$E|~
            ~9,3,2,-2,'%@E|~9,2E"
           x x x x))
 (foo 3.14159)  =>  "  3.14E+0| 31.42$-01|+.003E+03|  3.14E+0"
 (foo -3.14159) =>  " -3.14E+0|-31.42$-01|-.003E+03| -3.14E+0"
 (foo 1100.0)   =>  "  1.10E+3| 11.00$+02|+.001E+06|  1.10E+3"
 (foo 1100.0L0) =>  "  1.10L+3| 11.00$+02|+.001L+06|  1.10L+3"
 (foo 1.1E13)   =>  "*********| 11.00$+12|+.001E+16| 1.10E+13"
 (foo 1.1L120)  =>  "*********|??????????|%%%%%%%%%|1.10L+120"
 (foo 1.1L1200) =>  "*********|??????????|%%%%%%%%%|1.10L+1200"
```

As an example of the effects of varying the scale factor, the code

```
 (dotimes (k 13)
   (format t "~%Scale factor ~2D: |~13,6,2,VE|"
           (- k 5) (- k 5) 3.14159))
```

produces the following output:

```
Scale factor -5: | 0.000003E+06|
Scale factor -4: | 0.000031E+05|
Scale factor -3: | 0.000314E+04|
Scale factor -2: | 0.003142E+03|
Scale factor -1: | 0.031416E+02|
Scale factor  0: | 0.314159E+01|
Scale factor  1: | 3.141590E+00|
Scale factor  2: | 31.41590E-01|
Scale factor  3: | 314.1590E-02|
Scale factor  4: | 3141.590E-03|
Scale factor  5: | 31415.90E-04|
Scale factor  6: | 314159.0E-05|
Scale factor  7: | 3141590.E-06|
```

```
 (defun foo (x)
   (format nil "~9,2,1,,'*G|~9,3,2,3,'?,,'$G|~9,3,2,0,'%G|~9,2G"
          x x x x))
 (foo 0.0314159) =>  "  3.14E-2|314.2$-04|0.314E-01|  3.14E-2"
 (foo 0.314159)  =>  "  0.31   |0.314    |0.314    | 0.31    "
 (foo 3.14159)   =>  "   3.1   | 3.14    | 3.14    |  3.1    "
 (foo 31.4159)   =>  "   31.   | 31.4    | 31.4    |  31.    "
 (foo 314.159)   =>  "  3.14E+2| 314.    | 314.    |  3.14E+2"
 (foo 3141.59)   =>  "  3.14E+3|314.2$+01|0.314E+04|  3.14E+3"
 (foo 3141.59L0) =>  "  3.14L+3|314.2$+01|0.314L+04|  3.14L+3"
 (foo 3.14E12)   =>  "*********|314.0$+10|0.314E+13| 3.14E+12"
 (foo 3.14L120)  =>  "*********|?????????|%%%%%%%%%|3.14L+120"
 (foo 3.14L1200) =>  "*********|?????????|%%%%%%%%%|3.14L+1200"
```

```
 (format nil "~10<foo~;bar~>")   =>  "foo    bar"
 (format nil "~10:<foo~;bar~>")  =>  "  foo  bar"
 (format nil "~10<foobar~>")     =>  "    foobar"
 (format nil "~10:<foobar~>")    =>  "    foobar"
 (format nil "~10:@<foo~;bar~>") =>  "  foo bar "
 (format nil "~10@<foobar~>")    =>  "foobar    "
 (format nil "~10:@<foobar~>")   =>  "  foobar  "
```

```
  (FORMAT NIL "Written to ~A." #P"foo.bin")
  =>  "Written to foo.bin."
```

22.3.12 Notes about FORMAT

Formatted output is performed not only by **format**, but by certain other functions that accept a *format control* the way **format** does. For example, error-signaling functions such as **cerror** accept *format controls*.

Note that the meaning of **nil** and **t** as destinations to **format** are different than those of **nil** and **t** as *stream designators*.

The **~^** should appear only at the beginning of a **~<** clause, because it aborts the entire clause in which it appears (as well as all following clauses).